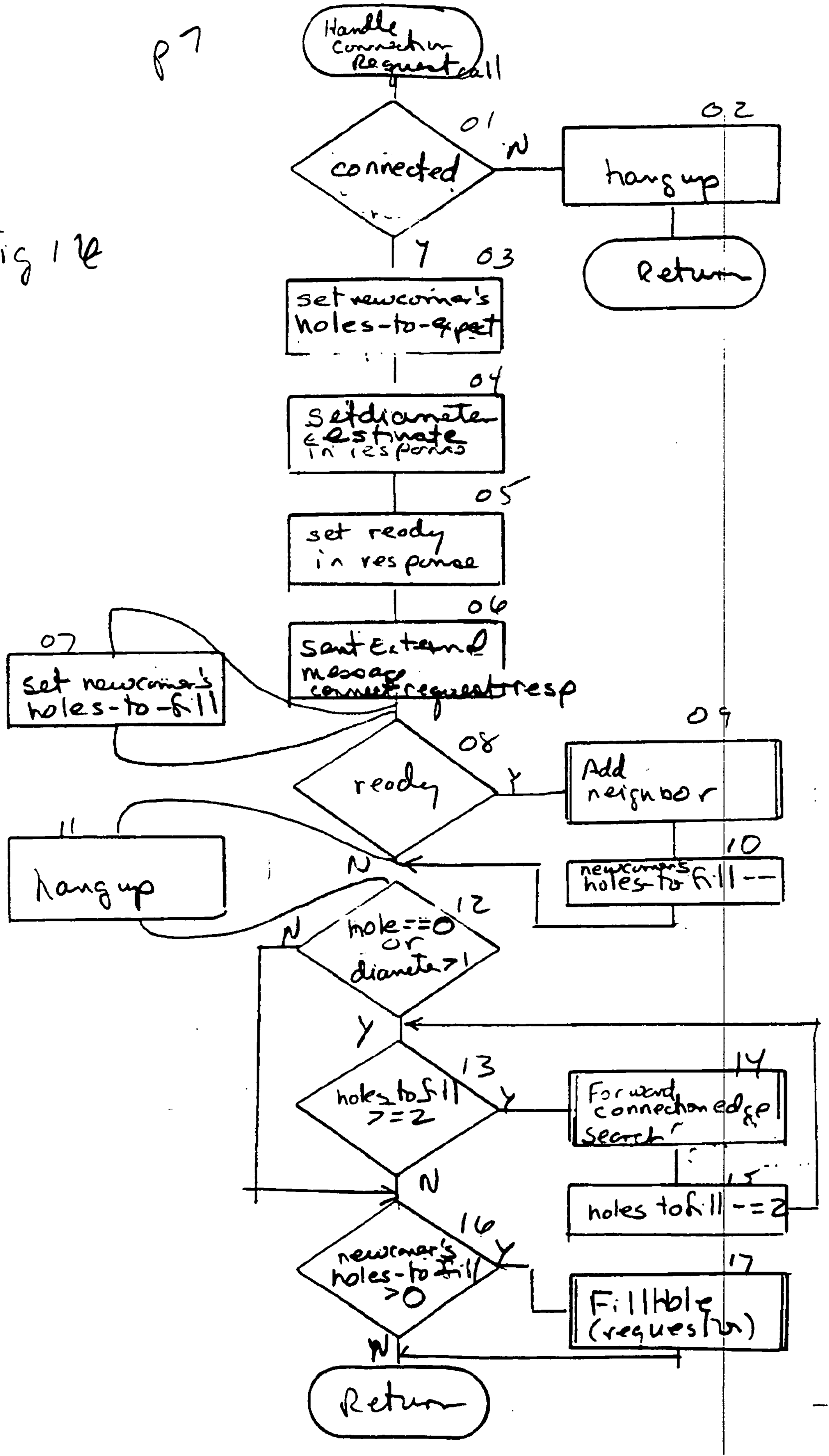
p 7

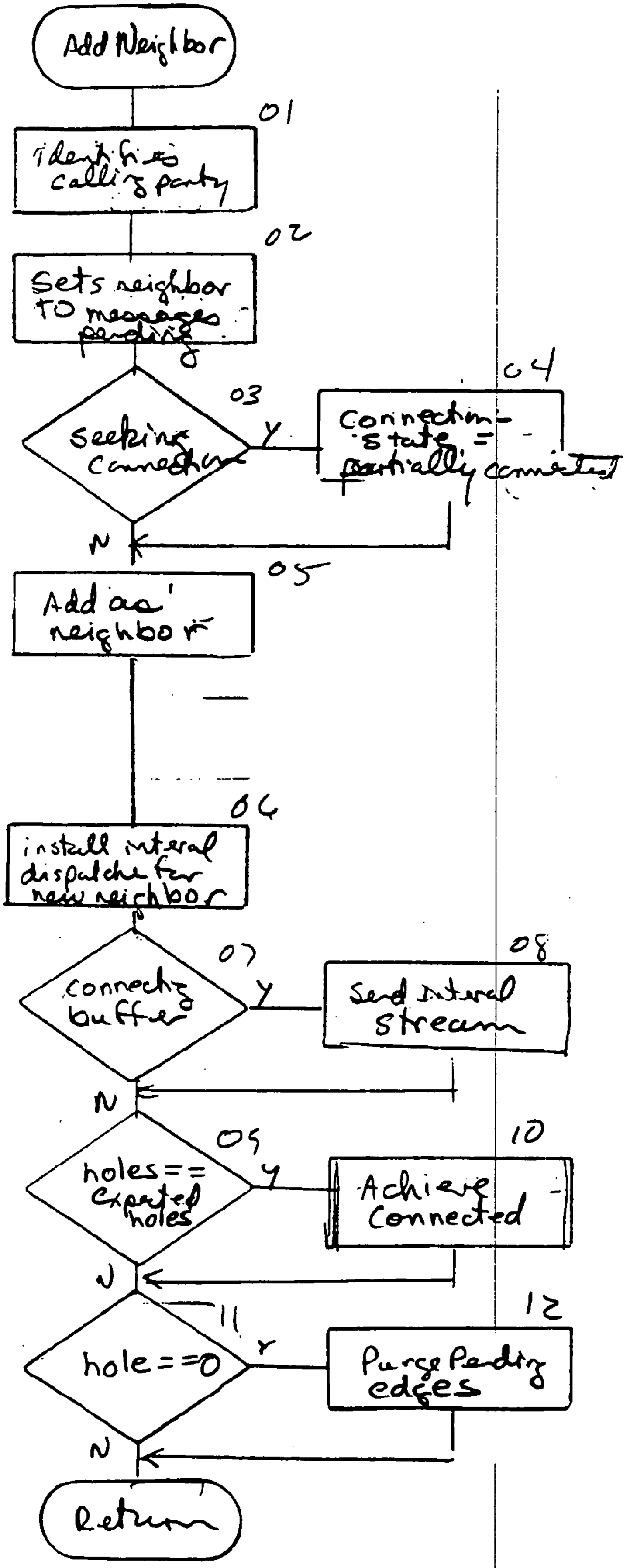
Fig 1e



Handle Connection Request call
01
connected
N
02
hang up
Return
Y
03
set newcomer's holes-to-expect
04
set diameter estimate in response
05
set ready in response
06
Send Extend message connect request resp
07
set newcomer's holes-to-fill
08
ready
Y
09
Add neighbor
10
newcomer's holes-to-fill --
11
hang up
N
12
hole==0 or diameter>1
N
Y
13
holes to fill >=2
Y
14
Forward connection edge search
15
holes to fill -=2
N
16
newcomer's holes-to-fill >0
Y
17
FillHole (requestor)
N
Return

p9

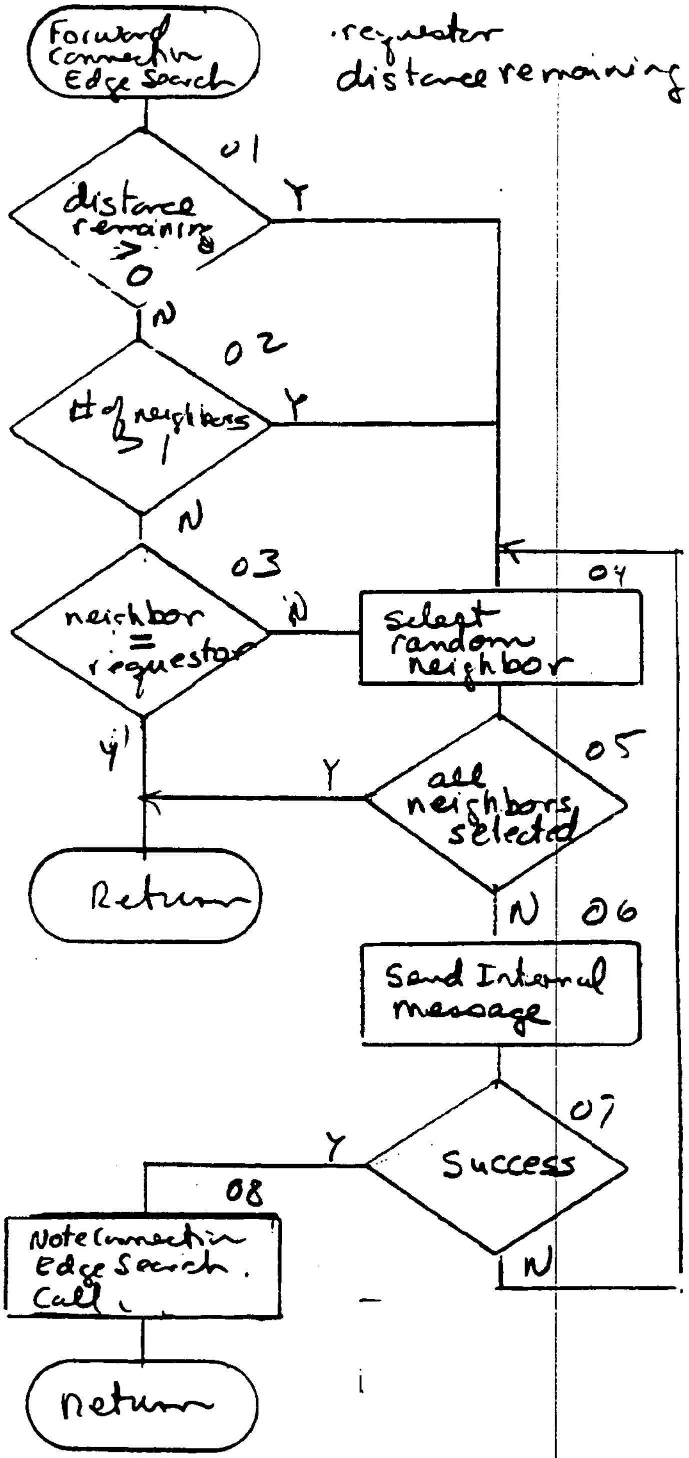
Fig 17


Add Neighbor
01
Identifies calling party
02
Sets neighbor TO messages pending
03
seeking Connection
Y
04
Connection-state = partially connected
N
05
Add as neighbor
06
install interal dispatche for new neighbor
07
connecting buffer
Y
08
Send Interal stream
N
09
holes== expected holes
Y
10
Achieve Connected
N
11
hole==0
Y
12
Purge Pending edges
N
Return

15

requester
distance remaining

Fig 18


Forward Connection Edge Search
01
distance remaining > 0
Y
N
02
# of neighbors > 1
Y
N
03
neighbor = requestor
N
Y
04
select random neighbor
05
all neighbors selected
Y
N
06
Send Internal message
07
Success
Y
N
08
Note Connection Edge Search Call
Return
Return

p 8

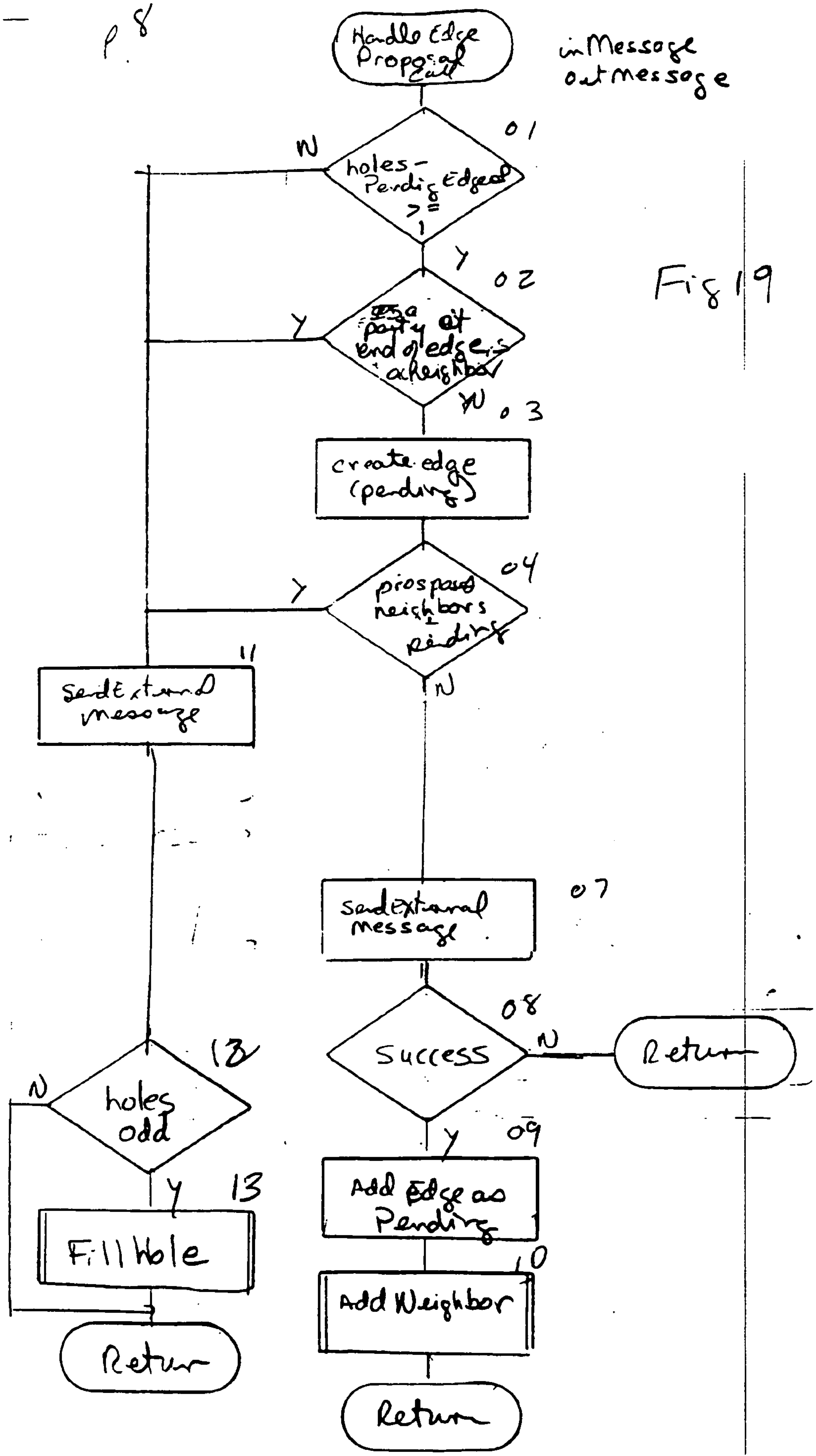

Handle Edge Proposal Call
inMessage
outmessage
01
holes-Pending Edges >= 1
N
Y
02
is a party at end of edge is a neighbor
Y
N
03
create edge (pending)
04
prospect neighbors pending
Y
N
11
sendExternal message
07
sendExternal message
08
Success
N
Return
Y
09
Add Edge as Pending
10
Add Neighbor
Return
12
holes odd
N
Y
13
Fill hole
Return

Fig 19

Fig 20

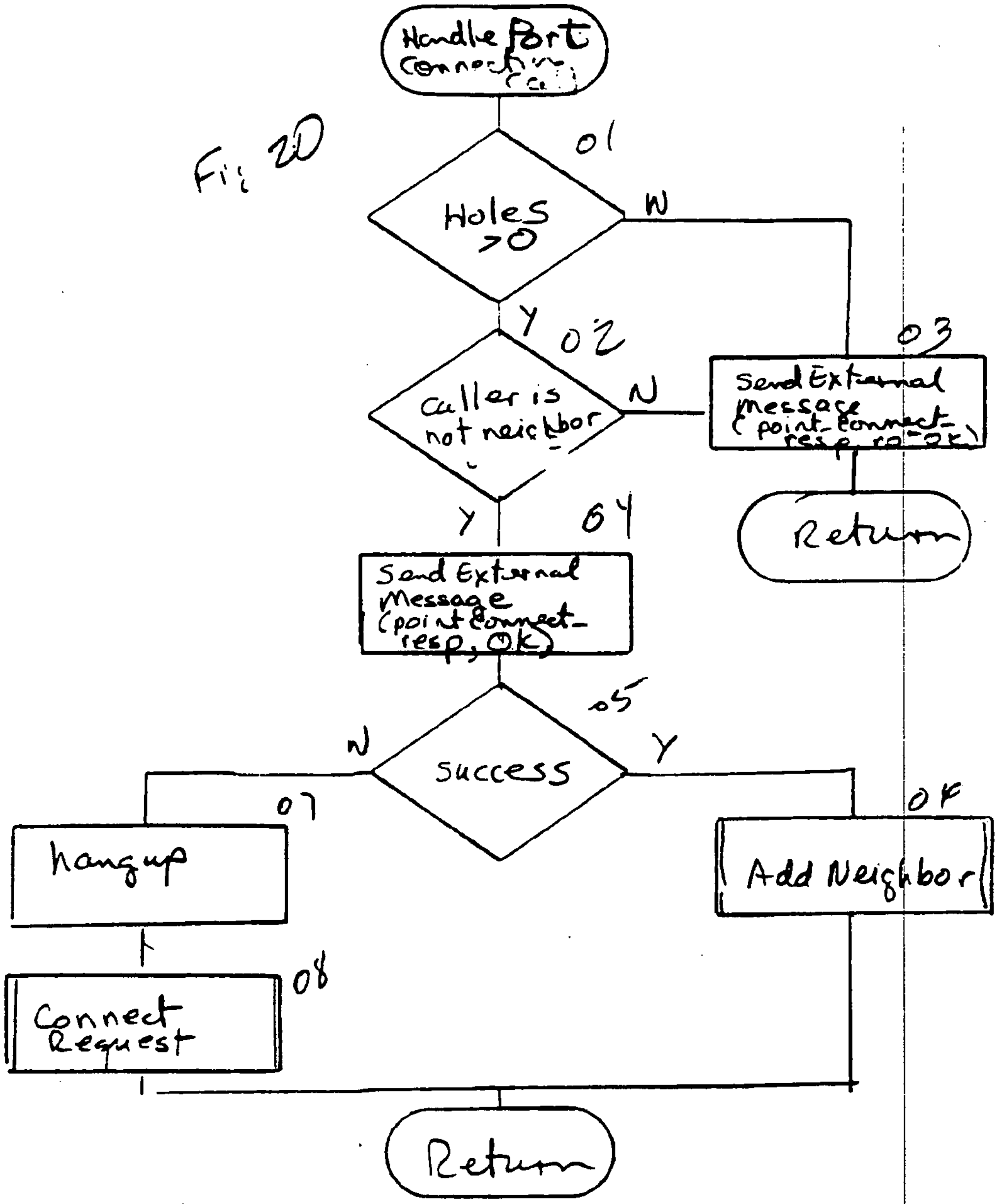
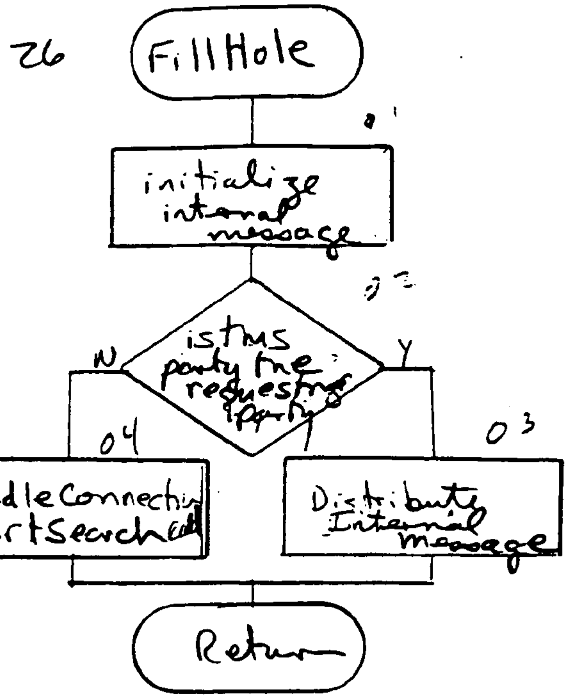
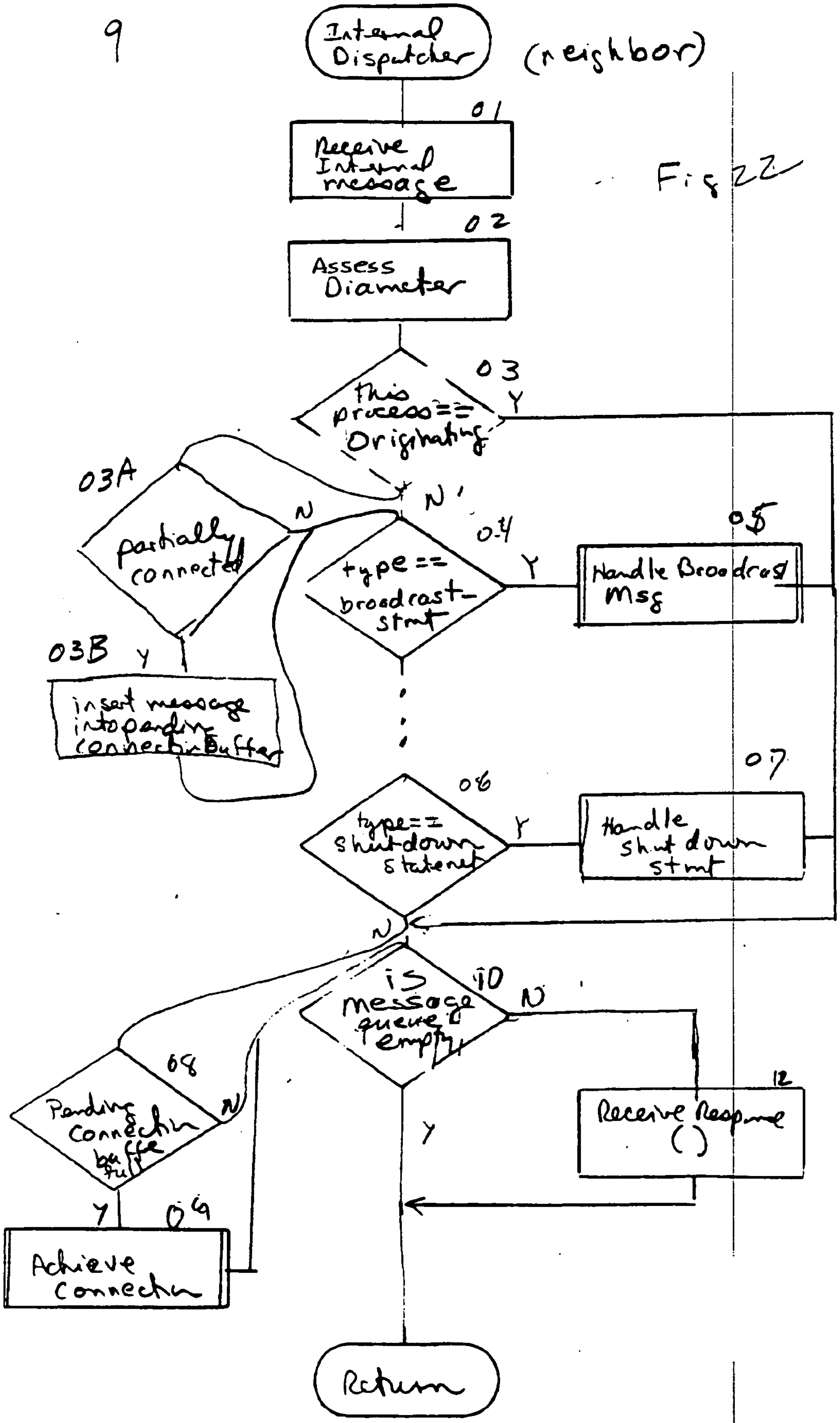

Handle Port Connection Call
01
Holes >0
N
Y
02
caller is not neighbor
N
03
Send External message (point_connect_resp, no-OK)
Return
Y
04
Send External Message (point_connect_resp, OK)
05
success
N
Y
07
hangup
06
Add Neighbor
08
Connect Request
Return

Fig 21


26
FillHole
01
initialize internal message
02
is this party the requesting party
N
Y
04
Handle Connection PortSearch
03
Distribute Internal message
Return

9

Fig 22



Internal Dispatcher
(neighbor)
01
Receive Internal message
02
Assess Diameter
03
this process == Originating
Y
N
03A
partially connected
N
03B
Y
insert message into pending connectionBuffer
04
type == broadcast_stmt
Y
05
Handle Broadcast Msg
06
type == shutdown statement
Y
07
Handle shutdown stmt
N
10
is message queue empty
N
12
Receive Response ( )
Y
08
Pending connection buffer full
N
Y
09
Achieve connection
Return

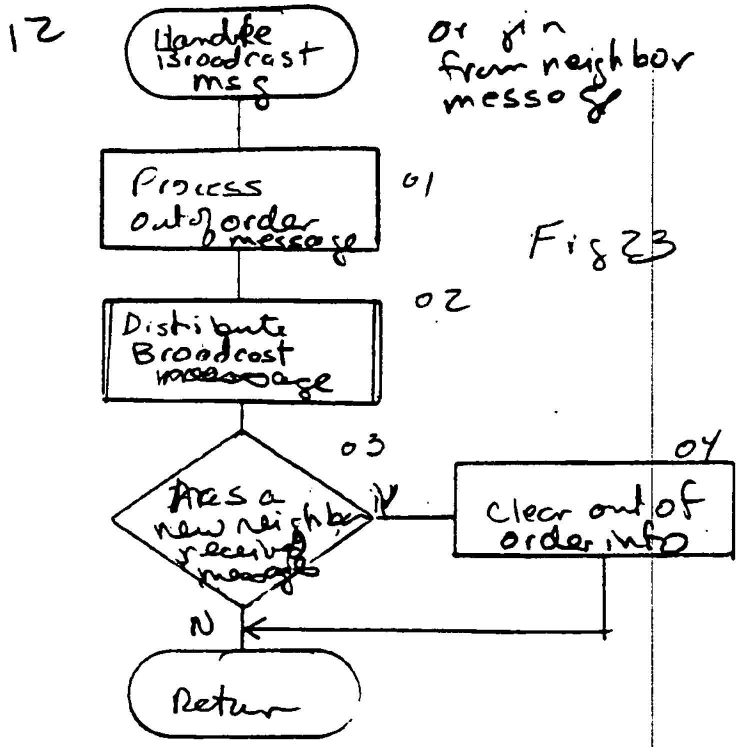

12
Handle Broadcast msg
or join from neighbor message
Process out of order message
01
Distribute Broadcast message
02
03
Was a new neighbor receive message
Y
04
Clear out of order info
N
Return

Fig 23

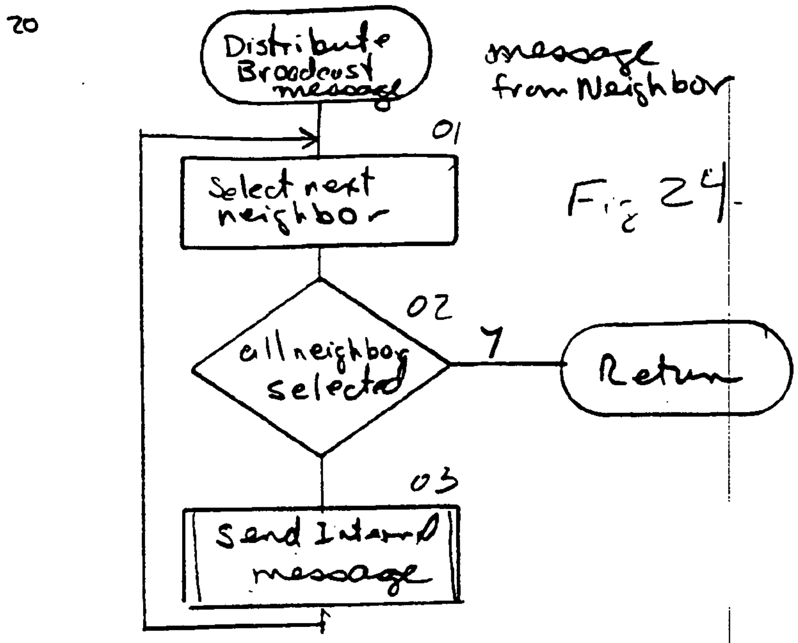
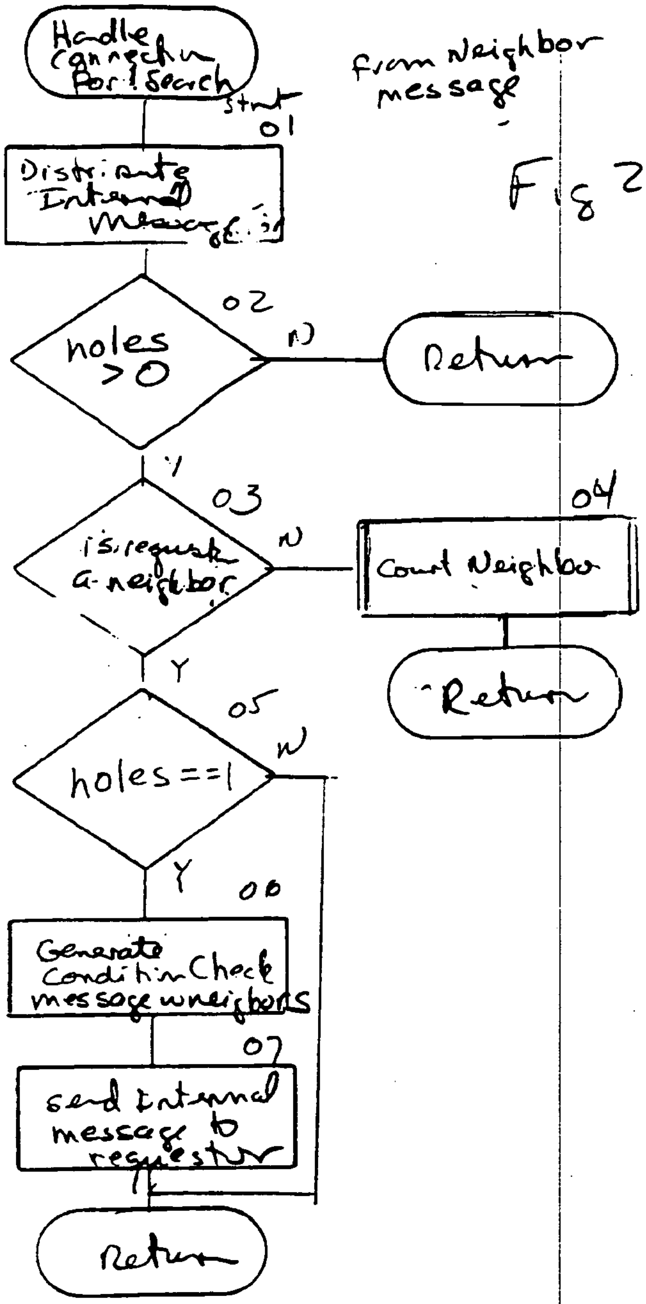

20
Distribute Broadcast message
message from Neighbor
01
Select next neighbor
02
all neighbor selected
Y
Return
03
Send Internal message

Fig 24.

13


Handle connection for !search
start
01
Distribute Internal Message
02
holes >0
N
Return
Y
03
is request a neighbor
N
04
Count Neighbor
Return
Y
05
holes==1
N
Y
06
Generate Condition Check message w neighbors
07
send Internal message to requester
Return

from Neighbor message

Fig 26

21

21

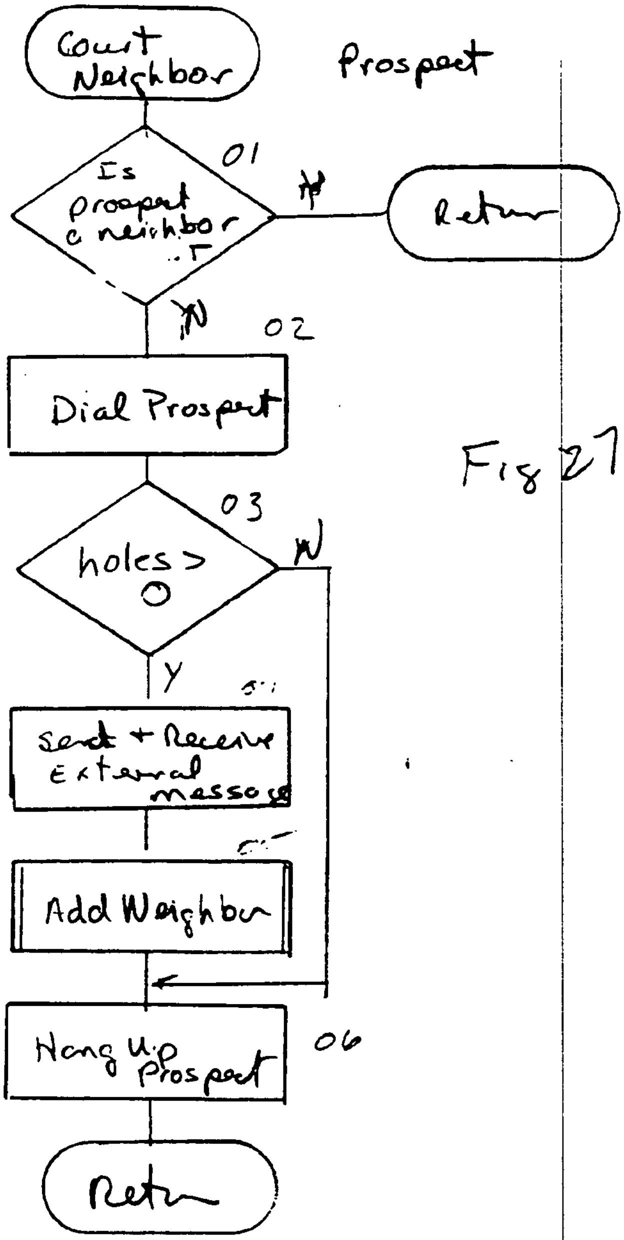
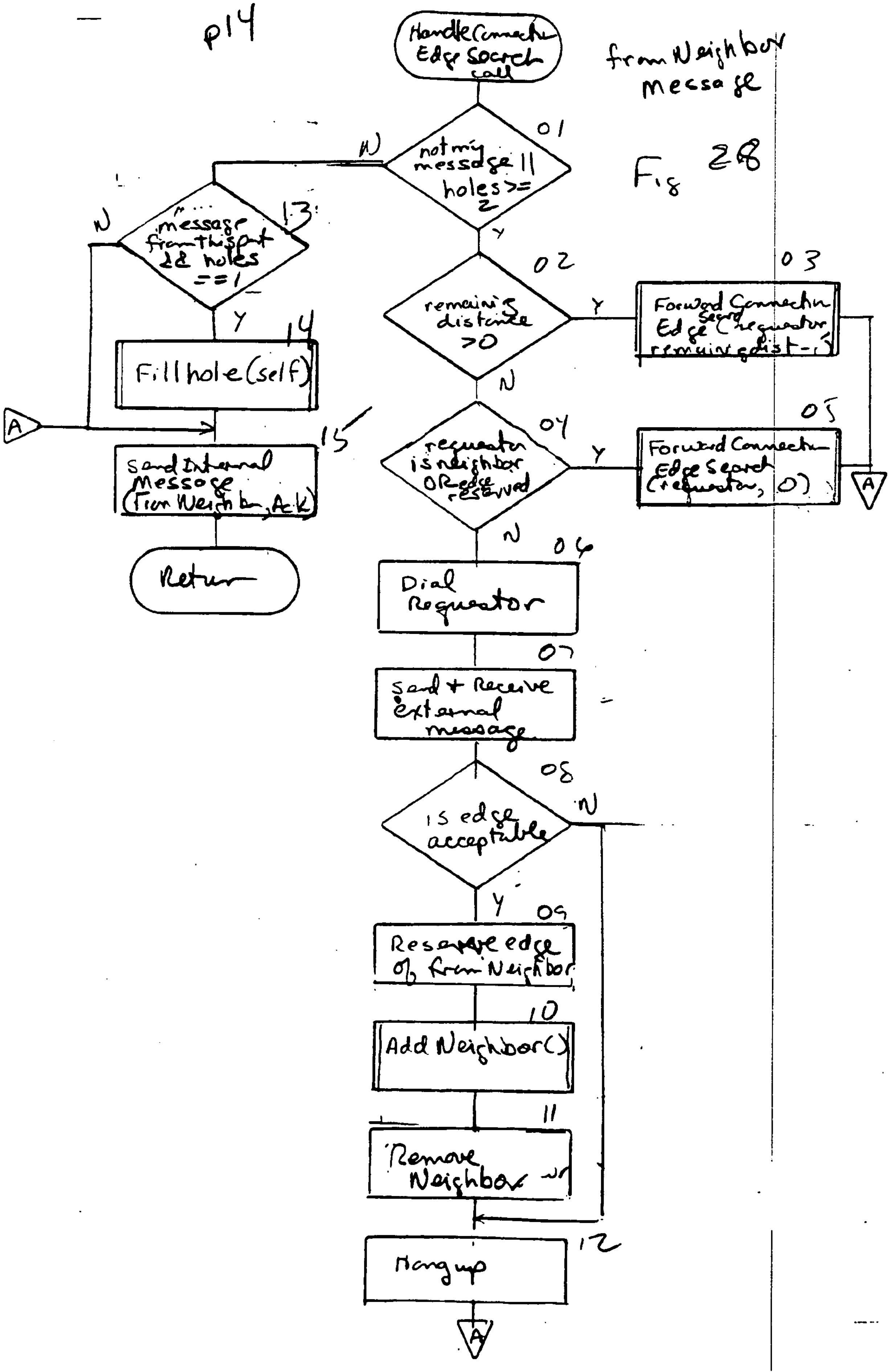

Count Neighbor
Prospect
01
Is prospect a neighbor ?
Y
Return
N
02
Dial Prospect
03
holes > 0
N
Y
04
Send + Receive external message
05
Add Neighbor
06
Hang up Prospect
Return

Fig 27

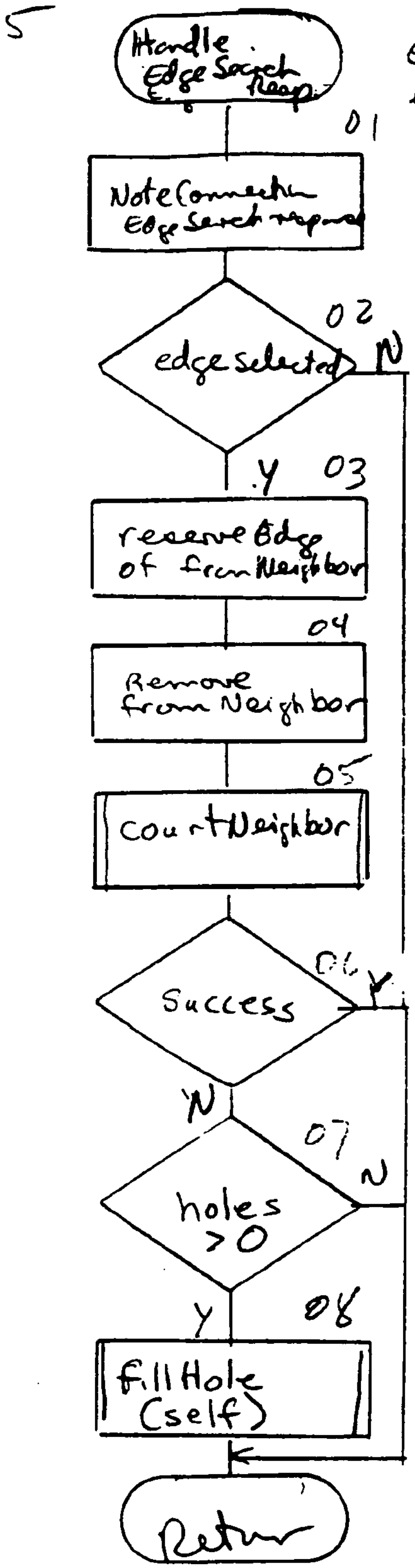
p14

from Neighbor message

Fig 28



Handle Connection Edge Search call
01
not my message || holes >= 2
N
Y
13
message from this spot && holes == 1
N
Y
14
Fill hole (self)
A
15
Send Internal Message (from Neighbor, Ack)
Return
02
remaining distance >0
Y
03
Forward Connection Edge Search (requestor, remaining dist-1)
N
04
requestor is neighbor OR edge reserved
Y
05
Forward Connection Edge Search (requestor, 0)
A
N
06
Dial Requestor
07
Send + Receive external message
08
is edge acceptable
N
Y
09
Reserve edge of from Neighbor
10
Add Neighbor()
11
Remove Neighbor
12
Hangup
A


5
Handle Edge Search Resp.
01
Note Connection Edge Search response
02
edge selected
N
Y
03
reserve Edge of fromNeighbor
04
Remove fromNeighbor
05
countNeighbor
06
Y
Success
N
07
N
holes >0
Y
08
fillHole (self)
Return

origin fromNeighbor message

Fig 29

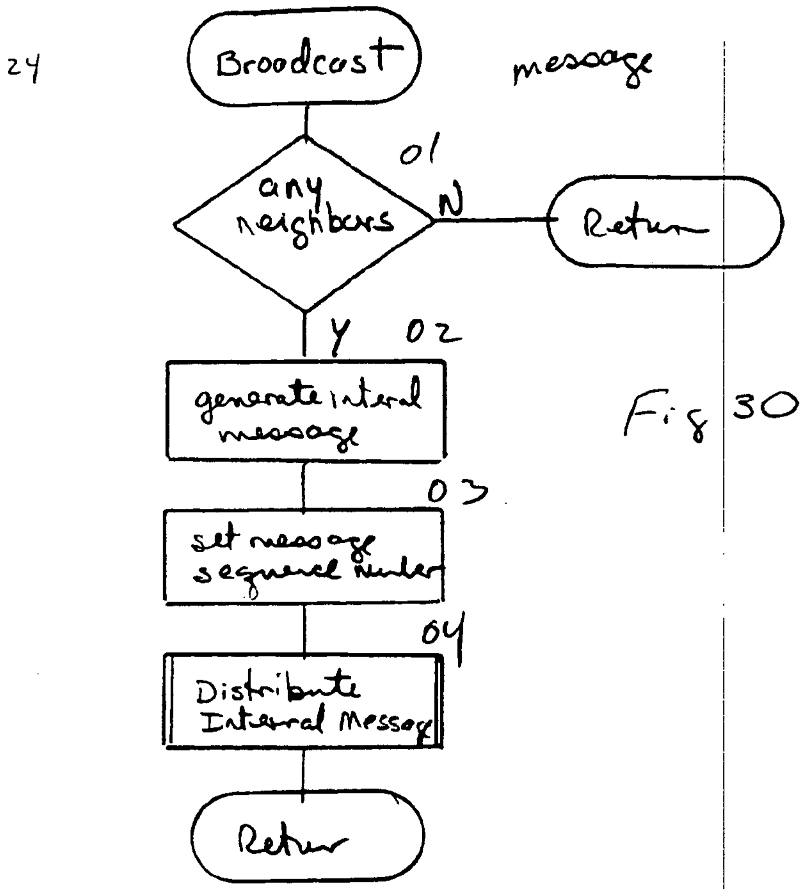

24
Broadcast
message
01
any neighbors
N
Return
Y
02
generate internal message
03
set message sequence number
04
Distribute Internal Message
Return

Fig 30

3

message

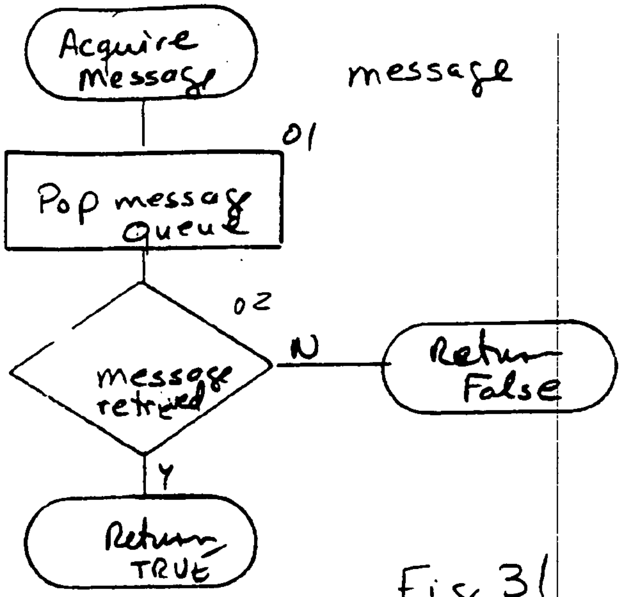

Acquire Message
01
Pop message queue
02
message retrieved
N
Return False
Y
Return TRUE

Fig 31

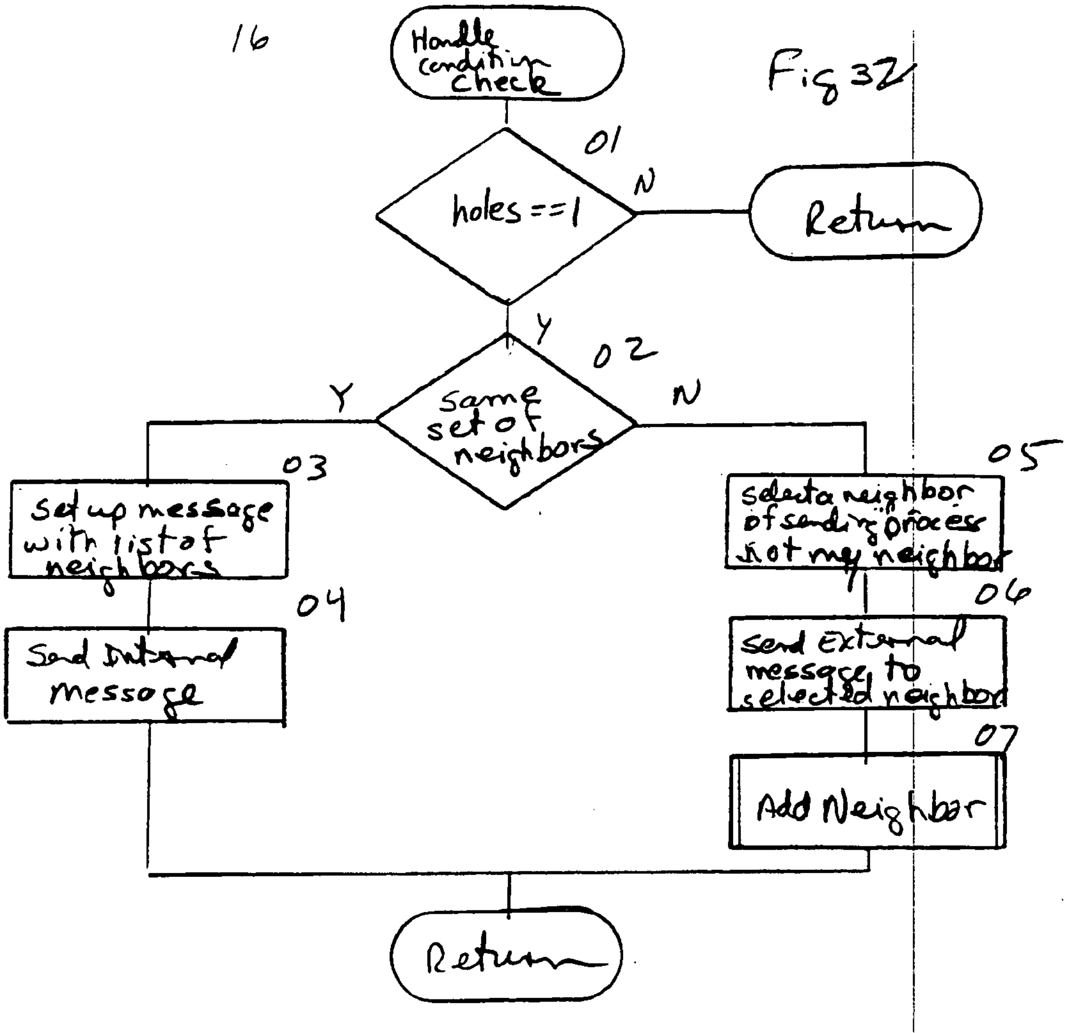

Handle condition check
01
holes ==1
N
Return
Y
02
same set of neighbors
Y
N
03
set up message with list of neighbors
04
Send Internal message
05
select a neighbor of sending process not my neighbor
06
send External message to selected neighbor
07
Add Neighbor
Return

Fig 32

23

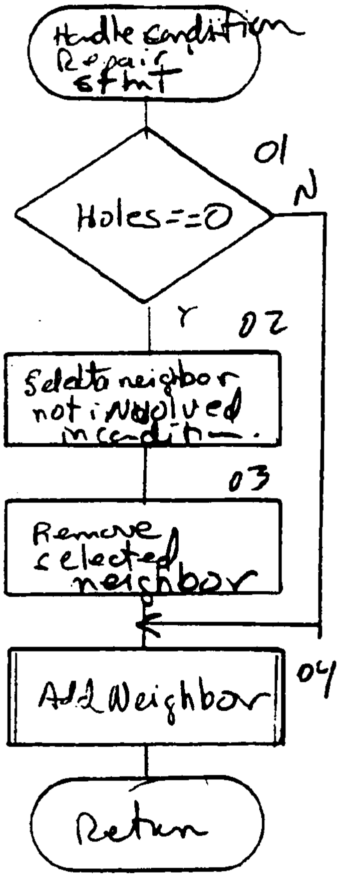
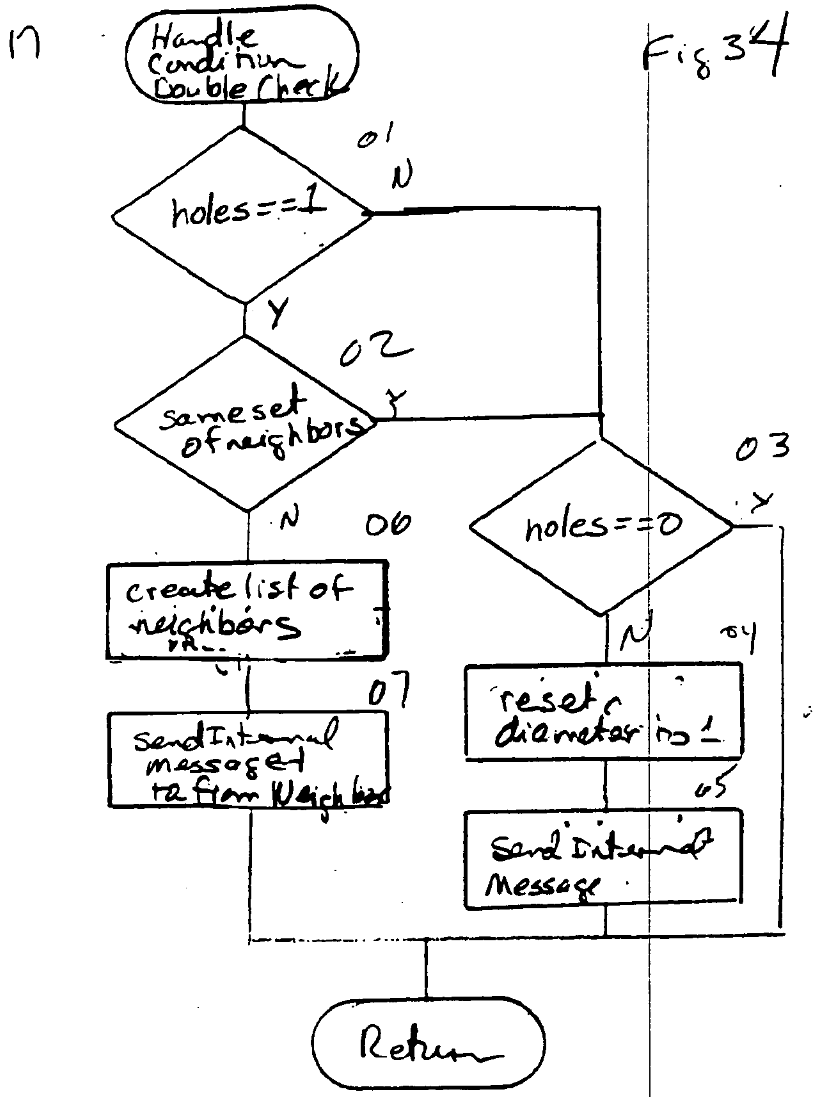

Handle condition Repair stmt
01
Holes==0
N
Y
02
Select a neighbor not involved in condition.
03
Remove selected neighbor
04
Add Neighbor
Return

Fig 33

Fig 34


17
Handle Condition Double Check
01
N
holes==1
Y
02
same set of neighbors
Y
N
06
03
Y
holes==0
create list of neighbors
N
04
07
reset diameter to 1
send Internal message to from Neighbors
05
Send Internal Message
Return

BROADCASTING ON A BROADCAST CHANNEL

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is related to U.S. Patent Application No. _____________, entitled "BROADCASTING NETWORK," filed on July 31, 2000 (Attorney Docket No. 030048001 US); U.S. Patent Application No._____________, entitled "JOINING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048002 US); U.S. Patent Application No._____________, "LEAVING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048003 US); U.S. Patent Application No._____________, entitled "BROADCASTING ON A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048004 US); U.S. Patent Application No._____________, entitled "CONTACTING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048005 US); U.S. Patent Application No._____________, entitled "DISTRIBUTED AUCTION SYSTEM," filed on July 31, 2000 (Attorney Docket No. 030048006 US); U.S. Patent Application No._____________, entitled "AN INFORMATION DELIVERY SERVICE," filed on July 31, 2000 (Attorney Docket No. 030048007 US); U.S. Patent Application No._____________, entitled "DISTRIBUTED CONFERENCING SYSTEM," filed on July 31, 2000 (Attorney Docket No. 030048008 US); and U.S. Patent Application No._____________, entitled "DISTRIBUTED GAME ENVIRONMENT," filed on July 31, 2000 (Attorney Docket No. 030048009 US), the disclosures of which are incorporated herein by reference.

TECHNICAL FIELD

The described technology relates generally to a computer network and more particularly, to a broadcast channel for a subset of a computers of an underlying network.

BACKGROUND

There are a wide variety of computer network communications techniques such as point-to-point network protocols, client/server middleware, multicasting network protocols, and peer-to-peer middleware. Each of these communications techniques have their advantages and disadvantages, but none is particularly well suited to the simultaneous sharing of information among computers that are widely distributed. For example, collaborative processing applications, such as a network meeting programs, have a need to distribute information in a timely manner to all participants who may be geographically distributed.

The point-to-point network protocols, such as UNIX pipes, TCP/IP, and UDP, allow processes on different computers to communicate via point-to-point connections. The interconnection of all participants using point-to-point connections, while theoretically possible, does not scale well as a number of participants grows. For example, each participating process would need to manage its direct connections to all other participating processes. Programmers, however, find it very difficult to manage single connections, and management of multiple connections is much more complex. In addition, participating processes may be limited to the number of direct connections that they can support. This limits the number of possible participants in the sharing of information.

The client/server middleware systems provide a server that coordinates the communications between the various clients who are sharing the information. The server functions as a central authority for controlling access to shared resources. Examples of client/server middleware systems include remote procedure calls ("RPC"), database servers, and the common object request broker architecture ("CORBA"). Client/server middleware systems are not particularly well suited to sharing of information among many participants. In particular, when a client stores information to be shared at the server, each other client would need to poll the server to determine that new information is being shared. Such polling places a very high overhead on the communications network. Alternatively, each client may register a callback with the server, which the server then invokes when new information is available to be shared. Such a callback technique presents a performance bottleneck because a single server needs to call back to each client whenever new information is to be shared. In addition, the reliability of the entire sharing of information depends upon the reliability of the single server. Thus, a failure at a single computer (*i.e.*, the server) would prevent communications between any of the clients.

The multicasting network protocols allow the sending of broadcast messages to multiple recipients of a network. The current implementations of such multicasting network protocols tend to place an unacceptable overhead on the underlying network. For example, UDP multicasting would swamp the Internet when trying to locate all possible participants. IP multicasting has other problems that include needing special-purpose infrastructure (*e.g.*, routers) to support the sharing of information efficiently.

The peer-to-peer middleware communications systems rely on a multicasting network protocol or a graph of point-to-point network protocols. Such peer-to-peer middleware is provided by the T.120 Internet standard, which is used in such products as Data Connection's D.C.-share and Microsoft's NetMeeting. These peer-to-peer middleware systems rely upon a user to assemble a point-to-point graph of the connections used for sharing the information. Thus, it is neither suitable nor desirable to use peer-to-peer middleware systems when more than a small number of participants is desired. In addition, the underlying architecture of the T.120 Internet standard is a tree structure, which relies on the root node of the tree for reliability of the entire network. That is, each message must pass through the root node in order to be received by all participants.

It would be desirable to have a reliable communications network that is suitable for the simultaneous sharing of information among a large number of the processes that are widely distributed.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a graph that is 4-regular and 4-connected which represents a broadcast channel.

Figure 2 illustrates a graph representing 20 computers connected to a broadcast channel.

Figures 3A and 3B illustrate the process of connecting a new computer Z to the broadcast channel.

Figure 4A illustrates the broadcast channel of Figure 1 with an added computer.

Figure 4B illustrates the broadcast channel of Figure 4A with an added computer.

Figure 4C also illustrates the broadcast channel of Figure 4A with an added computer.

Figure 5A illustrates the disconnecting of a computer from the broadcast channel in a planned manner.

Figure 5B illustrates the disconnecting of a computer from the broadcast channel in an unplanned manner.

Figure 5C illustrates the neighbors with empty ports condition.

Figure 5D illustrates two computers that are not neighbors who now have empty ports.

Figure 5E illustrates the neighbors with empty ports condition in the small regime.

Figure 5F illustrates the situation of Figure 5E when in the large regime.

Figure 6 is a block diagram illustrating components of a computer that is connected to a broadcast channel.

Figure 7 is a block diagram illustrating the sub-components of the broadcaster component in one embodiment.

Figure 8 is a flow diagram illustrating the processing of the connect routine in one embodiment.

Figure 9 is a flow diagram illustrating the processing of the seek portal computer routine in one embodiment.

Figure 10 is a flow diagram illustrating the processing of the contact process routine in one embodiment.

Figure 11 is a flow diagram illustrating the processing of the connect request routine in one embodiment.

Figure 12 is a flow diagram of the processing of the check for external call routine in one embodiment.

Figure 13 is a flow diagram of the processing of the achieve connection routine in one embodiment.

Figure 14 is a flow diagram illustrating the processing of the external dispatcher routine in one embodiment.

Figure 15 is a flow diagram illustrating the processing of the handle seeking connection call routine in one embodiment.

Figure 16 is a flow diagram illustrating processing of the handle connection request call routine in one embodiment.

Figure 17 is a flow diagram illustrating the processing of the add neighbor routine in one embodiment.

Figure 18 is a flow diagram illustrating the processing of the forward connection edge search routine in one embodiment.

Figure 19 is a flow diagram illustrating the processing of the handle edge proposal call routine.

Figure 20 is a flow diagram illustrating the processing of the handle port connection call routine in one embodiment.

Figure 21 is a flow diagram illustrating the processing of the fill hole routine in one embodiment.

Figure 22 is a flow diagram illustrating the processing of the internal dispatcher routine in one embodiment.

Figure 23 is a flow diagram illustrating the processing of the handle broadcast message routine in one embodiment.

Figure 24 is a flow diagram illustrating the processing of the distribute broadcast message routine in one embodiment.

Figure 26 is a flow diagram illustrating the processing of the handle connection port search statement routine in one embodiment.

Figure 27 is a flow diagram illustrating the processing of the court neighbor routine in one embodiment.

Figure 28 is a flow diagram illustrating the processing of the handle connection edge search call routine in one embodiment.

Figure 29 is a flow diagram illustrating the processing of the handle connection edge search response routine in one embodiment.

Figure 30 is a flow diagram illustrating the processing of the broadcast routine in one embodiment.

Figure 31 is a flow diagram illustrating the processing of the acquire message routine in one embodiment.

Figure 32 is a flow diagram illustrating processing of the handle condition check message in one embodiment.

Figure 33 is a flow diagram illustrating processing of the handle condition repair statement routine in one embodiment.

Figure 34 is a flow diagram illustrating the processing of the handle condition double check routine.

DETAILED DESCRIPTION

A broadcast technique in which a broadcast channel overlays a point-to-point communications network is provided. The broadcasting of a message over the broadcast channel is effectively a multicast to those computers of the network that are currently connected to the broadcast channel. In one embodiment, the broadcast technique provides a logical broadcast channel to which host computers through their executing processes can be connected. Each computer that is connected to the broadcast channel can broadcast messages onto and receive messages off of the broadcast channel. Each computer that is connected to the broadcast channel receives all messages that are broadcast while it is connected. The logical broadcast channel is implemented using an underlying network system (*e.g.*, the Internet) that allows each computer connected to the underlying network system to send messages to each other connected computer using each computer's address. Thus, the broadcast technique effectively provides a broadcast channel using an underlying network system that sends messages on a point-to-point basis.

The broadcast technique overlays the underlying network system with a graph of point-to-point connections (*i.e.*, edges) between host computers (*i.e.*, nodes) through which the broadcast channel is implemented. In one embodiment, each computer is connected to four other computers, referred to as neighbors. (Actually, a process executing on a computer is connected to four other processes executing on this or four other computers.) To broadcast a message, the originating computer sends the message to each of its neighbors using its point-to-point connections. Each computer that receives the message then sends the message to its three other neighbors using the point-to-point connections. In this way, the message is propagated to each computer using the underlying network to effect the broadcasting of the message to each computer over a logical broadcast channel. A graph in which each node is connected to four other nodes is referred to as a 4-regular graph. The use of a 4-regular graph means that a computer would become disconnected from the broadcast channel only if all four of the connections to its neighbors fail. The graph used by the broadcast technique also has the property that it would take a failure of four computers to divide the graph into disjoint sub-graphs, that is two separate broadcast channels. This property is referred to as being 4-connected. Thus, the graph is both 4-regular and 4-connected.

Figure 1 illustrates a graph that is 4-regular and 4-connected which represents the broadcast channel. Each of the nine nodes A-I represents a computer that is connected to the broadcast channel, and each of the edges represents an "edge" connection between two computers of the broadcast channel. The time it takes to broadcast a message to each computer on the broadcast channel depends on the speed of the connections between the computers and the number of connections between the originating computer and each other computer on the broadcast channel. The minimum number of connections that a message would need to traverse between each pair of computers is the "distance" between the computers (*i.e.*, the shortest path between the two nodes of the graph). For example, the distance between computers A and F is one because computer A is directly connected to computer F. The distance between computers A and B is two because there is no direct connection between computers A and B, but computer F is directly connected to computer B. Thus, a message originating at computer A would be sent directly to computer F, and then sent from computer F to computer B. The maximum of the distances between the computers is the "diameter" of broadcast channel. The diameter of the broadcast channel represented by Figure 1 is two. That is, a message sent by any computer would traverse no more than two connections to reach every other computer. Figure 2 illustrates a graph representing 20 computers connected to a broadcast channel. The diameter of this broadcast channel is 4. In particular, the shortest path between computers 1 and 3 contains four connections (1-12, 12-15, 15-18, and 18-3).

The broadcast technique includes (1) the connecting of computers to the broadcast channel (*i.e.*, composing the graph), (2) the broadcasting of messages over the broadcast channel (*i.e.*, broadcasting through the graph), and (3) the disconnecting of computers from the broadcast channel (*i.e.*, decomposing the graph) composing the graph.

Composing the Graph

To connect to the broadcast channel, the computer seeking the connection first locates a computer that is currently fully connected to the broadcast channel and then establishes a connection with four of the computers that are already connected to the broadcast channel. (This assumes that there are at least four computers already connected to the broadcast channel. When there are fewer than five computers connected, the broadcast channel cannot be a 4-regular graph. In such a case, the broadcast channel is considered to be in a "small regime." The broadcast technique for the small regime is described below in detail. When five or more computers are connected, the broadcast channel is considered to be in the "large regime." This description assumes that the broadcast channel is in the large regime, unless specified otherwise.) Thus, the process of connecting to the broadcast channel includes locating the broadcast channel, identifying the neighbors for the connecting computer, and then connecting to each identified neighbor. Each computer is aware of one or more "portal computers" through which that computer may locate the broadcast channel. A seeking computer locates the broadcast channel by contacting the portal computers until it finds one that is currently fully connected to the broadcast channel. The found portal computer then directs the identifying of four computers (*i.e.*, to be the seeking computer's neighbors) to which the seeking computer is to connect. Each of these four computers then cooperates with the seeking computer to effect the connecting of the seeking computer to the broadcast channel. A computer that has started the process of locating a portal computer, but does not yet have a neighbor, is in the "seeking connection state." A computer that is connected to at least one neighbor, but not yet four neighbors, is in the "partially connected state." A computer that is currently, or has been, previously connected to four neighbors is in the "fully connected state."

Since the broadcast channel is a 4-regular graph, each of the identified computers is already connected to four computers. Thus, some connections between computers need to be broken so that the seeking computer can connect to four computers. In one embodiment, the broadcast technique identifies two pairs of computers that are currently connected to each other. Each of these pairs of computers breaks the connection between them, and then each of the four computers (two from each pair) connects to the seeking computer. Figures 3A and 3B illustrate the process of a new computer Z connecting to the broadcast channel. Figure 3A illustrates the broadcast channel before computer Z is connected. The pairs of computers B and E and computers C and D are the two pairs that are identified as the neighbors for the new computer Z. The connections between each of these pairs is broken, and a connection between computer Z and each of computers B, C, D, and E is established as indicated by Figure 3B. The process of breaking the connection between two neighbors and reconnecting each of the former neighbors to another computer is referred to as "edge pinning" as the edge between two nodes may be considered to be stretched and pinned to a new node.

Each computer connected to the broadcast channel allocates five communications ports for communicating with other computers. Four of the ports are referred to as "internal" ports because they are the ports through which the messages of the broadcast channels are sent. The connections between internal ports of neighbors are referred to as "internal" connections. Thus, the internal connections of the broadcast channel form the 4-regular and 4-connected graph. The fifth port is referred to as an "external" port because it is used for sending non-broadcast messages between two computers. Neighbors can send non-broadcast messages either through their internal ports of their connection or through their external ports. A seeking computer uses external ports when locating a portal computer.

In one embodiment, the broadcast technique establishes the computer connections using the TCP/IP communications protocol, which is a point-to-point protocol, as the underlying network. The TCP/IP protocol provides for reliable and ordered delivery of messages between computers. The TCP/IP protocol provides each computer with a "port space" that is shared among all the processes that may execute on that computer. The ports are identified by numbers from 0 to 65,535. The first 2056 ports are reserved for specific applications (*e.g.*, port 80 for HTTP messages). The remainder of the ports are user ports that are available to any process. In one embodiment, a set of port numbers can be reserved for use by the computer connected to the broadcast channel. In an alternative embodiment, the port numbers used are dynamically identified by each computer. Each computer dynamically identifies an available port to be used as its call-in port. This call-in port is used to establish connections with the external port and the internal ports. Each computer that is connected to the broadcast channel can receive non-broadcast messages through its external port. A seeking computer tries "dialing" the port numbers of the portal computers until a portal computer "answers," a call on its call-in port. A portal computer answers when it is connected to or attempting to connect to the broadcast channel and its call-in port is dialed. (In this description, a telephone metaphor is used to describe the connections.) When a computer receives a call on its call-in port, it transfers the call to another port. Thus, the seeking computer actually communicates through that transfer-to port, which is the external port. The call is transferred so that other computers can place calls to that computer via the call-in port. The seeking computer then communicates via that external port to request the portal computer to assist in connecting the seeking computer to the broadcast channel. The seeking computer could identify the call-in port number of a portal computer by successively dialing each port in port number order. As discussed below in detail, the broadcast technique uses a hashing algorithm to select the port number order, which may result in improved performance.

A seeking computer could connect to the broadcast channel by connecting to computers either directly connected to the found portal computer or directly connected to one of its neighbors. A possible problem with such a scheme for identifying the neighbors for the seeking computer is that the diameter of the broadcast channel may increase when each seeking computer uses the same found portal computer and establishes a connection to the broadcast channel directly through that found portal computer. Conceptually, the graph becomes elongated in the direction of where the new nodes are added. Figures 4A-4C illustrate that possible problem. Figure 4A illustrates the broadcast channel of Figure 1 with an added computer. Computer J was connected to the broadcast channel by edge pinning edges C-D and E-H to computer J. The diameter of this broadcast channel is still two. Figure 4B illustrates the broadcast channel of Figure 4A with an added computer. Computer K was connected to the broadcast channel by edge pinning edges E-J and B-C to computer K. The diameter of this broadcast channel is three, because the shortest path from computer G to computer K is through edges G-A, A-E, and E-K. Figure 4C also illustrates the broadcast channel of Figure 4A with an added computer. Computer K was connected to the broadcast channel by edge pinning edges D-G and E-J to computer K. The diameter of this broadcast channel is, however, still two. Thus, the selection of neighbors impacts the diameter of the broadcast channel. To help minimize the diameter, the broadcast technique uses a random selection technique to identify the four neighbors of a computer in the seeking connection state. The random selection technique tends to distribute the connections to new seeking computers throughout the computers of the broadcast channel which may result in smaller overall diameters.

Broadcasting Through the Graph

As described above, each computer that is connected to the broadcast channel can broadcast messages onto the broadcast channel and does receive all messages that are broadcast on the broadcast channel. The computer that originates a message to be broadcast sends that message to each of its four neighbors using the internal connections. When a computer receives a broadcast message from a neighbor, it sends the message to its three other neighbors. Each computer on the broadcast channel, except the originating computer, will thus receive a copy of each broadcast message from each of its four neighbors. Each computer, however, only sends the first copy of the message that it receives to its neighbors and disregards subsequently received copies. Thus, the total number of copies of a message that is sent between the computers is 3N+1, where N is the number of computers connected to the broadcast channel. Each computer sends three copies of the message, except for the originating computer, which sends four copies of the message.

The redundancy of the message sending helps to ensure the overall reliability of the broadcast channel. Since each computer has four connections to the broadcast channel, if one computer fails during the broadcast of a message, its neighbors have three other connections through which they will receive copies of the broadcast message. Also, if the internal connection between two computers is slow, each computer has three other connections through which it may receive a copy of each message sooner.

Each computer that originates a message numbers its own messages sequentially. Because of the dynamic nature of the broadcast channel and because there are many possible connection paths between computers, the messages may be received out of order. For example, the distance between an originating computer and a certain receiving computer may be four. After sending the first message, the originating computer and receiving computer may become neighbors and thus the distance between them changes to one. The first message may have to travel a distance of four to reach the receiving computer. The second message only has to travel a distance of one. Thus, it is possible for the second message to reach the receiving computer before the first message.

When the broadcast channel is in a steady state (*i.e.*, no computers connecting or disconnecting from the broadcast channel), out-of-order messages are not a problem because each computer will eventually receive both messages and can queue messages until all earlier ordered messages are received. If, however, the broadcast channel is not in a steady state, then problems can occur. In particular, a computer may connect to the broadcast channel after the second message has already been received and forwarded on by its new neighbors. When a new neighbor eventually receives the first message, it sends the message to the newly connected computer. Thus, the newly connected computer will receive the first message, but will not receive the second message. If the newly connected computer needs to process the messages in order, it would wait indefinitely for the second message.

One solution to this problem is to have each computer queue all the messages that it receives until it can send them in their proper order to its neighbors. This solution, however, may tend to slow down the propagation of messages through the computers of the broadcast channel. Another solution that may have less impact on the propagation speed is to queue messages only at computers who are neighbors of the newly connected computers. Each already connected neighbor would forward messages as it receives them to its other neighbors who are not newly connected, but not to the newly connected neighbor. The already connected neighbor would only forward messages from each originating computer to the newly connected computer when it can ensure that no gaps in the messages from that originating computer will occur. In one embodiment, the already connected neighbor may track the highest sequence number of the messages already received and forwarded on from each originating computer. The already connected computer will send only higher numbered messages from the originating computers to the newly connected computer. Once all lower numbered messages have been received from all originating computers, then the already connected computer can treat the newly connected computer as its other neighbors and simply forward each message as it is received. In another embodiment, each computer may queue messages and only forwards to the newly connected computer those messages as the gaps are filled in. For example, a computer might receive messages 4 and 5 and then receive message 3. In such a case, the already connected computer would forward queue messages 4 and 5. When message 3 is finally received, the already connected computer will send messages 3, 4, and 5 to the newly connected computer. If messages 4 and 5 were sent to the newly connected computer before message 3, then the newly connected computer would process messages 4 and 5 and disregard message 3. Because the already connected computer queues messages 4 and 5, the newly connected computer will be able to process message 3. It is possible that a newly connected computer will receive a set of messages from an originating computer through one neighbor and then receive another set of message from the same originating computer through another neighbor. If the second set of messages contains a message that is ordered earlier than the messages of the first set received, then the newly connected computer may ignore that earlier ordered message if the computer already processed those later ordered messages.

Decomposing the Graph

A connected computer disconnects from the broadcast channel either in a planned or unplanned manner. When a computer disconnects in a planned manner, it sends a disconnect message to each of its four neighbors. The disconnect message includes a list that identifies the four neighbors of the disconnecting computer. When a neighbor receives the disconnect message, it tries to connect to one of the computers on the list. In one embodiment, the first computer in the list will try to connect to the second computer in the list, and the third computer in the list will try to connect to the fourth computer in the list. If a computer cannot connect (*e.g.*, the first and second computers are already connected), then the computers may try connecting in various other combinations. If connections cannot be established, each computer broadcasts a message that it needs to establish a connection with another computer. When a computer with an available internal port receives the message, it can then establish a connection with the computer that broadcast the message. Figures 5A-5D illustrate the disconnecting of a computer from the broadcast channel. Figure 5A illustrates the disconnecting of a computer from the broadcast channel in a planned manner. When computer H decides to disconnect, it sends its list of neighbors to each of its neighbors (computers A, E, F and I) and then disconnects from each of its neighbors. When computers A and I receive the message they establish a connection between them as indicated by the dashed line, and similarly for computers E and F.

When a computer disconnects in an unplanned manner, such as resulting from a power failure, the neighbors connected to the disconnected computer recognize the disconnection when each attempts to send its next message to the now disconnected computer. Each former neighbor of the disconnected computer recognizes that it is short one connection (*i.e.*, it has a hole or empty port). When a connected computer detects that one of its neighbors is now disconnected, it broadcasts a port connection request on the broadcast channel, which indicates that it has one internal port that needs a connection. The port connection request identifies the call-in port of the requesting computer. When a connected computer that is also short a connection receives the connection request, it communicates with the requesting computer through its external port to establish a connection between the two computers. Figure 5B illustrates the disconnecting of a computer from the broadcast channel in an unplanned manner. In this illustration, computer H has disconnected in an unplanned manner. When each of its neighbors, computers A, E, F, and I, recognizes the disconnection, each neighbor broadcasts a port connection request indicating that it needs to fill an empty port. As shown by the dashed lines, computers F and I and computers A and E respond to each other's requests and establish a connection.

It is possible that a planned or unplanned disconnection may result in two neighbors each having an empty internal port. In such a case, since they are neighbors, they are already connected and cannot fill their empty ports by connecting to each other. Such a condition is referred to as the "neighbors with empty ports" condition. Each neighbor broadcasts a port connection request when it detects that it has an empty port as described above. When a neighbor receives the port connection request from the other neighbor, it will recognize the condition that its neighbor also has an empty port. Such a condition may also occur when the broadcast channel is in the small regime. The condition can only be corrected when in the large regime. When in the small regime, each computer will have less than four neighbors. To detect this condition in the large regime, which would be a problem if not repaired, the first neighbor to receive the port connection request recognizes the condition and sends a condition check message to the other neighbor. The condition check message includes a list of the neighbors of the sending computer. When the receiving computer receives the list, it compares the list to its own list of neighbors. If the lists are different, then this condition has occurred in the large regime and repair is needed. To repair this condition, the receiving computer will send a condition repair request to one of the neighbors of the sending computer which is not already a neighbor of the receiving computer. When the computer receives the condition repair request, it disconnects from one of its neighbors (other than the neighbor that is involved with the condition) and connects to the computer that sent the condition repair request. Thus, one of the original neighbors involved in the condition will have had a port filled. However, two computers are still in need of a connection, the other original neighbor and the computer that is now disconnected from the computer that received the condition repair request. Those two computers send out port connection requests. If those two computers are not neighbors, then they will connect to each other when they receive the requests. If, however, the two computers are neighbors, then they repeat the condition repair process until two non-neighbors are in need of connections.

It is possible that the two original neighbors with the condition may have the same set of neighbors. When the neighbor that receives the condition check message determines that the sets of neighbors are the same, it sends a condition double check message to one of its neighbors other than the neighbor who also has the condition. When the computer receives the condition double check message, it determines whether it has the same set of neighbors as the sending computer. If so, the broadcast channel is in the small regime and the condition is not a problem. If the set of neighbors are different, then the computer that received the condition double check message sends a condition check message to the original neighbors with the condition. The computer that receives that condition check message directs one of it neighbors to connect to one of the original neighbors with the condition by sending a condition repair message. Thus, one of the original neighbors with the condition will have its port filled.

Figure 5C illustrates the neighbors with empty ports condition. In this illustration, computer H disconnected in an unplanned manner, but computers F and I responded to the port connection request of the other and are now connected together. The other former neighbors of computer H, computers A and E, are already neighbors, which gives rise to the neighbors with empty ports condition. In this example, computer E received the port connection request from computer A, recognized the possible condition, and sent (since they are neighbors via the internal connection) a condition check message with a list of its neighbors to computer A. When computer A received the list, it recognized that computer E has a different set of neighbor (*i.e.*, the broadcast channel is in the large regime). Computer A selected computer D, which is a neighbor of computer E and sent it a condition repair request. When computer D received the condition repair request, it disconnected from one of its neighbors (other than computer E), which is computer G in this example. Computer D then connected to computer A. Figure 5D illustrates two computers that are not neighbors who now have empty ports. Computers E and G now have empty ports and are not currently neighbors. Therefore, computers E and G can connect to each other.

Figures 5E and 5F further illustrate the neighbors with empty ports condition. Figure 5E illustrates the neighbors with empty ports condition in the small regime. In this example, if computer E disconnected in an unplanned manner, then each computer broadcasts a port connection request when it detects the disconnect. When computer A receives the port connection request form computer B, it detects the neighbors with empty ports condition and sends a condition check message to computer B. Computer B recognizes that it has the same set of neighbors (computer C and D) as computer A and then sends a condition double check message to computer C. Computer C recognizes that the broadcast channel is in the small regime because is also has the same set of neighbors as computers A and B, computer C may then broadcast a message indicating that the broadcast channel is in the small regime.

Figure 5F illustrates the situation of Figure 5E when in the large regime. As discussed above, computer C receives the condition double check message from computer B. In this case, computer C recognizes that the broadcast channel is in the large regime because it has a set of neighbors that is different from computer B. The edges extending up from computer C and D indicate connections to other computers. Computer C then sends a condition check message to computer B. When computer B receives the condition check message, it sends a condition repair message to one of the neighbors of computer C. The computer that receives the condition repair message disconnects from one of its neighbors, other than computer C, and tries to connect to computer B and the neighbor from which it disconnected tries to connect to computer A.

Port Selection

As described above, the TCP/IP protocol designates ports above number 2056 as user ports. The broadcast technique uses five user port numbers on each computer: one external port and four internal ports. Generally, user ports cannot be statically allocated to an application program because other applications programs executing on the same computer may use conflicting port numbers. As a result, in one embodiment, the computers connected to the broadcast channel dynamically allocate their port numbers. Each computer could simply try to locate the lowest number unused port on that computer and use that port as the call-in port. A seeking computer, however, does not know in advance the call-in port number of the portal computers when the port numbers are dynamically allocated. Thus, a seeking computer needs to dial ports of a portal computer starting with the lowest port number when locating the call-in port of a portal computer. If the portal computer is connected to (or attempting to connect to) the broadcast channel, then the seeking computer would eventually find the call-in port. If the portal computer is not connected, then the seeking computer would eventually dial every user port. In addition, if each application program on a computer tried to allocate low-ordered port numbers, then a portal computer may end up with a high-numbered port for its call-in port because many of the low-ordered port numbers would be used by other application programs. Since the dialing of a port is a relatively slow process, it would take the seeking computer a long time to locate the call-in port of a portal computer. To minimize this time, the broadcast technique uses a port ordering algorithm to identify the port number order that a portal computer should use when finding an available port for its call-in port. In one embodiment, the broadcast technique uses a hashing algorithm to identify the port order. The algorithm preferably distributes the ordering of the port numbers randomly through out the user port number space and only selects each port number once. In addition, every time the algorithm is executed on any computer for a given channel type and channel instance, it generates the same port ordering. As described below, it is possible for a computer to be connected to multiple broadcast channels that are uniquely identified by channel type and channel instance. The algorithm may be "seeded" with channel type and channel instance in order to generate a unique ordering of port numbers for each broadcast channel. Thus, a seeking computer will dial the ports of a portal computer in the same order as the portal computer used when allocating its call-in port.

If many computers are at the same time seeking connection to a broadcast channel through a single portal computer, then the ports of the portal computer may be busy when called by seeking computers. The seeking computers would typically need to keep on redialing a busy port. The process of locating a call-in port may be significantly slowed by such redialing. In one embodiment, each seeking computer may each reorder the first few port numbers generated by the hashing algorithm. For example, each seeking computer could randomly reorder the first eight port numbers generated by the hashing algorithm. The random ordering could also be weighted where the first port number generated by the hashing algorithm would have a 50% chance of being first in the reordering, the second port number would have a 25% chance of being first in the reordering, and so on. Because the seeking computers would use different orderings, the likelihood of finding a busy port is reduced. For example, if the first eight port numbers are randomly selected, then it is possible that eight seeking computers could be simultaneously dialing ports in different sequences which would reduce the chances of dialing a busy port.

Locating a Portal Computer

Each computer that can connect to the broadcast channel has a list of one or more portal computers through which it can connect to the broadcast channel. In one embodiment, each computer has the same set of portal computers. A seeking computer locates a portal computer that is connected to the broadcast channel by successively dialing the ports of each portal computer in the order specified by an algorithm. A seeking computer could select the first portal computer and then dial all its ports until a call-in port of a computer that is fully connected to the broadcast channel is found. If no call-in port is found, then the seeking computer would select the next portal computer and repeat the process until a portal computer with such a call-in port is found. A problem with such a seeking technique is that all user ports of each portal computer are dialed until a portal computer fully connected to the broadcast channel is found. In an alternate embodiment, the seeking computer selects a port number according to the algorithm and then dials each portal computer at that port number. If no acceptable call-in port to the broadcast channel is found, then the seeking computer selects the next port number and repeats the process. Since the call-in ports are likely allocated at lower-ordered port numbers, the seeking computer first dials the port numbers that are most likely to be call-in ports of the broadcast channel. The seeking computers may have a maximum search depth, that is the number of ports that it will dial when seeking a portal computer that is fully connected. If the seeking computer exhausts its search depth, then either the broadcast channel has not yet been established or, if the seeking computer is also a portal computer, it can then establish the broadcast channel with itself as the first fully connected computer.

When a seeking computer locates a portal computer that is itself not fully connected, the two computers do not connect when they first locate each other because the broadcast channel may already be established and accessible through a higher-ordered port number on another portal computer. If the two seeking computers were to connect to each other, then two disjoint broadcast channels would be formed. Each seeking computer can share its experience in trying to locate a portal computer with the other seeking computer. In particular, if one seeking computer has searched all the portal computers to a depth of eight, then the one seeking computer can share that it has searched to a depth of eight with another seeking computer. If that other seeking computer has searched to a depth of, for example, only four, it can skip searching through depths five through eight and that other seeking computer can advance its searching to a depth of nine.

In one embodiment, each computer may have a different set of portal computers and a different maximum search depth. In such a situation, it may be possible that two disjoint broadcast channels are formed because a seeking computer cannot locate a fully connected port computer at a higher depth. Similarly, if the set of portal computers are disjoint, then two separate broadcast channels would be formed.

Identifying Neighbors for a Seeking Computer

As described above, the neighbors of a newly connecting computer are preferably selected randomly from the set of currently connected computers. One advantage of the broadcast channel, however, is that no computer has global knowledge of the broadcast channel. Rather, each computer has local knowledge of itself and its neighbors. This limited local knowledge has the advantage that all the connected computers are peers (as far as the broadcasting is concerned) and the failure of any one computer (actually any three computers when in the 4-regular and 4-connect form) will not cause the broadcast channel to fail. This local knowledge makes it difficult for a portal computer to randomly select four neighbors for a seeking computer.

To select the four computers, a portal computer sends an edge connection request message through one of its internal connections that is randomly selected. The receiving computer again sends the edge connection request message through one of its internal connections that is randomly selected. This sending of the message corresponds to a random walk through the graph that represents the broadcast channel. Eventually, a receiving computer will decide that the message has traveled far enough to represent a randomly selected computer. That receiving computer will offer the internal connection upon which it received the edge connection request message to the seeking computer for edge pinning. Of course, if either of the computers at the end of the offered internal connection are already neighbors of the seeking computer, then the seeking computer cannot connect through that internal connection. The computer that decided that the message has traveled far enough will detect this condition of already being a neighbor and send the message to a randomly selected neighbor.

In one embodiment, the distance that the edge connection request message travels is established by the portal computer to be approximately twice the estimated diameter of the broadcast channel. The message includes an indication of the distance that it is to travel. Each receiving computer decrements that distance to travel before sending the message on. The computer that receives a message with a distance to travel that is zero is considered to be the randomly selected computer. If that randomly selected computer cannot connect to the seeking computer (*e.g.*, because it is already connected to it), then that randomly selected computer forwards the edge connection request to one of its neighbors with a new distance to travel. In one embodiment, the forwarding computer toggles the new distance to travel between zero and one to help prevent two computers from sending the message back and forth between each other.

Because of the local nature of the information maintained by each computer connected to the broadcast channel, the computers need not generally be aware of the diameter of the broadcast channel. In one embodiment, each message sent through the broadcast channel has a distance traveled field. Each computer that forwards a message increments the distance traveled field. Each computer also maintains an estimated diameter of the broadcast channel. When a computer receives a message that has traveled a distance that indicates that the estimated diameter is too small, it updates its estimated diameter and broadcasts an estimated diameter message. When a computer receives an estimated diameter message that indicates a diameter that is larger than its own estimated diameter, it updates its own estimated diameter. This estimated diameter is used to establish the distance that an edge connection request message should travel.

External Data Representation

The computers connected to the broadcast channel may internally store their data in different formats. For example, one computer may use 32-bit integers, and another computer may use 64-bit integers. As another example, one computer may use ASCII to represent text and another computer may use Unicode. To allow communications between heterogeneous computers, the messages sent over the broadcast channel may use the XDR ("eXternal Data Representation") format.

The underlying peer-to-peer communications protocol may send multiple messages in a single message stream. The traditional technique for retrieving messages from a stream has been to repeatedly invoke an operating system routine to retrieve the next message in the stream. The retrieval of each message may require two calls to the operating system: one to retrieve the size of the next message and the other to retrieve the number of bytes indicated by the retrieved size. Such calls to the operating system can, however, be very slow in comparison to the invocations of local routines. To overcome the inefficiencies of such repeated calls, the broadcast technique in one embodiment, uses XDR to identify the message boundaries in a stream of messages. The broadcast technique may request the operating system to provide the next, for example, 1,024 bytes from the stream. The broadcast technique can then repeatedly invoke the XDR routines to retrieve the messages and use the success or failure of each invocation to determine whether another block of 1,024 bytes needs to be retrieved from the operating system. The invocation of XDR routines do not involve system calls and are thus more efficient than repeated system calls.

M-Regular

In the embodiment described above, each fully connected computer has four internal connections. The broadcast technique can be used with other numbers of internal connections. For example, each computer could have 6, 8, or any even number of internal connections. As the number of internal connections increase, the diameter of the broadcast channel tends to decrease, and thus propagation time for a message tends to decrease. The time that it takes to connect a seeking computer to the broadcast channel may, however, increase as the number of internal connections increases. When the number of internal connectors is even, then the broadcast channel can be maintained as m-regular and m-connected (in the steady state). If the number of internal connections is odd, then when the broadcast channel has an odd number of computers connected, one of the computers will have less than that odd number of internal connections. In such a situation, the broadcast network is neither m-regular nor m-connected. When the next computer connects to the broadcast channel, it can again become m-regular and m-connected. Thus, with an odd number of internal connections, the broadcast channel toggles between being and not being m-regular and m-connected.

Components

Figure 6 is a block diagram illustrating components of a computer that is connected to a broadcast channel. The above description generally assumed that there was only one broadcast channel and that each computer had only one connection to that broadcast channel. More generally, a network of computers may have multiple broadcast channels, each computer may be connected to more than one broadcast channel, and each computer can have multiple connections to the same broadcast channel. The broadcast channel is well suited for computer processes (*e.g.*, application programs) that execute collaboratively, such as network meeting programs. Each computer process can connect to one or more broadcast channels. The broadcast channels can be identified by channel type (*e.g.*, application program name) and channel instance that represents separate broadcast channels for that channel type. When a process attempts to connect to a broadcast channel, it seeks a process currently connected to that broadcast channel that is executing on a portal computer. The seeking process identifies the broadcast channel by channel type and channel instance.

Computer 600 includes multiple application programs 601 executing as separate processes. Each application program interfaces with a broadcaster component 602 for each broadcast channel to which it is connected. The broadcaster component may be implement as an object that is instantiated within the process space of the application program. Alternatively, the broadcaster component may execute as a separate process or thread from the application program. In one embodiment, the broadcaster component provides functions (*e.g.*, methods of class) that can be invoked by the application programs. The primary functions provided may include a connect function that an application program invokes passing an indication of the broadcast channel to which the application program wants to connect. The application program may provide a callback routine that the broadcaster component invokes to notify the application program that the connection has been completed, that is the process enters the fully connected state. The broadcaster component may also provide an acquire message function that the application program can invoke to retrieve the next message that is broadcast on the broadcast channel. Alternatively, the application program may provide a callback routine (which may be a virtual function provided by the application program) that the broadcaster component invokes to notify the application program that a broadcast message has been received. Each broadcaster component allocates a call-in port using the hashing algorithm. When calls are answered at the call-in port, they are transferred to other ports that serve as the external and internal ports.

The computers connecting to the broadcast channel may include a central processing unit, memory, input devices (*e.g.*, keyboard and pointing device), output devices (*e.g.*, display devices), and storage devices (*e.g.*, disk drives). The memory and storage devices are computer-readable medium that may contain computer instructions that implement the broadcaster component. In addition, the data structures and message structures may be stored or transmitted via a signal transmitted on a computer-readable media, such as a communications link.

Figure 7 is a block diagram illustrating the sub-components of the broadcaster component in one embodiment. The broadcaster component includes a connect component 701, an external dispatcher 702, an internal dispatcher 703 for each internal connection, an acquire message component 704 and a broadcast component 712. The application program may provide a connect callback component 710 and a receive response component 711 that are invoked by the broadcaster component. The application program invokes the connect component to establish a connection to a designated broadcast channel. The connect component identifies the external port and installs the external dispatcher for handling messages that are received on the external port. The connect component invokes the seek portal computer component 705 to identify a portal computer that is connected to the broadcast channel and invokes the connect request component 706 to ask the portal computer (if fully connected) to select neighbor processes for the newly connecting process. The external dispatcher receives external messages, identifies the type of message, and invokes the appropriate handling routine 707. The internal dispatcher receives the internal messages, identifies the type of message, and invokes the appropriate handling routine 708. The received broadcast messages are stored in the broadcast message queue 709. The acquire message component is invoked to retrieve messages from the broadcast queue. The broadcast component is invoked by the application program to broadcast messages in the broadcast channel.

The following tables list messages sent by the broadcaster components.

EXTERNAL MESSAGES

Message Type	Description
seeking_connection_call	Indicates that a seeking process would like to know whether the receiving process is fully connected to the broadcast channel
connection_request_call	Indicates that the sending process would like the receiving process to initiate a connection of the sending process to the broadcast channel
edge_proposal_call	Indicates that the sending process is proposing an edge through which the receiving process can connect to the broadcast channel (*i.e.*, edge pinning)
port_connection_call	Indicates that the sending process is proposing a port through which the receiving process can connect to the broadcast channel
connected_stmt	Indicates that the sending process is connected to the broadcast channel
condition_repair_stmt	Indicates that the receiving process should disconnect from one of its neighbors and connect to one of the processes involved in the neighbors with empty port condition

INTERNAL MESSAGES

Message Type	Description
broadcast_stmt	Indicates a message that is being broadcast through the broadcast channel for the application programs
connection_port_search_stmt	Indicates that the designated process is looking for a port through which it can connect to the broadcast channel
connection_edge_search_call	Indicates that the requesting process is looking for an edge through which it can connect to the broadcast channel
connection_edge_search_resp	Indicates whether the edge between this process and the sending neighbor has been accepted by the requesting party
diameter_estimate_stmt	Indicates an estimated diameter of the broadcast channel
diameter_reset_stmt	Indicates to reset the estimated diameter to indicated diameter
disconnect_stmt	Indicates that the sending neighbor is disconnecting from the broadcast channel
condition_check_stmt	Indicates that neighbors with empty port condition have

	been detected
condition_double_check_stmt	Indicates that the neighbors with empty ports have the same set of neighbors
shutdown_stmt	Indicates that the broadcast channel is being shutdown

Flow Diagrams

Figures 8-34 are flow diagrams illustrating the processing of the broadcaster component in one embodiment. Figure 8 is a flow diagram illustrating the processing of the connect routine in one embodiment. This routine is passed a channel type (*e.g.*, application name) and channel instance (*e.g.*, session identifier), that identifies the broadcast channel to which this process wants to connect. The routine is also passed auxiliary information that includes the list of portal computers and a connection callback routine. When the connection is established, the connection callback routine is invoked to notify the application program. When this process invokes this routine, it is in the seeking connection state. When a portal computer is located that is connected and this routine connects to at least one neighbor, this process enters the partially connected state, and when the process eventually connects to four neighbors, it enters the fully connected state. When in the small regime, a fully connected process may have less than four neighbors. In block 801, the routine opens the call-in port through which the process is to communicate with other processes when establishing external and internal connections. The port is selected as the first available port using the hashing algorithm described above. In block 802, the routine sets the connect time to the current time. The connect time is used to identify the instance of the process that is connected through this external port. One process may connect to a broadcast channel of a certain channel type and channel instance using one call-in port and then disconnects, and another process may then connect to that same broadcast channel using the same call-in port. Before the other process becomes fully connected, another process may try to communicate with it thinking it is the fully connected old process. In such a case, the connect time can be used to identify this situation. In block 803, the routine invokes the seek portal computer routine passing the channel type and channel instance. The seek portal computer routine attempts to locate a portal computer through which this process can connect to the broadcast channel for the passed type and instance. In decision block 804, if the seek portal computer routine is successful in locating a fully connected process on that portal computer, then the routine continues at block 805, else the routine returns an unsuccessful indication. In decision block 805, if no portal computer other than the portal computer on which the process is executing was located, then this is the first process to fully connect to broadcast channel and the routine continues at block 806, else the routine continues at block 808. In block 806, the routine invokes the achieve connection routine to change the state of this process to fully connected. In block 807, the routine installs the external dispatcher for processing messages received through this process' external port for the passed channel type and channel instance. When a message is received through that external port, the external dispatcher is invoked. The routine then returns. In block 808, the routine installs an external dispatcher. In block 809, the routine invokes the connect request routine to initiate the process of identifying neighbors for the seeking computer. The routine then returns.

Figure 9 is a flow diagram illustrating the processing of the seek portal computer routine in one embodiment. This routine is passed the channel type and channel instance of the broadcast channel to which this process wishes to connect. This routine, for each search depth (*e.g.*, port number), checks the portal computers at that search depth. If a portal computer is located at that search depth with a process that is fully connected to the broadcast channel, then the routine returns an indication of success. In blocks 902-911, the routine loops selecting each search depth until a process is located. In block 902, the routine selects the next search depth using a port number ordering algorithm. In decision block 903, if all the search depths have already been selected during this execution of the loop, that is for the currently selected depth, then the routine returns a failure indication, else the routine continues at block 904. In blocks 904-911, the routine loops selecting each portal computer and determining whether a process of that portal computer is connected to (or attempting to connect to) the broadcast channel with the passed channel type and channel instance. In block 904, the routine selects the next portal computer. In decision block 905, if all the portal computers have already been selected, then the routine loops to block 902 to select the next search depth, else the routine continues at block 906. In block 906, the routine dials the selected portal computer through the port represented by the search depth. In decision block 907, if the dialing was successful, then the routine continues at block 908, else the routine loops to block 904 to select the next portal computer. The dialing will be successful if the dialed port is the call-in port of the broadcast channel of the passed channel type and channel instance of a process executing on that portal computer. In block 908, the routine invokes a contact process routine, which contacts the answering process of the portal computer through the dialed port and determines whether that process is fully connected to the broadcast channel. In block 909, the routine hangs up on the selected portal computer. In decision block 910, if the answering process is fully connected to the broadcast channel, then the routine returns a success indicator, else the routine continues at block 911. In block 911, the routine invokes the check for external call routine to determine whether an external call has been made to this process as a portal computer and processes that call. The routine then loops to block 904 to select the next portal computer.

Figure 10 is a flow diagram illustrating the processing of the contact process routine in one embodiment. This routine determines whether the process of the selected portal computer that answered the call-in to the selected port is fully connected to the broadcast channel. In block 1001, the routine sends an external message (*i.e.*, seeking_connection_call) to the answering process indicating that a seeking process wants to know whether the answering process is fully connected to the broadcast channel. In block 1002, the routine receives the external response message from the answering process. In decision block 1003, if the external response message is successfully received (*i.e.*, seeking_connection_resp), then the routine continues at block 1004, else the routine returns. Wherever the broadcast component requests to receive an external message, it sets a time out period. If the external message is not received within that time out period, the broadcaster component checks its own call-in port to see if another process is calling it. In particular, the dialed process may be calling the dialing process, which may result in a deadlock situation. The broadcaster component may repeat the receive request several times. If the expected message is not received, then the broadcaster component handles the error as appropriate. In decision block 1004, if the answering process indicates in its response message that it is fully connected to the broadcast channel, then the routine continues at block 1005, else the routine continues at block 1006. In block 1005, the routine adds the selected portal computer to a list of connected portal computers and then returns. In block 1006, the routine adds the answering process to a list of fellow seeking processes and then returns.

Figure 11 is a flow diagram illustrating the processing of the connect request routine in one embodiment. This routine requests a process of a portal computer that was identified as being fully connected to the broadcast channel to initiate the connection of this process to the broadcast channel. In decision block 1101, if at least one process of a portal computer was located that is fully connected to the broadcast channel, then the routine continues at block 1103, else the routine continues at block 1102. A process of the portal computer may no longer be in the list if it recently disconnected from the broadcast channel. In one embodiment, a seeking computer may always search its entire search depth and find multiple portal computers through which it can connect to the broadcast channel. In block 1102, the routine restarts the process of connecting to the broadcast channel and returns. In block 1103, the routine dials the process of one of the found portal computers through the call-in port. In decision block 1104, if the dialing is successful, then the routine continues at block 1105, else the routine continues at block 1113. The dialing may be unsuccessful if, for example, the dialed process recently disconnected from the broadcast channel. In block 1105, the routine sends an external message to the dialed process requesting a connection to the broadcast channel (*i.e.*, connection_request_call). In block 1106, the routine receives the response message (*i.e.*, connection_request_resp). In decision block 1107, if the response message is successfully received, then the routine continues at block 1108, else the routine continues at block 1113. In block 1108, the routine sets the expected number of holes (*i.e.*, empty internal connections) for this process based on the received response. When in the large regime, the expected number of holes is zero. When in the small regime, the expected number of holes varies from one to three. In block 1109, the routine sets the estimated diameter of the broadcast channel based on the received response. In decision block 1111, if the dialed process is ready to connect to this process as indicated by the response message, then the routine continues at block 1112, else the routine continues at block 1113. In block 1112, the routine invokes the add neighbor routine to add the answering process as a neighbor to this process. This adding of the answering process typically occurs when the broadcast channel is in the small regime. When in the large regime, the random walk search for a neighbor is performed. In block 1113, the routine hangs up the external connection with the answering process computer and then returns.

Figure 12 is a flow diagram of the processing of the check for external call routine in one embodiment. This routine is invoked to identify whether a fellow seeking process is attempting to establish a connection to the broadcast channel through this process. In block 1201, the routine attempts to answer a call on the call-in port. In decision block 1202, if the answer is successful, then the routine continues at block 1203, else the routine returns. In block 1203, the routine receives the external message from the external port. In decision block 1204, if the type of the message indicates that a seeking process is calling (*i.e.*, seeking_connection_call), then the routine continues at block 1205, else the routine returns. In block 1205, the routine sends an external message (*i.e.*, seeking_connection_resp) to the other seeking process indicating that this process is also is seeking a connection. In decision block 1206, if the sending of the external message is successful, then the routine continues at block 1207, else the routine returns. In block 1207, the routine adds the other seeking process to a list of fellow seeking processes and then returns. This list may be used if this process can find no process that is fully connected to the broadcast channel. In which case, this process may check to see if any fellow seeking process were successful in connecting to the broadcast channel. For example, a fellow seeking process may become the first process fully connected to the broadcast channel.

Figure 13 is a flow diagram of the processing of the achieve connection routine in one embodiment. This routine sets the state of this process to fully connected to the broadcast channel and invokes a callback routine to notify the application program that the process is now fully connected to the requested broadcast channel. In block 1301, the routine sets the connection state of this process to fully connected. In block 1302, the routine notifies fellow seeking processes that it is fully connected by sending a connected external message to them (*i.e.*, connected_stmt). In block 1303, the routine invokes the connect callback routine to notify the application program and then returns.

Figure 14 is a flow diagram illustrating the processing of the external dispatcher routine in one embodiment. This routine is invoked when the external port receives a message. This routine retrieves the message, identifies the external message type, and invokes the appropriate routine to handle that message. This routine loops processing each message until all the received messages have been handled. In block 1401, the routine answers (*e.g.*, picks up) the external port and retrieves an external message. In decision block 1402, if a message was retrieved, then the routine continues at block 1403, else the routine hangs up on the external port in block 1415 and returns. In decision block 1403, if the message type is for a process seeking a connection (*i.e.*, seeking_connection_call), then the routine invokes the handle seeking connection call routine in block 1404, else the routine continues at block 1405. In decision block 1405, if the message type is for a connection request call (*i.e.*, connection_request_call), then the routine invokes the handle connection request call routine in block 1406, else the routine continues at block 1407. In decision block 1407, if the message type is edge proposal call (*i.e.*, edge_proposal_call), then the routine invokes the handle edge proposal call routine in block 1408, else the routine continues at block 1409. In decision block 1409, if the message type is port connect call (*i.e.*, port_connect_call), then the routine invokes the handle port connection call routine in block 1410, else the routine continues at block 1411. In decision block 1411, if the message type is a connected statement (*i.e.*, connected_stmt), the routine invokes the handle connected statement in block 1112, else the routine continues at block 1212. In decision block 1412, if the message type is a condition repair statement (*i.e.*, condition_repair_stmt), then the routine invokes the handle condition repair routine in block 1413, else the routine loops to block 1414 to process the next message. After each handling routine is invoked, the routine loops to block 1414. In block 1414, the routine hangs up on the external port and continues at block 1401 to receive the next message.

Figure 15 is a flow diagram illustrating the processing of the handle seeking connection call routine in one embodiment. This routine is invoked when a seeking process is calling to identify a portal computer through which it can connect to the broadcast channel. In decision block 1501, if this process is currently fully connected to the broadcast channel identified in the message, then the routine continues at block 1502, else the routine continues at block 1503. In block 1502, the routine sets a message to indicate that this process is fully connected to the broadcast channel and continues at block 1505. In block 1503, the routine sets a message to indicate that this process is not fully connected. In block 1504, the routine adds the identification of the seeking process to a list of fellow seeking processes. If this process is not fully connected, then it is attempting to connect to the broadcast channel. In block 1505, the routine sends the external message response (*i.e.*, seeking_connection_resp) to the seeking process and then returns.

Figure 16 is a flow diagram illustrating processing of the handle connection request call routine in one embodiment. This routine is invoked when the calling process wants this process to initiate the connection of the process to the broadcast channel. This routine either allows the calling process to establish an internal connection with this process (*e.g.*, if in the small regime) or starts the process of identifying a process to which the calling process can connect. In decision block 1601, if this process is currently fully connected to the broadcast channel, then the routine continues at block 1603, else the routine hangs up on the external port in block 1602 and returns. In block 1603, the routine sets the number of holes that the calling process should expect in the response message. In block 1604, the routine sets the estimated diameter in the response message. In block 1605, the routine indicates whether this process is ready to connect to the calling process. This process is ready to connect when the number of its holes is greater than zero and the calling process is not a neighbor of this process. In block 1606, the routine sends to the calling process an external message that is responsive to the connection request call (*i.e.*, connection_request_resp). In block 1607, the routine notes the number of holes that the calling process needs to fill as indicated in the request message. In decision block 1608, if this process is ready to connect to the calling process, then the routine continues at block 1609, else the routine continues at block 1611. In block 1609, the routine invokes the add neighbor routine to add the calling process as a neighbor. In block 1610, the routine decrements the number of holes that the calling process needs to fill and continues at block 1611. In block 1611, the routine hangs up on the external port. In decision block 1612, if this process has no holes or the estimated diameter is greater than one (*i.e.*, in the large regime), then the routine continues at block 1613, else the routine continues at block 1616. In blocks 1613-1615, the routine loops forwarding a request for an edge through which to connect to the calling process to the broadcast channel. One request is forwarded for each pair of holes of the calling process that needs to be filled. In decision block 1613, if the number of holes of the calling process to be filled is greater than or equal to two, then the routine continues at block 1614, else the routine continues at block 1616. In block 1614, the routine invokes the forward connection edge search routine. The invoked routine is passed to an indication of the calling process and the random walk distance. In one embodiment, the distance is twice in the estimated diameter of the broadcast channel. In block 1614, the routine decrements the holes left to fill by two and loops to block 1613. In decision block 1616, if there is still a hole to fill, then the routine continues at block 1617, else the routine returns. In block 1617, the routine invokes the fill hole routine passing the identification of the calling process. The fill hole routine broadcasts a connection port search statement (*i.e.*, connection_port_search_stmt) for a hole of a connected process through which the calling process can connect to the broadcast channel. The routine then returns.

Figure 17 is a flow diagram illustrating the processing of the add neighbor routine in one embodiment. This routine adds the process calling on the external port as a neighbor to this process. In block 1701, the routine identifies the calling process on the external port. In block 1702, the routine sets a flag to indicate that the neighbor has not yet received the broadcast messages from this process. This flag is used to ensure that there are no gaps in the messages initially sent to the new neighbor. The external port becomes the internal port for this connection. In decision block 1703, if this process is in the seeking connection state, then this process is connecting to its first neighbor and the routine continues at block 1704, else the routine continues at block 1705. In block 1704, the routine sets the connection state of this process to partially connected. In block 1705, the routine adds the calling process to the list of neighbors of this process. In block 1706, the routine installs an internal dispatcher for the new neighbor. The internal dispatcher is invoked when a message is received from that new neighbor through the internal port of that new neighbor. In decision block 1707, if this process buffered up messages while not fully connected, then the routine continues at block 1708, else the routine continues at block 1709. In one embodiment, a process that is partially connected may buffer the messages that it receives through an internal connection so that it can send these messages as it connects to new neighbors. In block 1708, the routine sends the buffered messages to the new neighbor through the internal port. In decision block 1709, if the number of holes of this process equals the expected number of holes, then this process is fully connected and the routine continues at block 1710, else the routine continues at block 1711. In block 1710, the routine invokes the achieve connected routine to indicate that this process is fully connected. In decision block 1711, if the number of holes for this process is zero, then the routine continues at block 1712, else the routine returns. In block 1712, the routine deletes any pending edges and then returns. A pending edge is an edge that has been proposed to this process for edge pinning, which in this case is no longer needed.

Figure 18 is a flow diagram illustrating the processing of the forward connection edge search routine in one embodiment. This routine is responsible for passing along a request to connect a requesting process to a randomly selected neighbor of this process through the internal port of the selected neighbor, that is part of the random walk. In decision block 1801, if the forwarding distance remaining is greater than zero, then the routine continues at block 1804, else the routine continues at block 1802. In decision block 1802, if the number of neighbors of this process is greater than one, then the routine continues at block 1804, else this broadcast channel is in the small regime and the routine continues at block 1803. In decision block 1803, if the requesting process is a neighbor of this process, then the routine returns, else the routine continues at block 1804. In blocks 1804-1807, the routine loops attempting to send a connection edge search call internal message (*i.e.*, connection_edge_search_call) to a randomly selected neighbor. In block 1804, the routine randomly selects a neighbor of this process. In decision block 1805, if all the neighbors of this process have already been selected, then the routine cannot forward the message and the routine returns, else the routine continues at block 1806. In block 1806, the routine sends a connection edge search call internal message to the selected neighbor. In decision block 1807, if the sending of the message is successful, then the routine continues at block 1808, else the routine loops to block 1804 to select the next neighbor. When the sending of an internal message is unsuccessful, then the neighbor may have disconnected from the broadcast channel in an unplanned manner. Whenever such a situation is detected by the broadcaster component, it attempts to find another neighbor by invoking the fill holes routine to fill a single hole or the forward connecting edge search routine to fill two holes. In block 1808, the routine notes that the recently sent connection edge search call has not yet been acknowledged and indicates that the edge to this neighbor is reserved if the remaining forwarding distance is less than or equal to one. It is reserved because the selected neighbor may offer this edge to the requesting process for edge pinning. The routine then returns.

Figure 19 is a flow diagram illustrating the processing of the handle edge proposal call routine. This routine is invoked when a message is received from a proposing process that proposes to connect an edge between the proposing process and one of its neighbors to this process for edge pinning. In decision block 1901, if the number of holes of this process minus the number of pending edges is greater than or equal to one, then this process still has holes to be filled and the routine continues at block 1902, else the routine continues at block 1911. In decision block 1902, if the proposing process or its neighbor is a neighbor of this process, then the routine continues at block 1911, else the routine continues at block 1903. In block 1903, the routine indicates that the edge is pending between this process and the proposing process. In decision block 1904, if a proposed neighbor is already pending as a proposed neighbor, then the routine continues at block 1911, else the routine continues at block 1907. In block 1907, the routine sends an edge proposal response as an external message to the proposing process (*i.e.*, edge_proposal_resp) indicating that the proposed edge is accepted. In decision block 1908, if the sending of the message was successful, then the routine continues at block 1909, else the routine returns. In block 1909, the routine adds the edge as a pending edge. In block 1910, the routine invokes the add neighbor routine to add the proposing process on the external port as a neighbor. The routine then returns. In block 1911, the routine sends an external message (*i.e.*, edge_proposal_resp) indicating that this proposed edge is not accepted. In decision block 1912, if the number of holes is odd, then the routine continues at block 1913, else the routine returns. In block 1913, the routine invokes the fill hole routine and then returns.

Figure 20 is a flow diagram illustrating the processing of the handle port connection call routine in one embodiment. This routine is invoked when an external message is received then indicates that the sending process wants to connect to one hole of this process. In decision block 2001, if the number of holes of this process is greater than zero, then the routine continues at block 2002, else the routine continues at block 2003. In decision block 2002, if the sending process is not a neighbor, then the routine continues at block 2004, else the routine continues to block 2003. In block 2003, the routine sends a port connection response external message (*i.e.*, port_connection_resp) to the sending process that indicates that it is not okay to connect to this process. The routine then returns. In block 2004, the routine sends a port connection response external message to the sending process that indicates that is okay to connect this process. In decision block 2005, if the sending of the message was successful, then the routine continues at block 2006, else the routine continues at block 2007. In block 2006, the routine invokes the add neighbor routine to add the sending process as a neighbor of this process and then returns. In block 2007, the routine hangs up the external connection. In block 2008, the routine invokes the connect request routine to request that a process connect to one of the holes of this process. The routine then returns.

Figure 21 is a flow diagram illustrating the processing of the fill hole routine in one embodiment. This routine is passed an indication of the requesting process. If this process is requesting to fill a hole, then this routine sends an internal message to other processes. If another process is requesting to fill a hole, then this routine invokes the routine to handle a connection port search request. In block 2101, the routine initializes a connection port search statement internal message (*i.e.*, connection_port_search_stmt). In decision block 2102, if this process is the requesting process, then the routine continues at block 2103, else the routine continues at block 2104. In block 2103, the routine distributes the message to the neighbors of this process through the internal ports and then returns. In block 2104, the routine invokes the handle connection port search routine and then returns.

Figure 22 is a flow diagram illustrating the processing of the internal dispatcher routine in one embodiment. This routine is passed an indication of the neighbor who sent the internal message. In block 2201, the routine receives the internal message. This routine identifies the message type and invokes the appropriate routine to handle the message. In block 2202, the routine assesses whether to change the estimated diameter of the broadcast channel based on the information in the received message. In decision block 2203, if this process is the originating process of the message or the message has already been received (*i.e.*, a duplicate), then the routine ignores the message and continues at block 2208, else the routine continues at block 2203A. In decision block 2203A, if the process is partially connected, then the routine continues at block 2203B, else the routine continues at block 2204. In block 2203B, the routine adds the message to the pending connection buffer and continues at block 2204. In decision blocks 2204-2207, the routine decodes the message type and invokes the appropriate routine to handle the message. For example, in decision block 2204, if the type of the message is broadcast statement (*i.e.*, broadcast_stmt), then the routine invokes the handle broadcast message routine in block 2205. After invoking the appropriate handling routine, the routine continues at block 2208. In decision block 2208, if the partially connected buffer is full, then the routine continues at block 2209, else the routine continues at block 2210. The broadcaster component collects all its internal messages in a buffer while partially connected so that it can forward the messages as it connects to new neighbors. If, however, that buffer becomes full, then the process assumes that it is now fully connected and that the expected number of connections was too high, because the broadcast channel is now in the small regime. In block 2209, the routine invokes the achieve connection routine and then continues in block 2210. In decision block 2210, if the application program message queue is empty, then the routine returns, else the routine continues at block 2212. In block 2212, the routine invokes the receive response routine passing the acquired message and then returns. The received response routine is a callback routine of the application program.

Figure 23 is a flow diagram illustrating the processing of the handle broadcast message routine in one embodiment. This routine is passed an indication of the originating process, an indication of the neighbor who sent the broadcast message, and the broadcast message itself. In block 2301, the routine performs the out of order processing for this message. The broadcaster component queues messages from each originating process until it can send them in sequence number order to the application program. In block 2302, the routine invokes the distribute broadcast message routine to forward the message to the neighbors of this process. In decision block 2303, if a newly connected neighbor is waiting to receive messages, then the routine continues at block 2304, else the routine returns. In block 2304, the routine sends the messages in the correct order if possible for each originating process and then returns.

Figure 24 is a flow diagram illustrating the processing of the distribute broadcast message routine in one embodiment. This routine sends the broadcast message to each of the neighbors of this process, except for the neighbor who sent the message to this process. In block 2401, the routine selects the next neighbor other than the neighbor who sent the message. In decision block 2402, if all such neighbors have already been selected, then the routine returns. In block 2403, the routine sends the message to the selected neighbor and then loops to block 2401 to select the next neighbor.

Figure 26 is a flow diagram illustrating the processing of the handle connection port search statement routine in one embodiment. This routine is passed an indication of the neighbor that sent the message and the message itself. In block 2601, the routine invokes the distribute internal message which sends the message to each of its neighbors other than the sending neighbor. In decision block 2602, if the number of holes of this process is greater than zero, then the routine continues at block 2603, else the routine returns. In decision block 2603, if the requesting process is a neighbor, then the routine continues at block 2605, else the routine continues at block 2604. In block 2604, the routine invokes the court neighbor routine and then returns. The court neighbor routine connects this process to the requesting process if possible. In block 2605, if this process has one hole, then the neighbors with empty ports condition exists and the routine continues at block 2606, else the routine returns. In block 2606, the routine generates a condition check message (*i.e.*, condition_check) that includes a list of this process' neighbors. In block 2607, the routine sends the message to the requesting neighbor.

Figure 27 is a flow diagram illustrating the processing of the court neighbor routine in one embodiment. This routine is passed an indication of the prospective neighbor for this process. If this process can connect to the prospective neighbor, then it sends a port connection call external message to the prospective neighbor and adds the prospective neighbor as a neighbor. In decision block 2701, if the prospective neighbor is already a neighbor, then the routine returns, else the routine continues at block 2702. In block 2702, the routine dials the prospective neighbor. In decision block 2703, if the number of holes of this process is greater than zero, then the routine continues at block 2704, else the routine continues at block 2706. In block 2704, the routine sends a port connection call external message (*i.e.*, port_connection_call) to the prospective neighbor and receives its response (*i.e.*, port_connection_resp). Assuming the response is successfully received, in block 2705, the routine adds the prospective neighbor as a neighbor of this process by invoking the add neighbor routine. In block 2706, the routine hangs up with the prospect and then returns.

Figure 28 is a flow diagram illustrating the processing of the handle connection edge search call routine in one embodiment. This routine is passed a indication of the neighbor who sent the message and the message itself. This routine either forwards the message to a neighbor or proposes the edge between this process and the sending neighbor to the requesting process for edge pinning. In decision block 2801, if this process is not the requesting process or the number of holes of the requesting process is still greater than or equal to two, then the routine continues at block 2802, else the routine continues at block 2813. In decision block 2802, if the forwarding distance is greater than zero, then the random walk is not complete and the routine continues at block 2803, else the routine continues at block 2804. In block 2803, the routine invokes the forward connection edge search routine passing the identification of the requesting process and the decremented forwarding distance. The routine then continues at block 2815. In decision block 2804, if the requesting process is a neighbor or the edge between this process and the sending neighbor is reserved because it has already been offered to a process, then the routine continues at block 2805, else the routine continues at block 2806. In block 2805, the routine invokes the forward connection edge search routine passing an indication of the requesting party and a toggle indicator that alternatively indicates to continue the random walk for one or two more computers. The routine then continues at block 2815. In block 2806, the routine dials the requesting process via the call-in port. In block 2807, the routine sends an edge proposal call external message (*i.e.*, edge_proposal_call) and receives the response (*i.e.*, edge_proposal_resp). Assuming that the response is successfully received, the routine continues at block 2808. In decision block 2808, if the response indicates that the edge is acceptable to the requesting process, then the routine continues at block 2809, else the routine continues at block 2812. In block 2809, the routine reserves the edge between this process and the sending neighbor. In block 2810, the routine adds the requesting process as a neighbor by invoking the add neighbor routine. In block 2811, the routine removes the sending neighbor as a neighbor. In block 2812, the routine hangs up the external port and continues at block 2815. In decision block 2813, if this process is the requesting process and the number of holes of this process equals one, then the routine continues at block 2814, else the routine continues at block 2815. In block 2814, the routine invokes the fill hole routine. In block 2815, the routine sends an connection edge search response message (*i.e.*, connection_edge_search_response) to the sending neighbor indicating acknowledgement and then returns. The graphs are sensitive to parity. That is, all possible paths starting from a node and ending at that node will have an even length unless the graph has a cycle whose length is odd. The broadcaster component uses a toggle indicator to vary the random walk distance between even and odd distances.

Figure 29 is a flow diagram illustrating the processing of the handle connection edge search response routine in one embodiment. This routine is passed as indication of the requesting process, the sending neighbor, and the message. In block 2901, the routine notes that the connection edge search response (*i.e.*, connection_edge_search_resp) has been received and if the forwarding distance is less than or equal to one unreserves the edge between this process and the sending neighbor. In decision block 2902, if the requesting process indicates that the edge is acceptable as indicated in the message, then the routine continues at block 2903, else the routine returns. In block 2903, the routine reserves the edge between this process and the sending neighbor. In block 2904, the routine removes the sending neighbor as a neighbor. In block 2905, the routine invokes the court neighbor routine to connect to the requesting process. In decision block 2906, if the invoked routine was unsuccessful, then the routine continues at block 2907, else the routine returns. In decision block 2907, if the number of holes of this process is greater than zero, then the routine continues at block 2908, else the routine returns. In block 2908, the routine invokes the fill hole routine and then returns.

Figure 30 is a flow diagram illustrating the processing of the broadcast routine in one embodiment. This routine is invoked by the application program to broadcast a message on the broadcast channel. This routine is passed the message to be broadcast. In decision block 3001, if this process has at least one neighbor, then the routine continues at block 3002, else the routine returns since it is the only process connected to be broadcast channel. In block 3002, the routine generates an internal message of the broadcast statement type (*i.e.*, broadcast _stmt). In block 3003, the routine sets the sequence number of the message. In block 3004, the routine invokes the distribute internal message routine to broadcast the message on the broadcast channel. The routine returns.

Figure 31 is a flow diagram illustrating the processing of the acquire message routine in one embodiment. The acquire message routine may be invoked by the application program or by a callback routine provided by the application program. This routine returns a message. In block 3101, the routine pops the message from the message queue of the broadcast channel. In decision block 3102, if a message was retrieved, then the routine returns an indication of success, else the routine returns indication of failure.

Figures 32-34 are flow diagrams illustrating the processing of messages associated with the neighbors with empty ports condition. Figure 32 is a flow diagram illustrating processing of the handle condition check message in one embodiment. This message is sent by a neighbor process that has one hole and has received a request to connect to a hole of this process. In decision block 3201, if the number of holes of this process is equal to one, then the routine continues at block 3202, else the neighbors with empty ports condition does not exist any more and the routine returns. In decision block 3202, if the sending neighbor and this process have the same set of neighbors, the routine continues at block 3203, else the routine continues at block 3205. In block 3203, the routine initializes a condition double check message (*i.e.*, condition_double_check) with the list of neighbors of this process. In block 3204, the routine sends the message internally to a neighbor other than sending neighbor. The routine then returns. In block 3205, the routine selects a neighbor of the sending process that is not also a neighbor of this process. In block 3206, the routine sends a condition repair message (*i.e.*, condition_repair_stmt) externally to the selected process. In block 3207, the routine invokes the add neighbor routine to add the selected neighbor as a neighbor of this process and then returns.

Figure 33 is a flow diagram illustrating processing of the handle condition repair statement routine in one embodiment. This routine removes an existing neighbor and connects to the process that sent the message. In decision block 3301, if this process has no holes, then the routine continues at block 3302, else the routine continues at block 3304. In block 3302, the routine selects a neighbor that is not involved in the neighbors with empty ports condition. In block 3303, the routine removes the selected neighbor as a neighbor of this process. Thus, this process that is executing the routine now has at least one hole. In block 3304, the routine invokes the add neighbor routine to add the process that sent the message as a neighbor of this process. The routine then returns.

Figure 34 is a flow diagram illustrating the processing of the handle condition double check routine. This routine determines whether the neighbors with empty ports condition really is a problem or whether the broadcast channel is in the small regime. In decision block 3401, if this process has one hole, then the routine continues at block 3402, else the routine continues at block 3403. If this process does not have one hole, then the set of neighbors of this process is not the same as the set of neighbors of the sending process. In decision block 3402, if this process and the sending process have the same set of neighbors, then the broadcast channel is not in the small regime and the routine continues at block 3403, else the routine continues at block 3406. In decision block 3403, if this process has no holes, then the routine returns, else the routine continues at block 3404. In block 3404, the routine sets the estimated diameter for this process to one. In block 3405, the routine broadcasts a diameter reset internal message (*i.e.*, diameter_reset) indicating that the estimated diameter is one and then returns. In block 3406, the routine creates a list of neighbors of this process. In block 3407, the routine sends the condition check message (*i.e.*, condition_check_stmt) with the list of neighbors to the neighbor who sent the condition double check message and then returns.

From the above description, it will be appreciated that although specific embodiments of the technology have been described, various modifications may be made without deviating from the spirit and scope of the invention. For example, the communications on the broadcast channel may be encrypted. Also, the channel instance or session identifier may be a very large number (*e.g.*, 128 bits) to help prevent an unauthorized user to maliciously tap into a broadcast channel. The portal computer may also enforce security and not allow an unauthorized user to connect to the broadcast channel. Accordingly, the invention is not limited except by the claims.

CLAIMS

1. A method of broadcasting data through a computer network, the method comprising:

receiving at a computer the data from a neighbor computer;

determining whether the received data has already been transmitted from the receiving computer to its neighbor computers;

when it is determined that the data has already been transmitted, disregarding the received data; and

when it is determined that the data has not already been transmitted, transmitting the received data to neighbor computers of the receiving computer.

2. The method of claim 1 wherein the computer network is a 4-regular graph.

3. The method of claim 1 wherein the computer network implements a broadcast channel wherein the neighbor computers of the computer network are connected using point-to-point connections.

4. The method of claim 3 wherein the connections are TCP/IP connections.

5. The method of claim 1 wherein the computer network is a broadcast channel that is implemented using an underlying network that connects computers using point-to-point connections.

6. The method of claim 5 wherein the underlying network is the Internet.

7. A broadcaster component in a computer connected to a computer network, comprising:

an originating module that transmits data that originates from the computer to each of the neighbor computers;

a receiving module that receives multiple copies of data that originates from another computer, each copy of the data being received from a different neighbor computer; and

a forwarding module that transmits a copy of the received data to each neighbor computer other than that neighbor computer from which the copy was received.

8. The broadcaster component of claim 7 including

a sending module that provides a copy of the received data to an application program.

9. The broadcaster component of claim 7 wherein the computer network is a broadcast channel implemented using an underlying point-to-point computer network.

10. The broadcaster component of claim 7 including:

a locating module for locating a portal computer that is connected to the computer network.

11. The broadcaster component of claim 7 including:

a connecting module for connecting the computer to the computer network.

12. The broadcaster component of claim 7 including:

a portal module for initiating joining of a requesting computer to the computer network.

13. The broadcast component of claim 7 wherein the computer is connected to its neighbor computer using a point-to-point connection.

14. A method of broadcasting data on a computer network, the method comprising:

establishing connections between each computer of the computer network and at least three other computers of the computer network;

when a computer originates data, sending the data to each of the computers to which it is connected; and

when a computer receives data, sending a first copy of the data that it receives to each of the computers to which it is connected other than the computer from which it received the data.

15. The method of claim 14 wherein computers and connections of the computer network form an m-regular graph.

16. The method of claim 15 wherein each computer is connected to an even number of computers.

17. The method of claim 14 wherein the computers and connections of the computer network form an m-regular and m-connected graph.

18. The method of claim 17 wherein m is even.

19. The method of claim 17 wherein m is 4.

20. The method of claim 14 wherein the computers are connected using point-to-point connections.

21. The method of claim 14 wherein the computers are connected using the Internet.

22. A computer-readable medium containing instructions for controlling a computer system to broadcast data on a broadcast channel, by a method comprising:

establishing connections between each computer of the broadcast channel and three other computers of the broadcast channel using point-to-point connections;

when a computer originates data, sending the data to each of the computers to which it is connected; and

when a computer receives data, sending a copy of the data that it receives to each of the computers to which it is connected other than the computer from which it received the data.

23. The computer-readable medium of claim 22 wherein computers and connections of the computer network form an m-regular graph.

24. The computer-readable medium of claim 23 wherein each computer is connected to an even number of computers.

25. The computer-readable medium of claim 22 wherein the computers and connections of the broadcast channel form an m-regular and m-connected graph.

26. The computer-readable medium of claim 25 wherein m is even.

27. The computer-readable medium of claim 25 wherein m is 4.

28. The computer-readable medium of claim 22 wherein the computers are connected using the Internet.

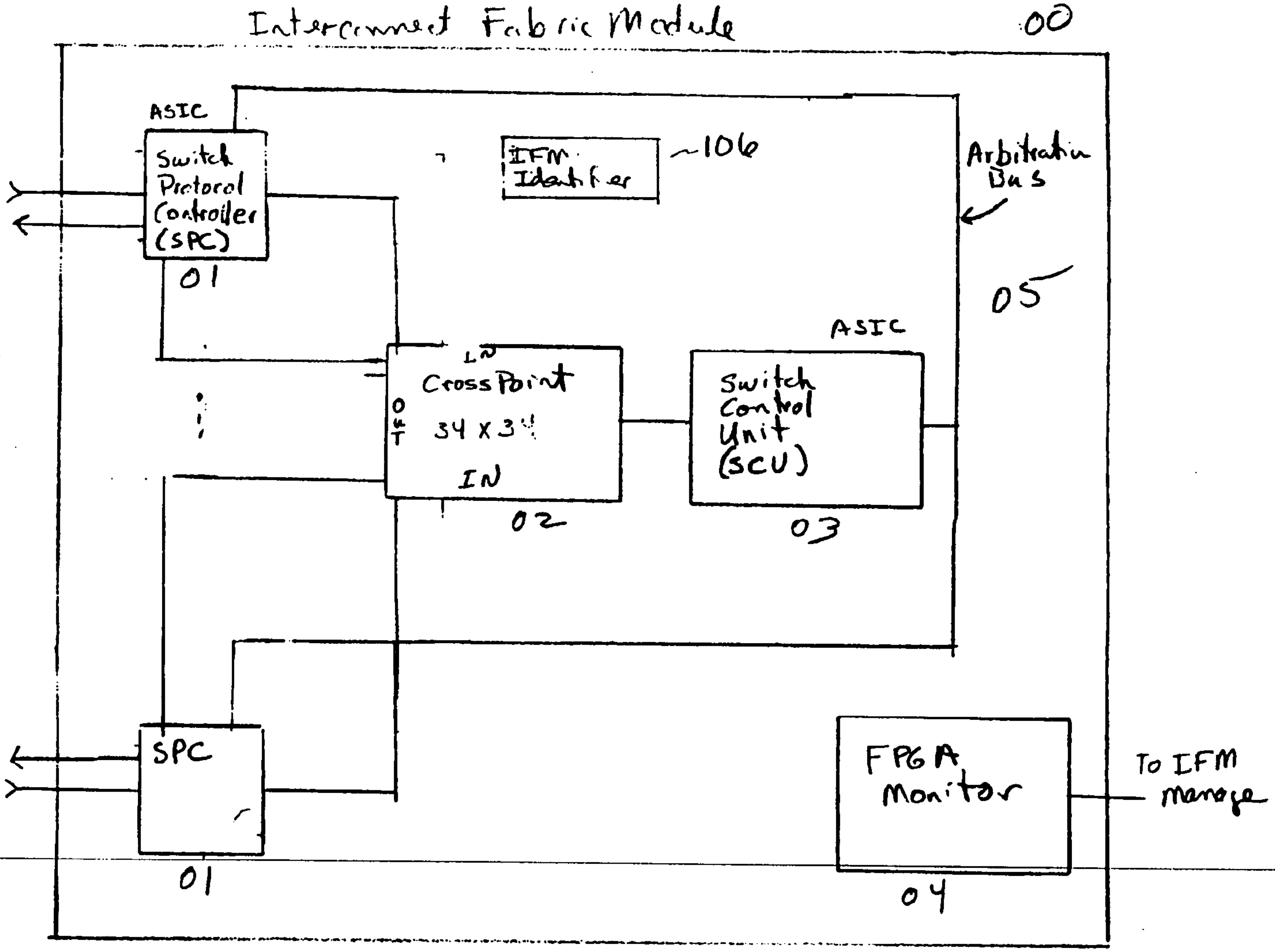

Interconnect Fabric Module
00
ASIC
Switch Protocol Controller (SPC)
01
IFM Identifier
~106
Arbitration Bus
05
IN
CrossPoint
34 X 34
OUT
IN
02
ASIC
Switch Control Unit (SCU)
03
SPC
01
FPGA Monitor
04
To IFM manage

Fig 1

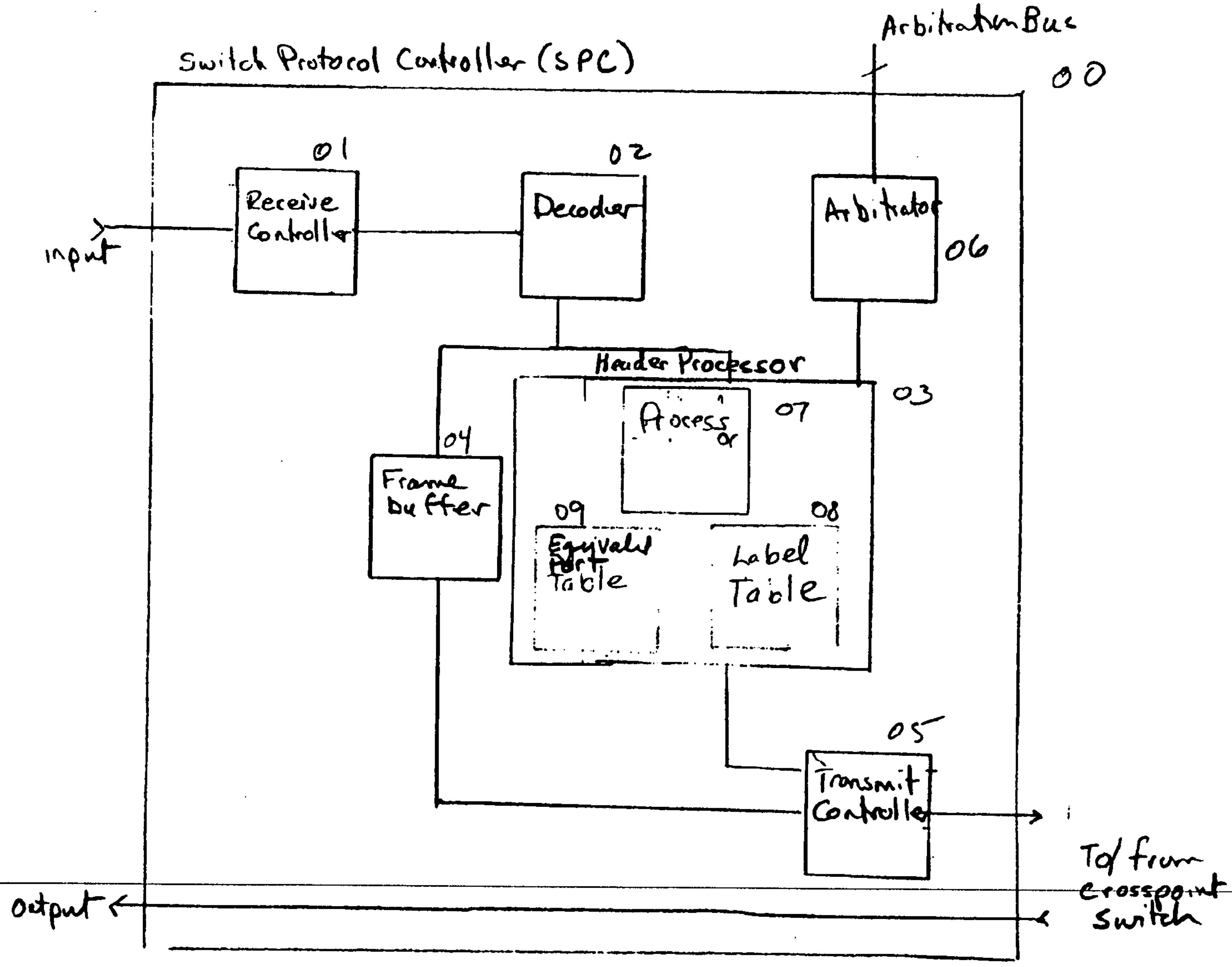

Switch Protocol Controller (SPC)
00
Arbitration Bus
01
Receive Controller
02
Decoder
Arbitrator
06
input
Header Processor
03
Processor
07
04
Frame buffer
09
Equivalent Port Table
08
Label Table
05
Transmit Controller
To/from crosspoint switch
output

Fig 2

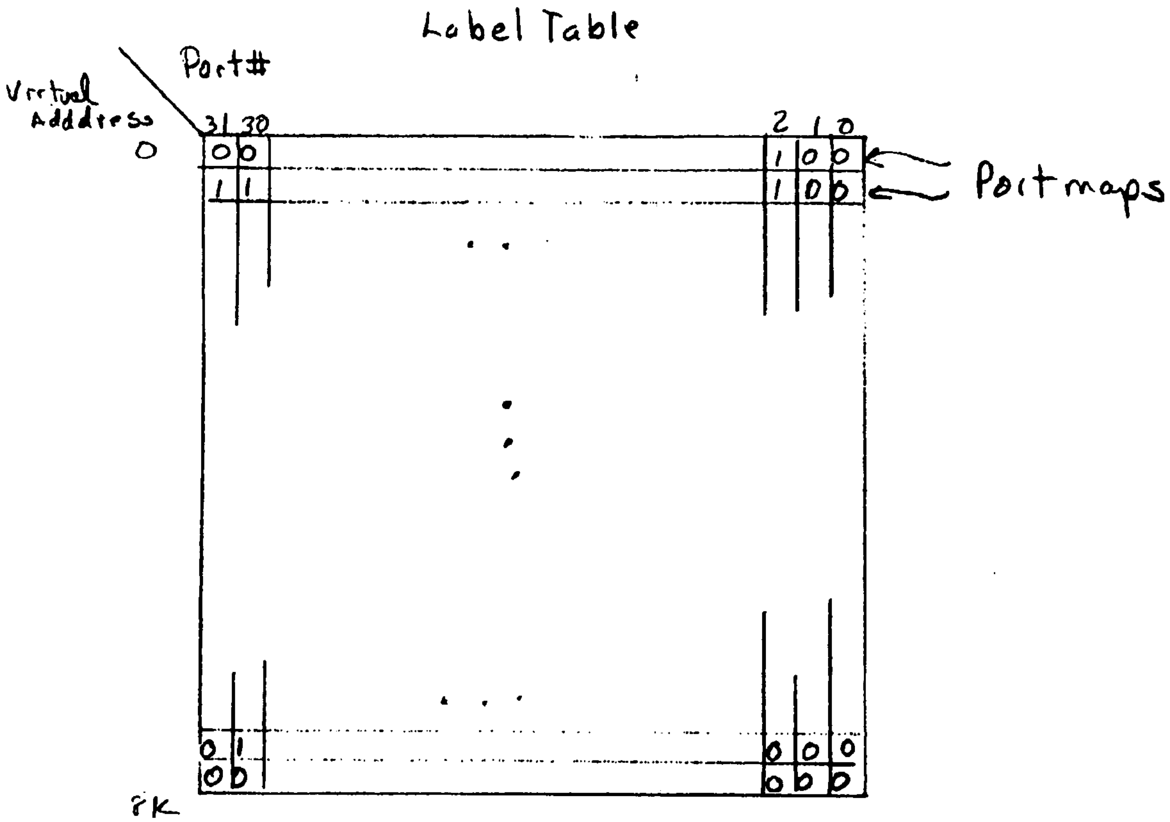

Label Table
Port#
Virtual Adddress
31 30
2 1 0
0
8K
0 0
1 1
1 0 0
1 0 0
0 1
0 0
0 0 0
0 0 0
Portmaps

Fig 3

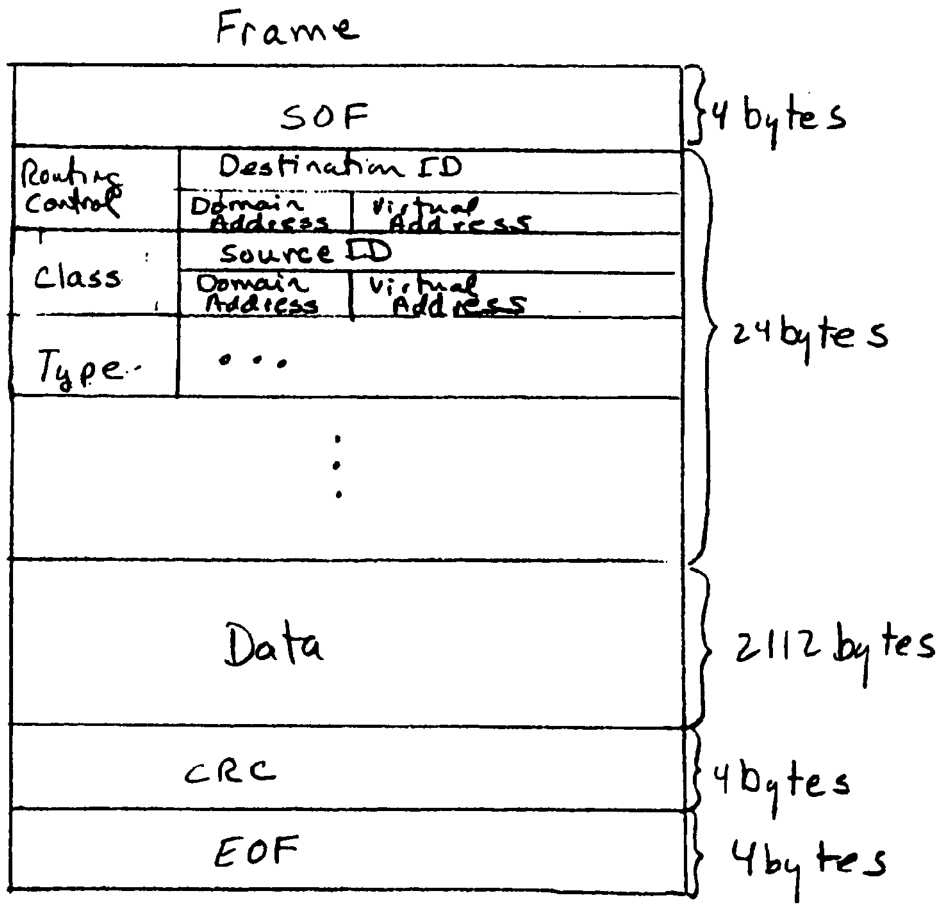

Frame
SOF
4 bytes
Routing Control
Destination ID
Domain Address
Virtual Address
Class
Source ID
Domain Address
Virtual Address
Type
24 bytes
Data
2112 bytes
CRC
4 bytes
EOF
4 bytes

Fig 4

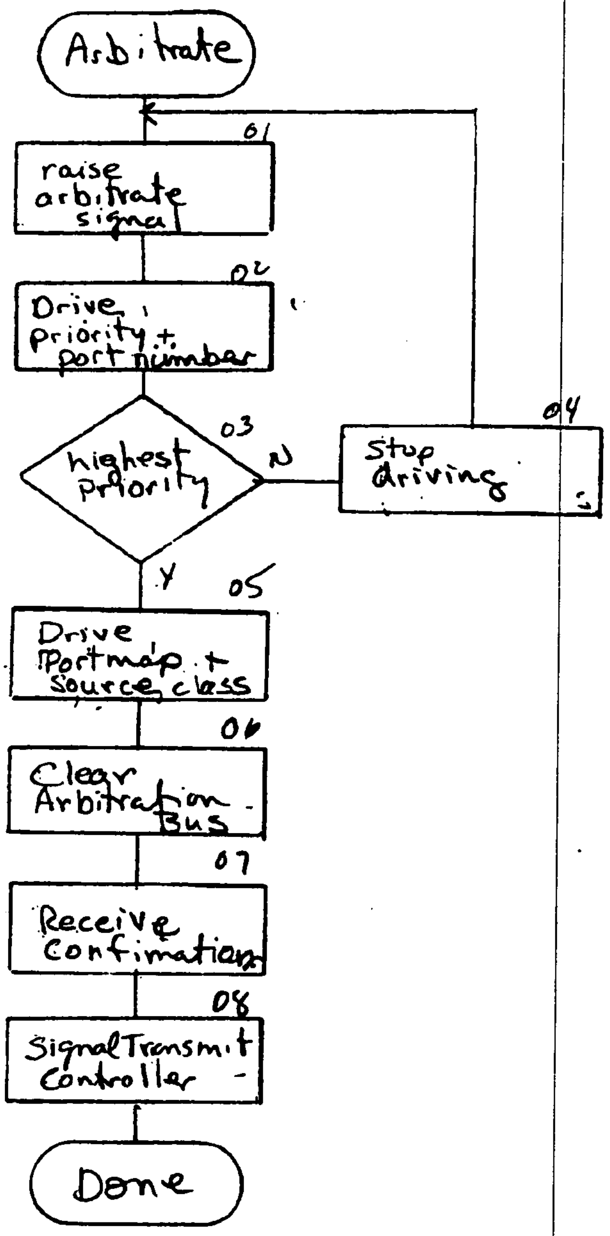

Arbitrate
01
raise arbitrate signal
02
Drive priority + port number
03
highest priority
N
04
Stop driving
Y
05
Drive Portmap + source class
06
Clear Arbitration Bus
07
Receive Confimation
08
Signal Transmit Controller
Done

Fig 5

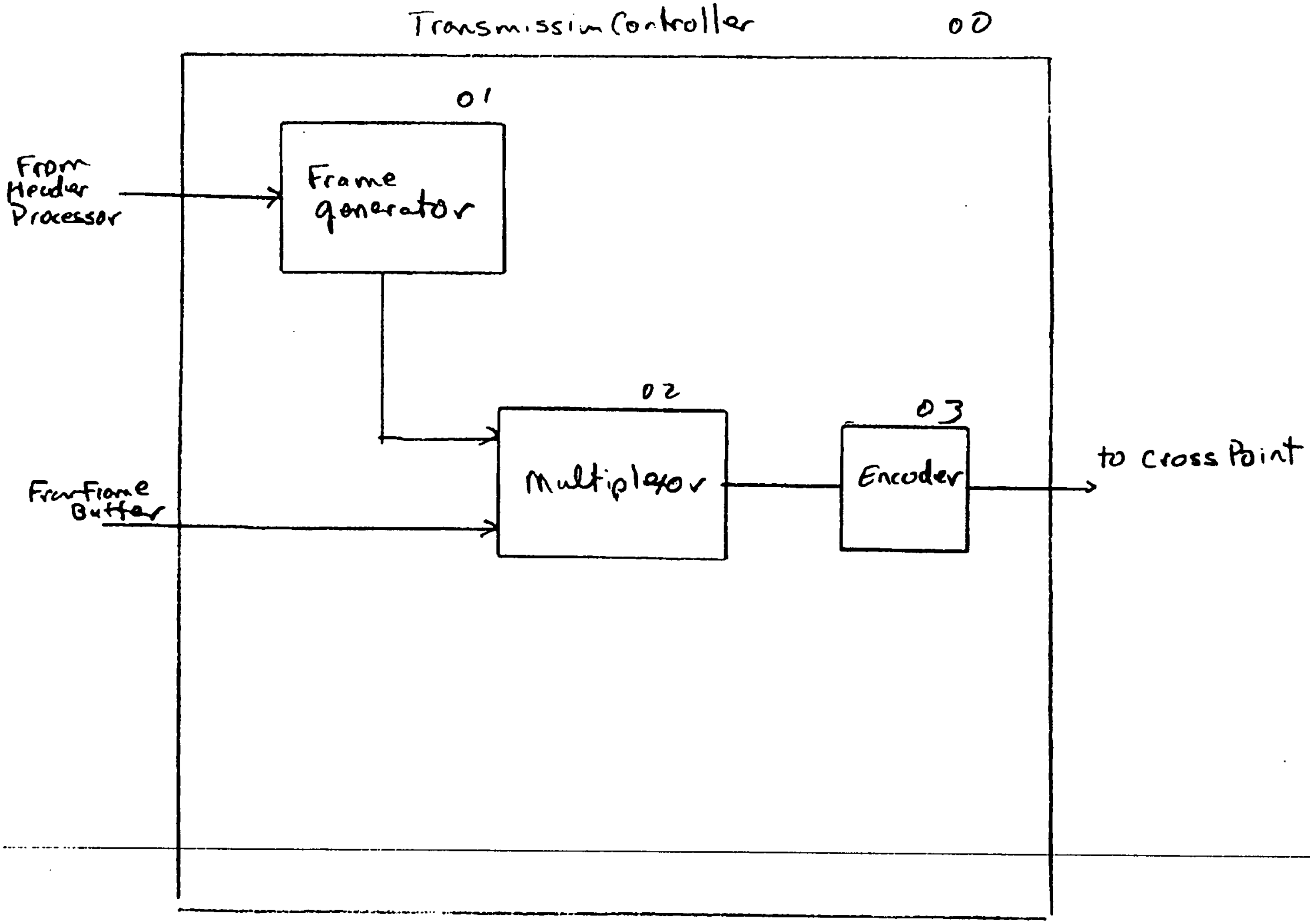

Transmissin Controller
00
01
Frame generator
From Header Processor
02
Multiplexor
Frm Frame Buffer
03
Encoder
to Cross Point

Fig 6

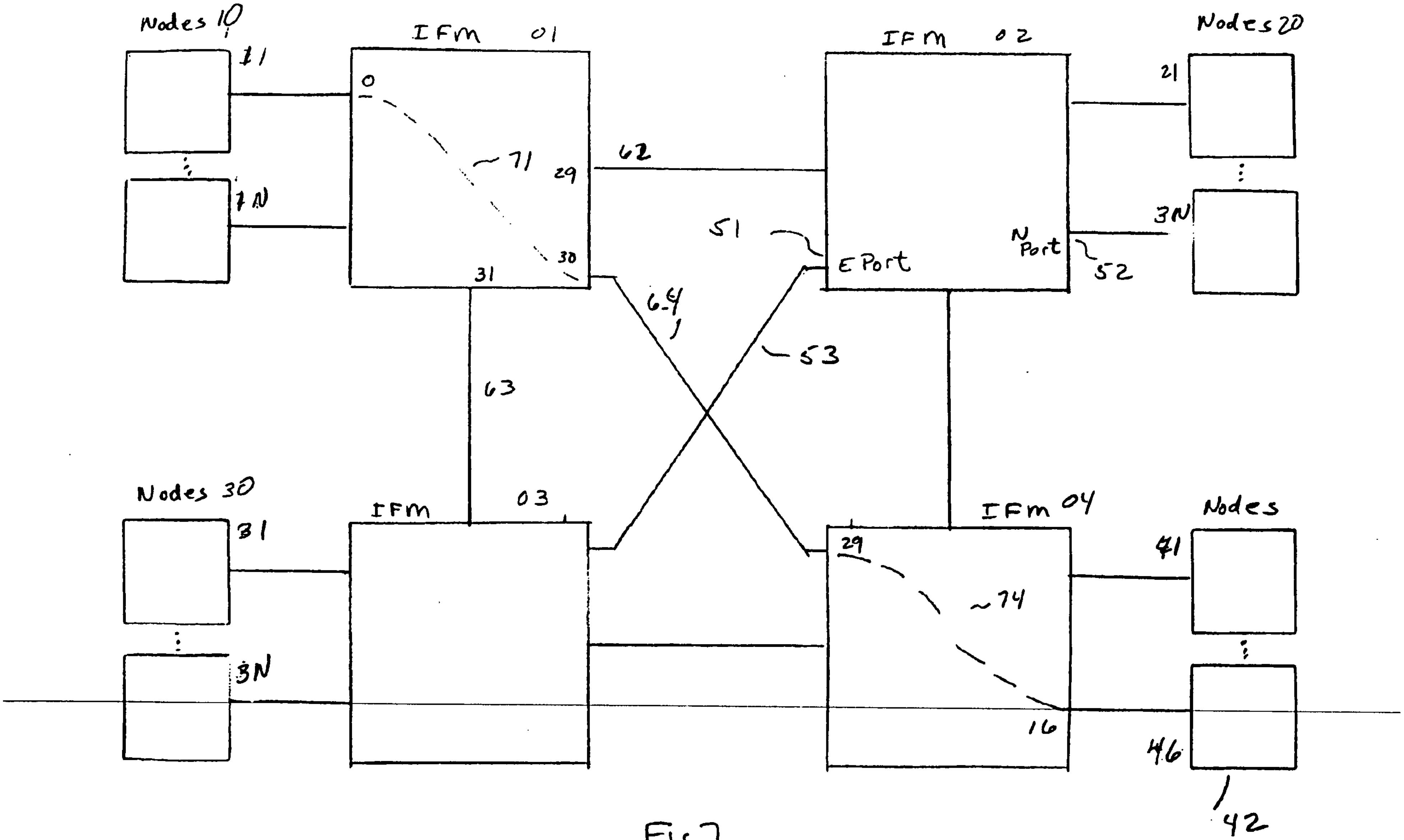

Nodes 10
11
1N
IFM 01
0
~71
29
30
31
62
63
64
IFM 02
51
E Port
N Port
52
53
Nodes 20
21
3N
Nodes 30
31
3N
IFM 03
IFM 04
29
~74
16
Nodes
41
46
42

Fig 7

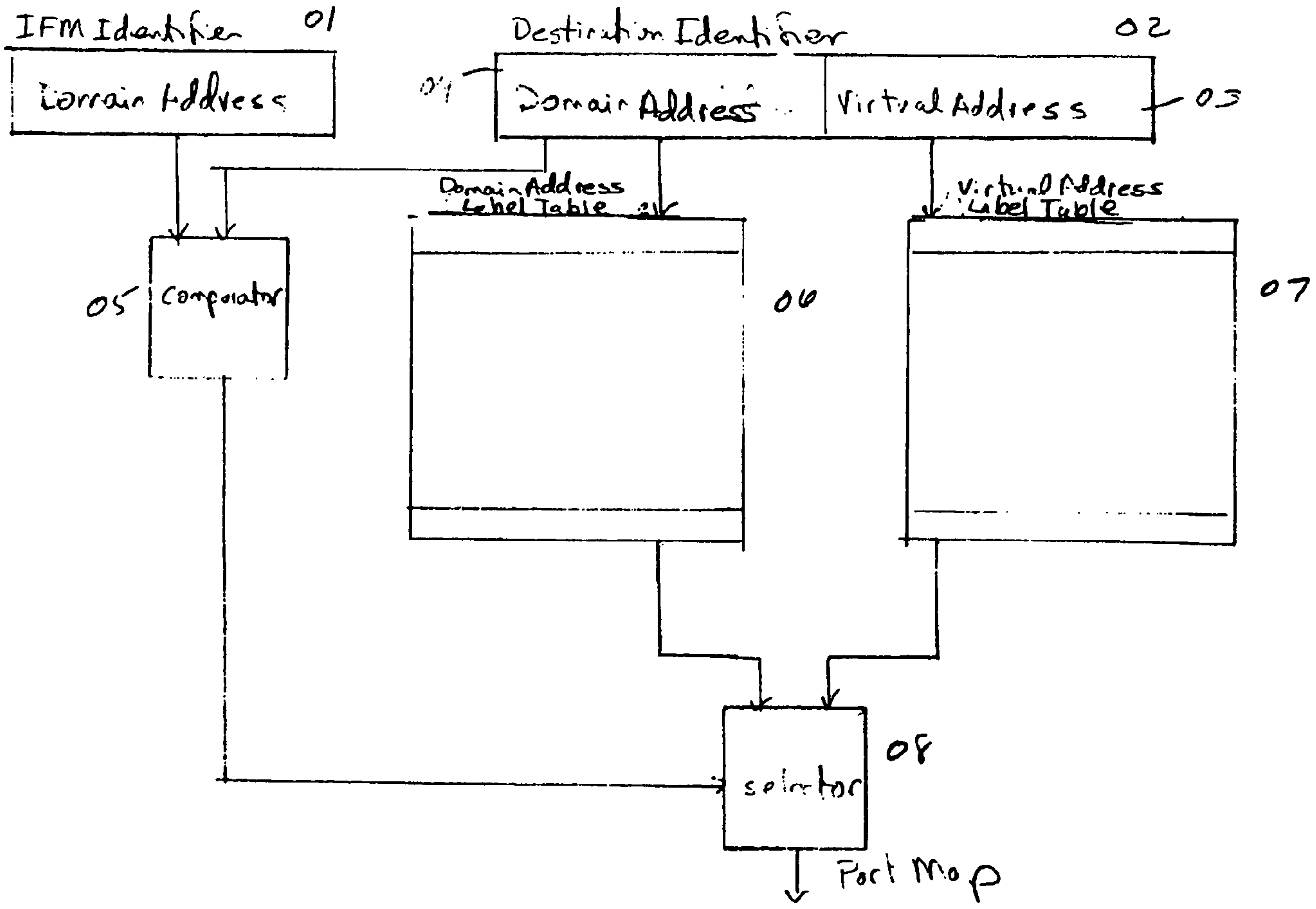

IFM Identifier
01
Domain Address
Destination Identifier
02
04
Domain Address
Virtual Address
03
Domain Address Label Table
Virtual Address Label Table
05
Comparator
06
07
08
selector
Port Map

Fig. 8

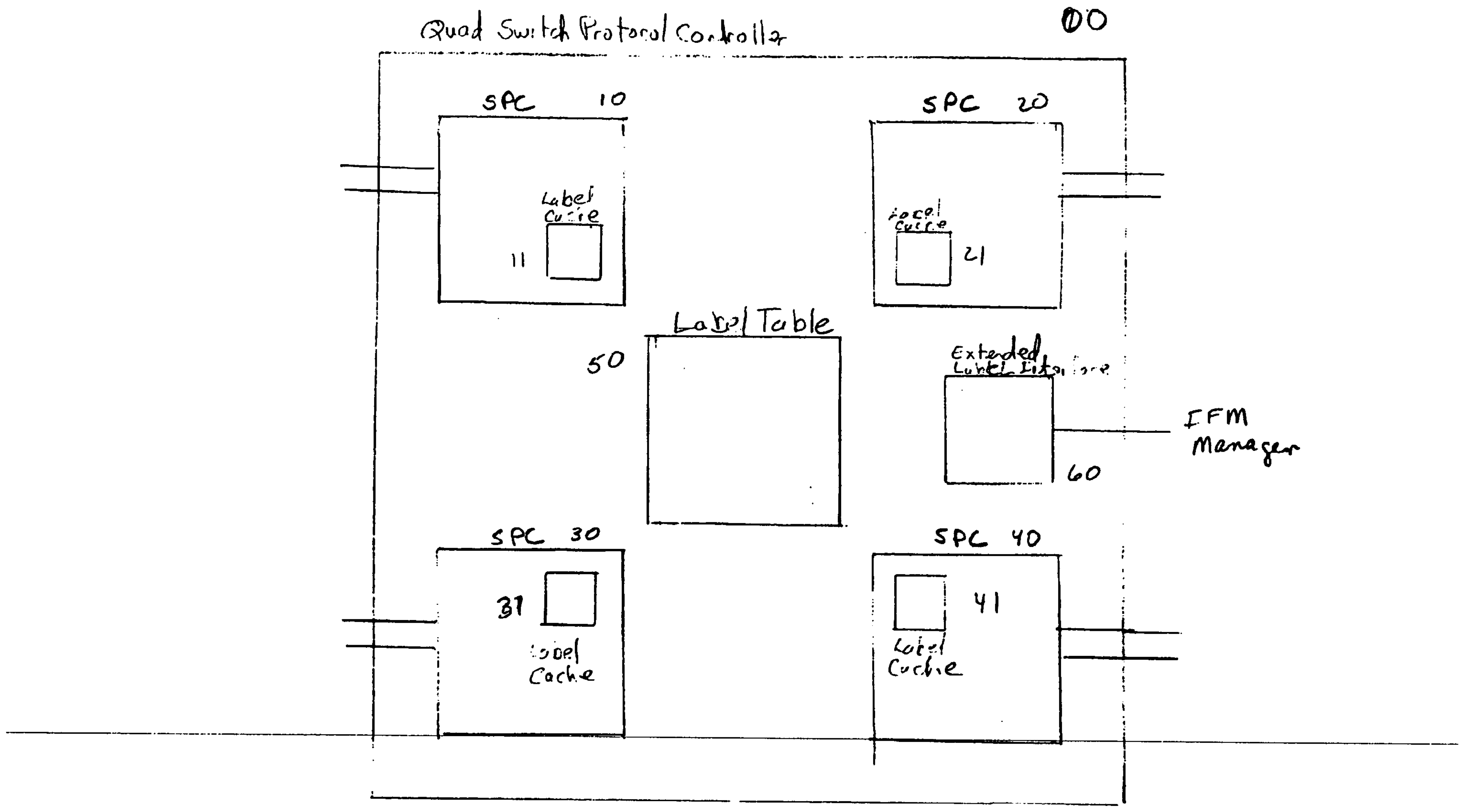

Quad Switch Protocol Controller
00
SPC 10
Label Cache
11
SPC 20
Label Cache
21
Label Table
50
Extended Label Interface
60
IFM Manager
SPC 30
31
Label Cache
SPC 40
41
Label Cache

Fig 9

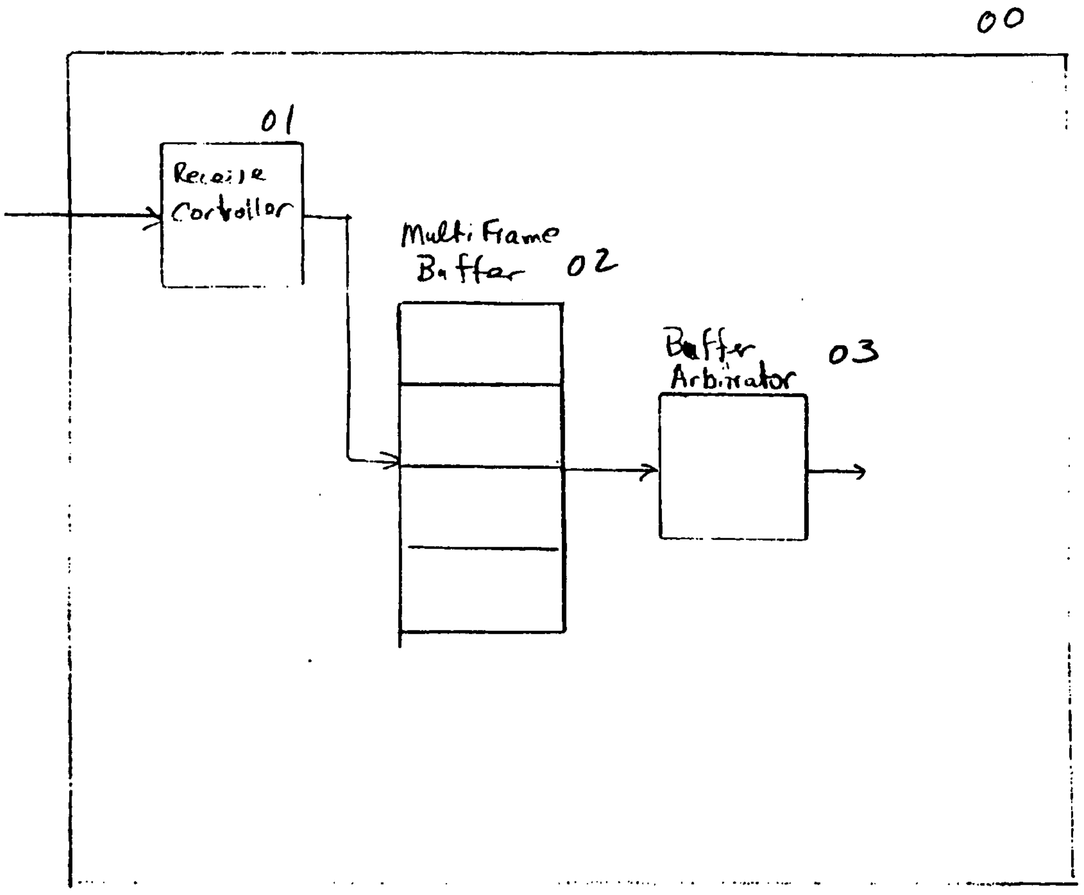

00
01
Receive Controller
Multi Frame Buffer
02
Buffer Arbitrator
03

Fig 10

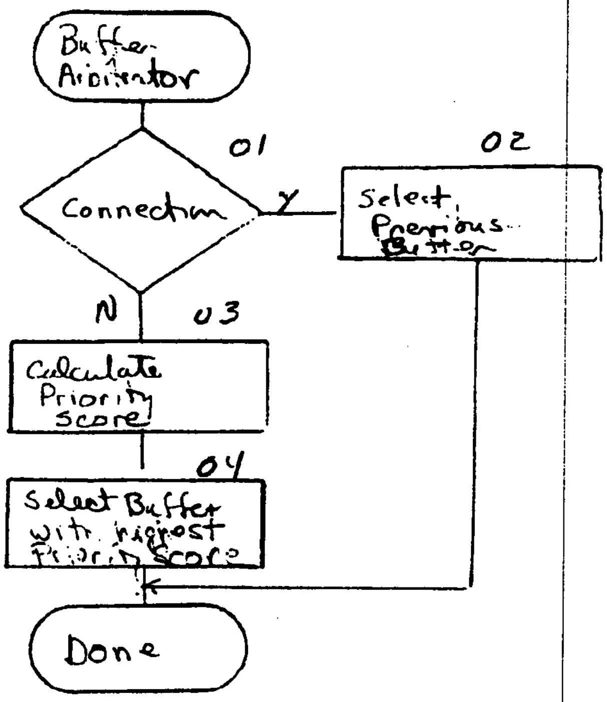

Buffer Arbitrator
01
Connection
Y
02
Select Previous Buffer
N
03
Calculate Priority Score
04
Select Buffer with highest Priority Score
Done

Fig 11

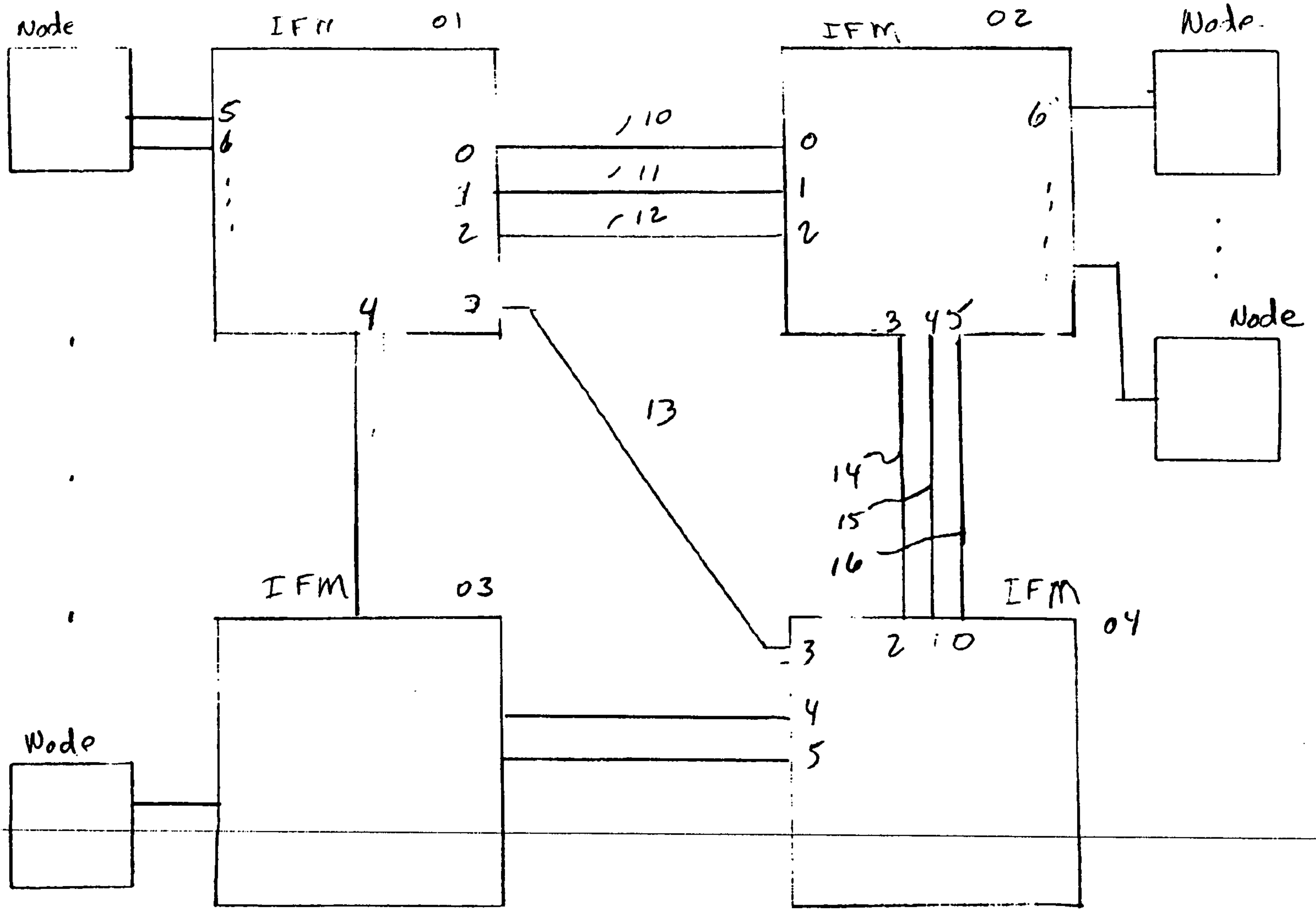

Node
IFM
IFM
Node
Node
IFM
IFM
Node

Fig 12

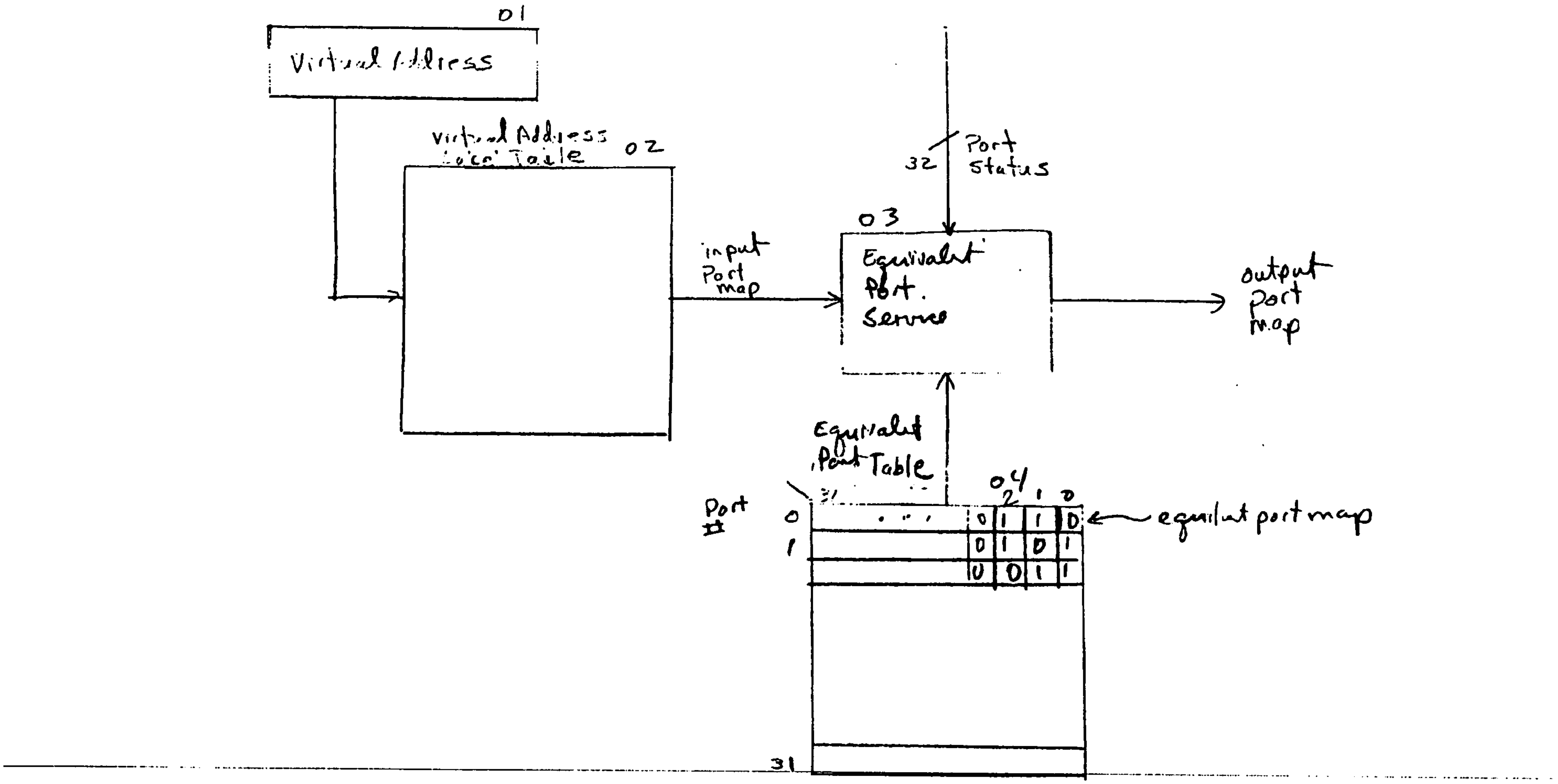

Virtual Address
Virtual Address Look Table
input Port map
Equivalent Port Service
Port status
output port map
Equivalent Port Table
. . .
Port #
equivalent port map

Fig 13

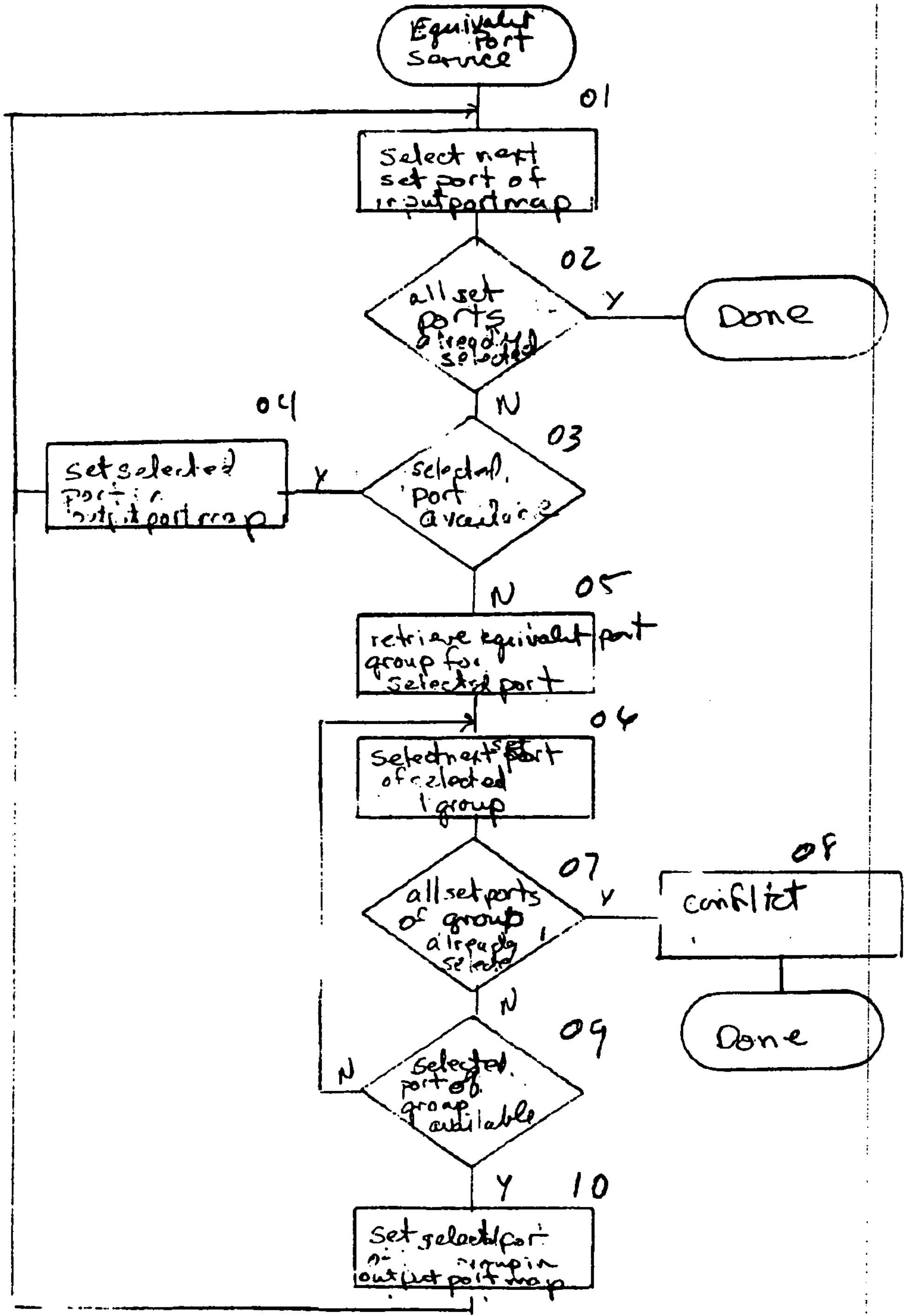

Equivalent Port Service
01
Select next set port of inputportmap
02
all set ports already selected
Y
Done
N
04
set selected port in outputportmap
Y
03
selected port available
N
05
retrieve equivalent port group for selected port
06
select next set port of selected group
07
all set ports of group already selected
Y
08
conflict
Done
N
09
N
selected port of group available
Y
10
set selected port of group in output port map

Fig 14

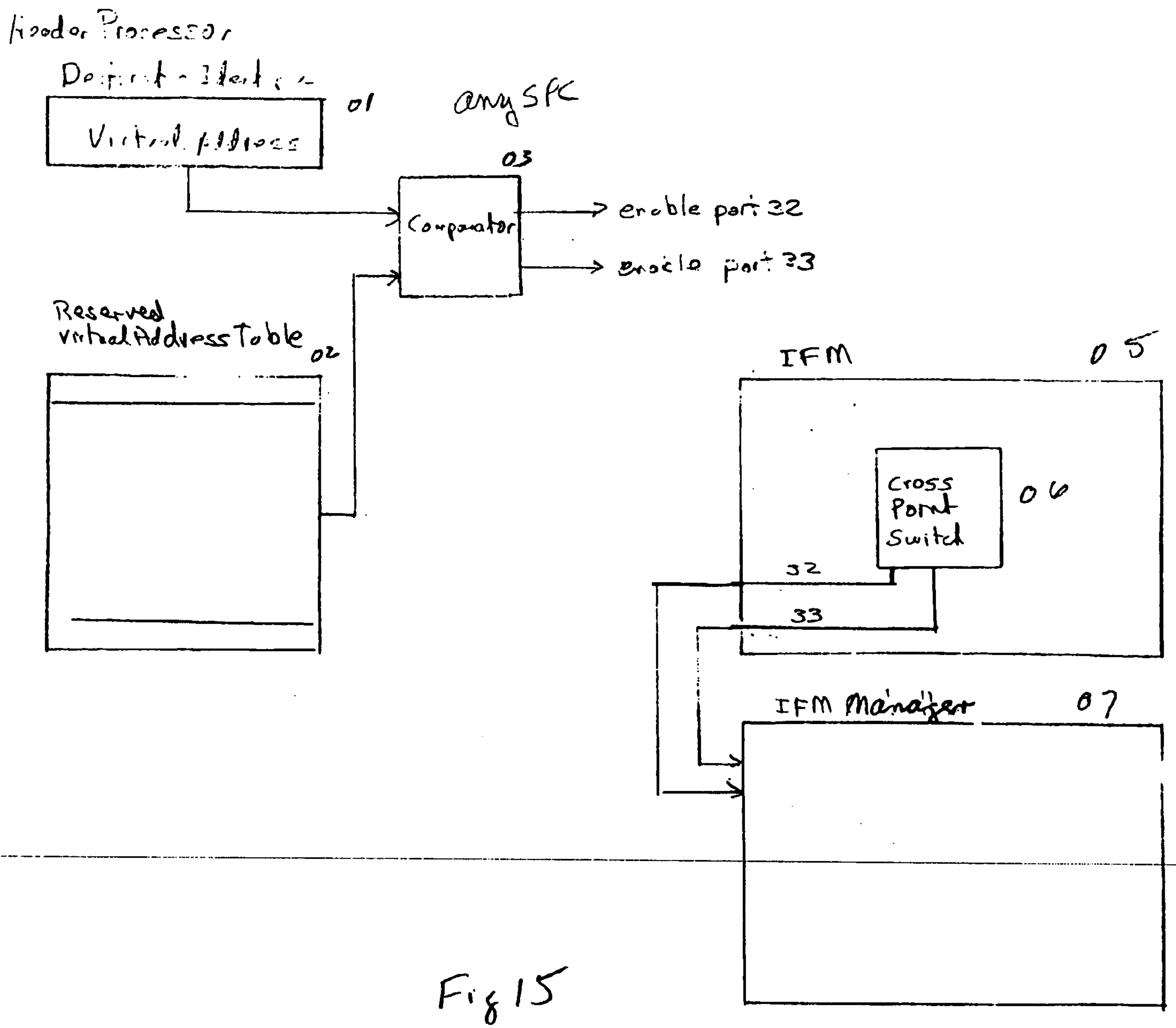
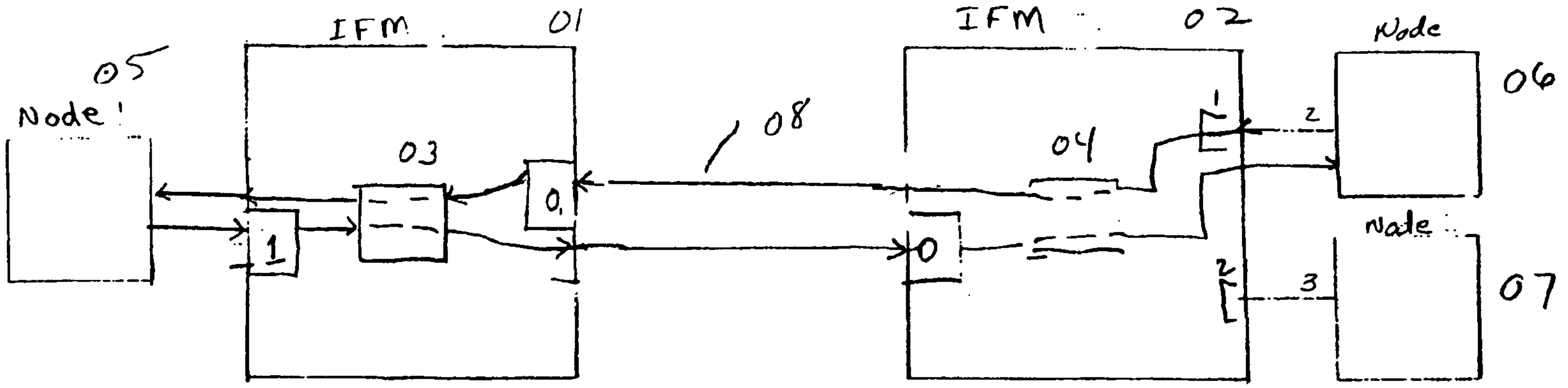

Header Processor
Destination Identifier
Virtual Address
01
any SFC
03
Comparator
enable port 32
enable port 33
Reserved virtual Address Table
02
IFM
05
Cross Point Switch
06
32
33
IFM Manager
07

Fig 15

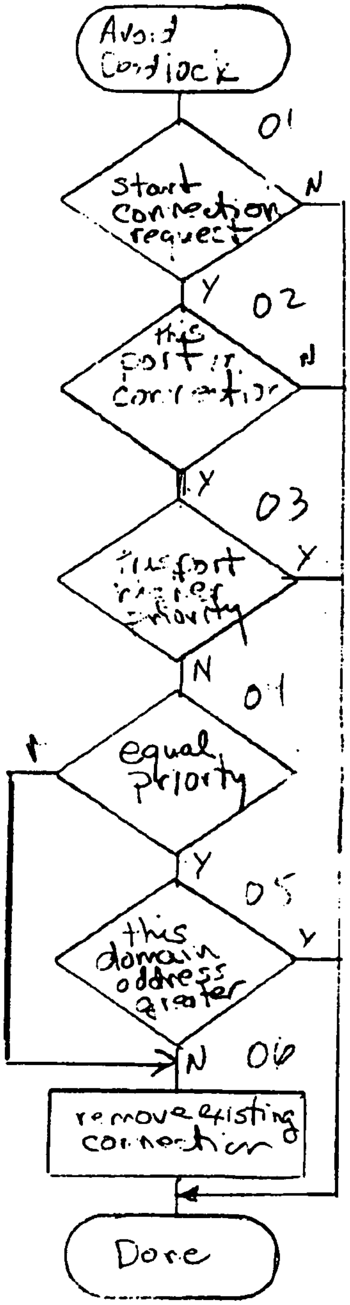

IFM
01
Node
05
03
1
0
08
IFM
02
04
0
1
2
Node
06
Node
07
2
3

Deadlock

16

Time	Node 1605	IFM 1601	Node 1606	IFM 1602
0	Send start connect		Send start connect	
1		Connect 1↔0		connect 2↔0
2		Forward start connect		Forward start connect
3		Can't forward start connect Node 1		Can't forward start connect

Fig 16


Avoid Deadlock
01
Start connection request
N
Y
02
this port in connection
N
Y
03
this port higher priority
Y
N
04
equal priority
N
Y
05
this domain address greater
Y
N
06
remove existing connection
Done

When end-to-end
not established

Fig 17

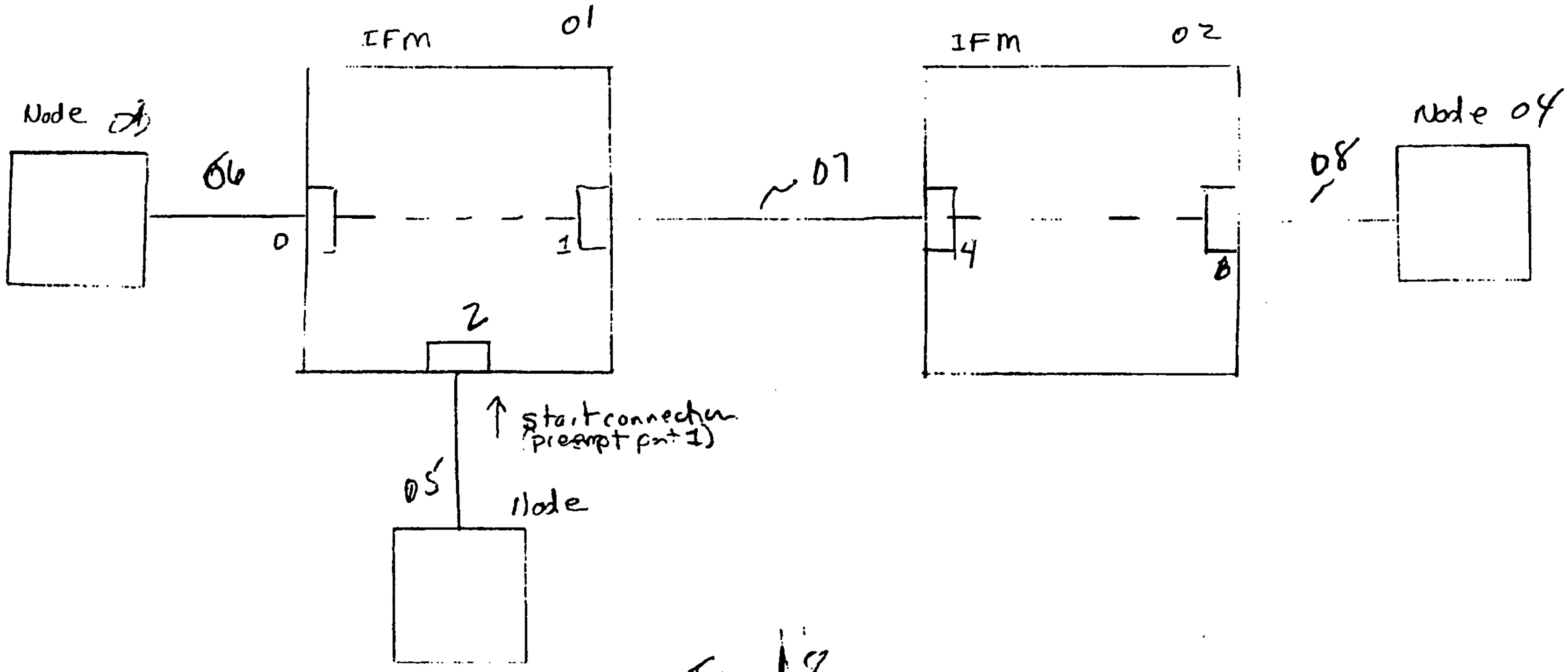
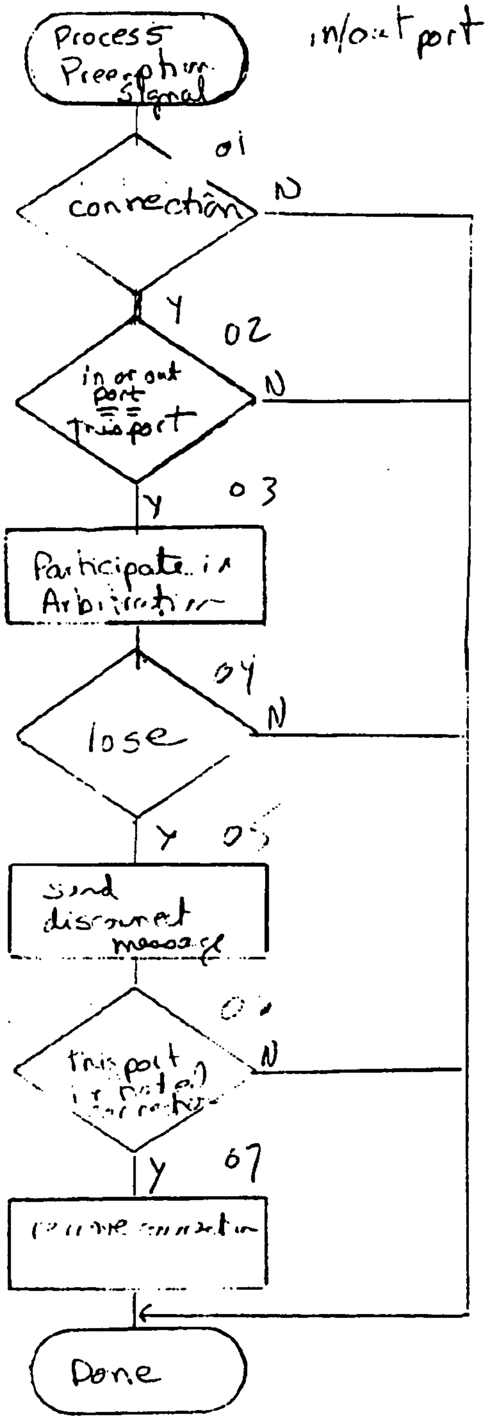

IFM
01
IFM
02
Node 03
06
0
1
2
~07
4
8
08
Node 04
↑ start connection (preempt port 1)
05
Node

Fig 18


in/out port
Process Preemption Signal
01
connection
N
Y
02
in or out port == this port
N
Y
03
Participate in Arbitration
04
lose
N
Y
05
send disconnect message
06
this port is not in connection
N
Y
07
remove connection
Done

19

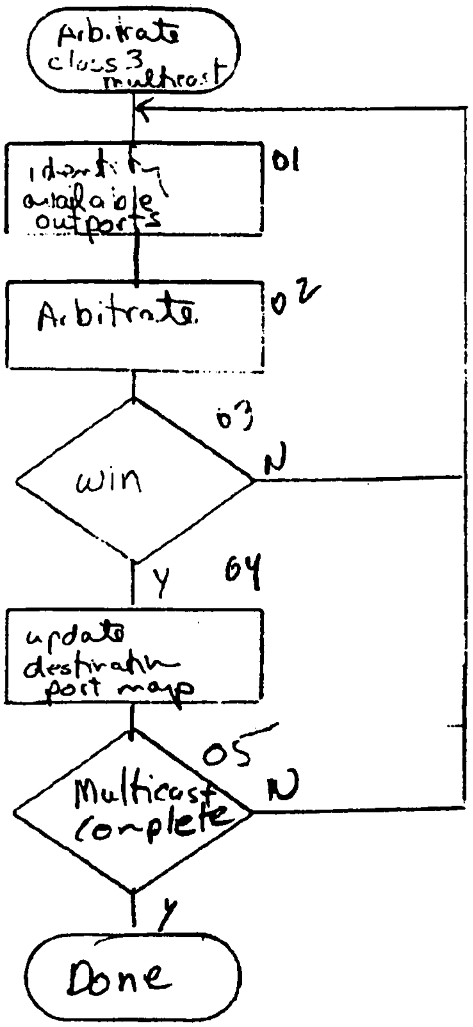

Arbitrate class 3 multicast
identify available outports
01
Arbitrate
02
03
win
N
Y
04
update destination port map
05
Multicast complete
N
Y
Done

Figure 20

DISTRIBUTED CONFERENCING SYSTEM

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is related to U.S. Patent Application No. ____________, entitled "BROADCASTING NETWORK," filed on July 31, 2000 (Attorney Docket No. 030048001 US); U.S. Patent Application No.____________, entitled "JOINING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048002 US); U.S. Patent Application No.____________, "LEAVING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048003 US); U.S. Patent Application No.____________, entitled "BROADCASTING ON A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048004 US); U.S. Patent Application No.____________, entitled "CONTACTING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048005 US); U.S. Patent Application No.____________, entitled "DISTRIBUTED AUCTION SYSTEM," filed on July 31, 2000 (Attorney Docket No. 030048006 US); U.S. Patent Application No.____________, entitled "AN INFORMATION DELIVERY SERVICE," filed on July 31, 2000 (Attorney Docket No. 030048007 US); U.S. Patent Application No.____________, entitled "DISTRIBUTED CONFERENCING SYSTEM," filed on July 31, 2000 (Attorney Docket No. 030048008 US); and U.S. Patent Application No.____________, entitled "DISTRIBUTED GAME ENVIRONMENT," filed on July 31, 2000 (Attorney Docket No. 030048009 US), the disclosures of which are incorporated herein by reference.

TECHNICAL FIELD

The described technology relates generally to a computer network and more particularly, to a broadcast channel for a subset of a computers of an underlying network.

BACKGROUND

There are a wide variety of computer network communications techniques such as point-to-point network protocols, client/server middleware, multicasting network protocols, and peer-to-peer middleware. Each of these communications techniques have their advantages and disadvantages, but none is particularly well suited to the simultaneous sharing of information among computers that are widely distributed. For example, collaborative processing applications, such as a network meeting programs, have a need to distribute information in a timely manner to all participants who may be geographically distributed.

The point-to-point network protocols, such as UNIX pipes, TCP/IP, and UDP, allow processes on different computers to communicate via point-to-point connections. The interconnection of all participants using point-to-point connections, while theoretically possible, does not scale well as a number of participants grows. For example, each participating process would need to manage its direct connections to all other participating processes. Programmers, however, find it very difficult to manage single connections, and management of multiple connections is much more complex. In addition, participating processes may be limited to the number of direct connections that they can support. This limits the number of possible participants in the sharing of information.

The client/server middleware systems provide a server that coordinates the communications between the various clients who are sharing the information. The server functions as a central authority for controlling access to shared resources. Examples of client/server middleware systems include remote procedure calls ("RPC"), database servers, and the common object request broker architecture ("CORBA"). Client/server middleware systems are not particularly well suited to sharing of information among many participants. In particular, when a client stores information to be shared at the server, each other client would need to poll the server to determine that new information is being shared. Such polling places a very high overhead on the communications network. Alternatively, each client may register a callback with the server, which the server then invokes when new information is available to be shared. Such a callback technique presents a performance bottleneck because a single server needs to call back to each client whenever new information is to be shared. In addition, the reliability of the entire sharing of information depends upon the reliability of the single server. Thus, a failure at a single computer (*i.e.*, the server) would prevent communications between any of the clients.

The multicasting network protocols allow the sending of broadcast messages to multiple recipients of a network. The current implementations of such multicasting network protocols tend to place an unacceptable overhead on the underlying network. For example, UDP multicasting would swamp the Internet when trying to locate all possible participants. IP multicasting has other problems that include needing special-purpose infrastructure (*e.g.*, routers) to support the sharing of information efficiently.

The peer-to-peer middleware communications systems rely on a multicasting network protocol or a graph of point-to-point network protocols. Such peer-to-peer middleware is provided by the T.120 Internet standard, which is used in such products as Data Connection's D.C.-share and Microsoft's NetMeeting. These peer-to-peer middleware systems rely upon a user to assemble a point-to-point graph of the connections used for sharing the information. Thus, it is neither suitable nor desirable to use peer-to-peer middleware systems when more than a small number of participants is desired. In addition, the underlying architecture of the T.120 Internet standard is a tree structure, which relies on the root node of the tree for reliability of the entire network. That is, each message must pass through the root node in order to be received by all participants.

It would be desirable to have a reliable communications network that is suitable for the simultaneous sharing of information among a large number of the processes that are widely distributed.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a graph that is 4-regular and 4-connected which represents a broadcast channel.

Figure 2 illustrates a graph representing 20 computers connected to a broadcast channel.

Figures 3A and 3B illustrate the process of connecting a new computer Z to the broadcast channel.

Figure 4A illustrates the broadcast channel of Figure 1 with an added computer.

Figure 4B illustrates the broadcast channel of Figure 4A with an added computer.

Figure 4C also illustrates the broadcast channel of Figure 4A with an added computer.

Figure 5A illustrates the disconnecting of a computer from the broadcast channel in a planned manner.

Figure 5B illustrates the disconnecting of a computer from the broadcast channel in an unplanned manner.

Figure 5C illustrates the neighbors with empty ports condition.

Figure 5D illustrates two computers that are not neighbors who now have empty ports.

Figure 5E illustrates the neighbors with empty ports condition in the small regime.

Figure 5F illustrates the situation of Figure 5E when in the large regime.

Figure 6 is a block diagram illustrating components of a computer that is connected to a broadcast channel.

Figure 7 is a block diagram illustrating the sub-components of the broadcaster component in one embodiment.

Figure 8 is a flow diagram illustrating the processing of the connect routine in one embodiment.

Figure 9 is a flow diagram illustrating the processing of the seek portal computer routine in one embodiment.

Figure 10 is a flow diagram illustrating the processing of the contact process routine in one embodiment.

Figure 11 is a flow diagram illustrating the processing of the connect request routine in one embodiment.

Figure 12 is a flow diagram of the processing of the check for external call routine in one embodiment.

Figure 13 is a flow diagram of the processing of the achieve connection routine in one embodiment.

Figure 14 is a flow diagram illustrating the processing of the external dispatcher routine in one embodiment.

Figure 15 is a flow diagram illustrating the processing of the handle seeking connection call routine in one embodiment.

Figure 16 is a flow diagram illustrating processing of the handle connection request call routine in one embodiment.

Figure 17 is a flow diagram illustrating the processing of the add neighbor routine in one embodiment.

Figure 18 is a flow diagram illustrating the processing of the forward connection edge search routine in one embodiment.

Figure 19 is a flow diagram illustrating the processing of the handle edge proposal call routine.

Figure 20 is a flow diagram illustrating the processing of the handle port connection call routine in one embodiment.

Figure 21 is a flow diagram illustrating the processing of the fill hole routine in one embodiment.

Figure 22 is a flow diagram illustrating the processing of the internal dispatcher routine in one embodiment.

Figure 23 is a flow diagram illustrating the processing of the handle broadcast message routine in one embodiment.

Figure 24 is a flow diagram illustrating the processing of the distribute broadcast message routine in one embodiment.

Figure 26 is a flow diagram illustrating the processing of the handle connection port search statement routine in one embodiment.

Figure 27 is a flow diagram illustrating the processing of the court neighbor routine in one embodiment.

Figure 28 is a flow diagram illustrating the processing of the handle connection edge search call routine in one embodiment.

Figure 29 is a flow diagram illustrating the processing of the handle connection edge search response routine in one embodiment.

Figure 30 is a flow diagram illustrating the processing of the broadcast routine in one embodiment.

Figure 31 is a flow diagram illustrating the processing of the acquire message routine in one embodiment.

Figure 32 is a flow diagram illustrating processing of the handle condition check message in one embodiment.

Figure 33 is a flow diagram illustrating processing of the handle condition repair statement routine in one embodiment.

Figure 34 is a flow diagram illustrating the processing of the handle condition double check routine.

DETAILED DESCRIPTION

A broadcast technique in which a broadcast channel overlays a point-to-point communications network is provided. The broadcasting of a message over the broadcast channel is effectively a multicast to those computers of the network that are currently connected to the broadcast channel. In one embodiment, the broadcast technique provides a logical broadcast channel to which host computers through their executing processes can be connected. Each computer that is connected to the broadcast channel can broadcast messages onto and receive messages off of the broadcast channel. Each computer that is connected to the broadcast channel receives all messages that are broadcast while it is connected. The logical broadcast channel is implemented using an underlying network system (*e.g.*, the Internet) that allows each computer connected to the underlying network system to send messages to each other connected computer using each computer's address. Thus, the broadcast technique effectively provides a broadcast channel using an underlying network system that sends messages on a point-to-point basis.

The broadcast technique overlays the underlying network system with a graph of point-to-point connections (*i.e.*, edges) between host computers (*i.e.*, nodes) through which the broadcast channel is implemented. In one embodiment, each computer is connected to four other computers, referred to as neighbors. (Actually, a process executing on a computer is connected to four other processes executing on this or four other computers.) To broadcast a message, the originating computer sends the message to each of its neighbors using its point-to-point connections. Each computer that receives the message then sends the message to its three other neighbors using the point-to-point connections. In this way, the message is propagated to each computer using the underlying network to effect the broadcasting of the message to each computer over a logical broadcast channel. A graph in which each node is connected to four other nodes is referred to as a 4-regular graph. The use of a 4-regular graph means that a computer would become disconnected from the broadcast channel only if all four of the connections to its neighbors fail. The graph used by the broadcast technique also has the property that it would take a failure of four computers to divide the graph into disjoint sub-graphs, that is two separate broadcast channels. This property is referred to as being 4-connected. Thus, the graph is both 4-regular and 4-connected.

Figure 1 illustrates a graph that is 4-regular and 4-connected which represents the broadcast channel. Each of the nine nodes A-I represents a computer that is connected to the broadcast channel, and each of the edges represents an "edge" connection between two computers of the broadcast channel. The time it takes to broadcast a message to each computer on the broadcast channel depends on the speed of the connections between the computers and the number of connections between the originating computer and each other computer on the broadcast channel. The minimum number of connections that a message would need to traverse between each pair of computers is the "distance" between the computers (*i.e.*, the shortest path between the two nodes of the graph). For example, the distance between computers A and F is one because computer A is directly connected to computer F. The distance between computers A and B is two because there is no direct connection between computers A and B, but computer F is directly connected to computer B. Thus, a message originating at computer A would be sent directly to computer F, and then sent from computer F to computer B. The maximum of the distances between the computers is the "diameter" of broadcast channel. The diameter of the broadcast channel represented by Figure 1 is two. That is, a message sent by any computer would traverse no more than two connections to reach every other computer. Figure 2 illustrates a graph representing 20 computers connected to a broadcast channel. The diameter of this broadcast channel is 4. In particular, the shortest path between computers 1 and 3 contains four connections (1-12, 12-15, 15-18, and 18-3).

The broadcast technique includes (1) the connecting of computers to the broadcast channel (*i.e.*, composing the graph), (2) the broadcasting of messages over the broadcast channel (*i.e.*, broadcasting through the graph), and (3) the disconnecting of computers from the broadcast channel (*i.e.*, decomposing the graph) composing the graph.

Composing the Graph

To connect to the broadcast channel, the computer seeking the connection first locates a computer that is currently fully connected to the broadcast channel and then establishes a connection with four of the computers that are already connected to the broadcast channel. (This assumes that there are at least four computers already connected to the broadcast channel. When there are fewer than five computers connected, the broadcast channel cannot be a 4-regular graph. In such a case, the broadcast channel is considered to be in a "small regime." The broadcast technique for the small regime is described below in detail. When five or more computers are connected, the broadcast channel is considered to be in the "large regime." This description assumes that the broadcast channel is in the large regime, unless specified otherwise.) Thus, the process of connecting to the broadcast channel includes locating the broadcast channel, identifying the neighbors for the connecting computer, and then connecting to each identified neighbor. Each computer is aware of one or more "portal computers" through which that computer may locate the broadcast channel. A seeking computer locates the broadcast channel by contacting the portal computers until it finds one that is currently fully connected to the broadcast channel. The found portal computer then directs the identifying of four computers (*i.e.*, to be the seeking computer's neighbors) to which the seeking computer is to connect. Each of these four computers then cooperates with the seeking computer to effect the connecting of the seeking computer to the broadcast channel. A computer that has started the process of locating a portal computer, but does not yet have a neighbor, is in the "seeking connection state." A computer that is connected to at least one neighbor, but not yet four neighbors, is in the "partially connected state." A computer that is currently, or has been, previously connected to four neighbors is in the "fully connected state."

Since the broadcast channel is a 4-regular graph, each of the identified computers is already connected to four computers. Thus, some connections between computers need to be broken so that the seeking computer can connect to four computers. In one embodiment, the broadcast technique identifies two pairs of computers that are currently connected to each other. Each of these pairs of computers breaks the connection between them, and then each of the four computers (two from each pair) connects to the seeking computer. Figures 3A and 3B illustrate the process of a new computer Z connecting to the broadcast channel. Figure 3A illustrates the broadcast channel before computer Z is connected. The pairs of computers B and E and computers C and D are the two pairs that are identified as the neighbors for the new computer Z. The connections between each of these pairs is broken, and a connection between computer Z and each of computers B, C, D, and E is established as indicated by Figure 3B. The process of breaking the connection between two neighbors and reconnecting each of the former neighbors to another computer is referred to as "edge pinning" as the edge between two nodes may be considered to be stretched and pinned to a new node.

Each computer connected to the broadcast channel allocates five communications ports for communicating with other computers. Four of the ports are referred to as "internal" ports because they are the ports through which the messages of the broadcast channels are sent. The connections between internal ports of neighbors are referred to as "internal" connections. Thus, the internal connections of the broadcast channel form the 4-regular and 4-connected graph. The fifth port is referred to as an "external" port because it is used for sending non-broadcast messages between two computers. Neighbors can send non-broadcast messages either through their internal ports of their connection or through their external ports. A seeking computer uses external ports when locating a portal computer.

In one embodiment, the broadcast technique establishes the computer connections using the TCP/IP communications protocol, which is a point-to-point protocol, as the underlying network. The TCP/IP protocol provides for reliable and ordered delivery of messages between computers. The TCP/IP protocol provides each computer with a "port space" that is shared among all the processes that may execute on that computer. The ports are identified by numbers from 0 to 65,535. The first 2056 ports are reserved for specific applications (*e.g.*, port 80 for HTTP messages). The remainder of the ports are user ports that are available to any process. In one embodiment, a set of port numbers can be reserved for use by the computer connected to the broadcast channel. In an alternative embodiment, the port numbers used are dynamically identified by each computer. Each computer dynamically identifies an available port to be used as its call-in port. This call-in port is used to establish connections with the external port and the internal ports. Each computer that is connected to the broadcast channel can receive non-broadcast messages through its external port. A seeking computer tries "dialing" the port numbers of the portal computers until a portal computer "answers," a call on its call-in port. A portal computer answers when it is connected to or attempting to connect to the broadcast channel and its call-in port is dialed. (In this description, a telephone metaphor is used to describe the connections.) When a computer receives a call on its call-in port, it transfers the call to another port. Thus, the seeking computer actually communicates through that transfer-to port, which is the external port. The call is transferred so that other computers can place calls to that computer via the call-in port. The seeking computer then communicates via that external port to request the portal computer to assist in connecting the seeking computer to the broadcast channel. The seeking computer could identify the call-in port number of a portal computer by successively dialing each port in port number order. As discussed below in detail, the broadcast technique uses a hashing algorithm to select the port number order, which may result in improved performance.

A seeking computer could connect to the broadcast channel by connecting to computers either directly connected to the found portal computer or directly connected to one of its neighbors. A possible problem with such a scheme for identifying the neighbors for the seeking computer is that the diameter of the broadcast channel may increase when each seeking computer uses the same found portal computer and establishes a connection to the broadcast channel directly through that found portal computer. Conceptually, the graph becomes elongated in the direction of where the new nodes are added. Figures 4A-4C illustrate that possible problem. Figure 4A illustrates the broadcast channel of Figure 1 with an added computer. Computer J was connected to the broadcast channel by edge pinning edges C-D and E-H to computer J. The diameter of this broadcast channel is still two. Figure 4B illustrates the broadcast channel of Figure 4A with an added computer. Computer K was connected to the broadcast channel by edge pinning edges E-J and B-C to computer K. The diameter of this broadcast channel is three, because the shortest path from computer G to computer K is through edges G-A, A-E, and E-K. Figure 4C also illustrates the broadcast channel of Figure 4A with an added computer. Computer K was connected to the broadcast channel by edge pinning edges D-G and E-J to computer K. The diameter of this broadcast channel is, however, still two. Thus, the selection of neighbors impacts the diameter of the broadcast channel. To help minimize the diameter, the broadcast technique uses a random selection technique to identify the four neighbors of a computer in the seeking connection state. The random selection technique tends to distribute the connections to new seeking computers throughout the computers of the broadcast channel which may result in smaller overall diameters.

Broadcasting Through the Graph

As described above, each computer that is connected to the broadcast channel can broadcast messages onto the broadcast channel and does receive all messages that are broadcast on the broadcast channel. The computer that originates a message to be broadcast sends that message to each of its four neighbors using the internal connections. When a computer receives a broadcast message from a neighbor, it sends the message to its three other neighbors. Each computer on the broadcast channel, except the originating computer, will thus receive a copy of each broadcast message from each of its four neighbors. Each computer, however, only sends the first copy of the message that it receives to its neighbors and disregards subsequently received copies. Thus, the total number of copies of a message that is sent between the computers is 3N+1, where N is the number of computers connected to the broadcast channel. Each computer sends three copies of the message, except for the originating computer, which sends four copies of the message.

The redundancy of the message sending helps to ensure the overall reliability of the broadcast channel. Since each computer has four connections to the broadcast channel, if one computer fails during the broadcast of a message, its neighbors have three other connections through which they will receive copies of the broadcast message. Also, if the internal connection between two computers is slow, each computer has three other connections through which it may receive a copy of each message sooner.

Each computer that originates a message numbers its own messages sequentially. Because of the dynamic nature of the broadcast channel and because there are many possible connection paths between computers, the messages may be received out of order. For example, the distance between an originating computer and a certain receiving computer may be four. After sending the first message, the originating computer and receiving computer may become neighbors and thus the distance between them changes to one. The first message may have to travel a distance of four to reach the receiving computer. The second message only has to travel a distance of one. Thus, it is possible for the second message to reach the receiving computer before the first message.

When the broadcast channel is in a steady state (*i.e.*, no computers connecting or disconnecting from the broadcast channel), out-of-order messages are not a problem because each computer will eventually receive both messages and can queue messages until all earlier ordered messages are received. If, however, the broadcast channel is not in a steady state, then problems can occur. In particular, a computer may connect to the broadcast channel after the second message has already been received and forwarded on by its new neighbors. When a new neighbor eventually receives the first message, it sends the message to the newly connected computer. Thus, the newly connected computer will receive the first message, but will not receive the second message. If the newly connected computer needs to process the messages in order, it would wait indefinitely for the second message.

One solution to this problem is to have each computer queue all the messages that it receives until it can send them in their proper order to its neighbors. This solution, however, may tend to slow down the propagation of messages through the computers of the broadcast channel. Another solution that may have less impact on the propagation speed is to queue messages only at computers who are neighbors of the newly connected computers. Each already connected neighbor would forward messages as it receives them to its other neighbors who are not newly connected, but not to the newly connected neighbor. The already connected neighbor would only forward messages from each originating computer to the newly connected computer when it can ensure that no gaps in the messages from that originating computer will occur. In one embodiment, the already connected neighbor may track the highest sequence number of the messages already received and forwarded on from each originating computer. The already connected computer will send only higher numbered messages from the originating computers to the newly connected computer. Once all lower numbered messages have been received from all originating computers, then the already connected computer can treat the newly connected computer as its other neighbors and simply forward each message as it is received. In another embodiment, each computer may queue messages and only forwards to the newly connected computer those messages as the gaps are filled in. For example, a computer might receive messages 4 and 5 and then receive message 3. In such a case, the already connected computer would forward queue messages 4 and 5. When message 3 is finally received, the already connected computer will send messages 3, 4, and 5 to the newly connected computer. If messages 4 and 5 were sent to the newly connected computer before message 3, then the newly connected computer would process messages 4 and 5 and disregard message 3. Because the already connected computer queues messages 4 and 5, the newly connected computer will be able to process message 3. It is possible that a newly connected computer will receive a set of messages from an originating computer through one neighbor and then receive another set of message from the same originating computer through another neighbor. If the second set of messages contains a message that is ordered earlier than the messages of the first set received, then the newly connected computer may ignore that earlier ordered message if the computer already processed those later ordered messages.

Decomposing the Graph

A connected computer disconnects from the broadcast channel either in a planned or unplanned manner. When a computer disconnects in a planned manner, it sends a disconnect message to each of its four neighbors. The disconnect message includes a list that identifies the four neighbors of the disconnecting computer. When a neighbor receives the disconnect message, it tries to connect to one of the computers on the list. In one embodiment, the first computer in the list will try to connect to the second computer in the list, and the third computer in the list will try to connect to the fourth computer in the list. If a computer cannot connect (*e.g.*, the first and second computers are already connected), then the computers may try connecting in various other combinations. If connections cannot be established, each computer broadcasts a message that it needs to establish a connection with another computer. When a computer with an available internal port receives the message, it can then establish a connection with the computer that broadcast the message. Figures 5A-5D illustrate the disconnecting of a computer from the broadcast channel. Figure 5A illustrates the disconnecting of a computer from the broadcast channel in a planned manner. When computer H decides to disconnect, it sends its list of neighbors to each of its neighbors (computers A, E, F and I) and then disconnects from each of its neighbors. When computers A and I receive the message they establish a connection between them as indicated by the dashed line, and similarly for computers E and F.

When a computer disconnects in an unplanned manner, such as resulting from a power failure, the neighbors connected to the disconnected computer recognize the disconnection when each attempts to send its next message to the now disconnected computer. Each former neighbor of the disconnected computer recognizes that it is short one connection (*i.e.*, it has a hole or empty port). When a connected computer detects that one of its neighbors is now disconnected, it broadcasts a port connection request on the broadcast channel, which indicates that it has one internal port that needs a connection. The port connection request identifies the call-in port of the requesting computer. When a connected computer that is also short a connection receives the connection request, it communicates with the requesting computer through its external port to establish a connection between the two computers. Figure 5B illustrates the disconnecting of a computer from the broadcast channel in an unplanned manner. In this illustration, computer H has disconnected in an unplanned manner. When each of its neighbors, computers A, E, F, and I, recognizes the disconnection, each neighbor broadcasts a port connection request indicating that it needs to fill an empty port. As shown by the dashed lines, computers F and I and computers A and E respond to each other's requests and establish a connection.

It is possible that a planned or unplanned disconnection may result in two neighbors each having an empty internal port. In such a case, since they are neighbors, they are already connected and cannot fill their empty ports by connecting to each other. Such a condition is referred to as the "neighbors with empty ports" condition. Each neighbor broadcasts a port connection request when it detects that it has an empty port as described above. When a neighbor receives the port connection request from the other neighbor, it will recognize the condition that its neighbor also has an empty port. Such a condition may also occur when the broadcast channel is in the small regime. The condition can only be corrected when in the large regime. When in the small regime, each computer will have less than four neighbors. To detect this condition in the large regime, which would be a problem if not repaired, the first neighbor to receive the port connection request recognizes the condition and sends a condition check message to the other neighbor. The condition check message includes a list of the neighbors of the sending computer. When the receiving computer receives the list, it compares the list to its own list of neighbors. If the lists are different, then this condition has occurred in the large regime and repair is needed. To repair this condition, the receiving computer will send a condition repair request to one of the neighbors of the sending computer which is not already a neighbor of the receiving computer. When the computer receives the condition repair request, it disconnects from one of its neighbors (other than the neighbor that is involved with the condition) and connects to the computer that sent the condition repair request. Thus, one of the original neighbors involved in the condition will have had a port filled. However, two computers are still in need of a connection, the other original neighbor and the computer that is now disconnected from the computer that received the condition repair request. Those two computers send out port connection requests. If those two computers are not neighbors, then they will connect to each other when they receive the requests. If, however, the two computers are neighbors, then they repeat the condition repair process until two non-neighbors are in need of connections.

It is possible that the two original neighbors with the condition may have the same set of neighbors. When the neighbor that receives the condition check message determines that the sets of neighbors are the same, it sends a condition double check message to one of its neighbors other than the neighbor who also has the condition. When the computer receives the condition double check message, it determines whether it has the same set of neighbors as the sending computer. If so, the broadcast channel is in the small regime and the condition is not a problem. If the set of neighbors are different, then the computer that received the condition double check message sends a condition check message to the original neighbors with the condition. The computer that receives that condition check message directs one of it neighbors to connect to one of the original neighbors with the condition by sending a condition repair message. Thus, one of the original neighbors with the condition will have its port filled.

Figure 5C illustrates the neighbors with empty ports condition. In this illustration, computer H disconnected in an unplanned manner, but computers F and I responded to the port connection request of the other and are now connected together. The other former neighbors of computer H, computers A and E, are already neighbors, which gives rise to the neighbors with empty ports condition. In this example, computer E received the port connection request from computer A, recognized the possible condition, and sent (since they are neighbors via the internal connection) a condition check message with a list of its neighbors to computer A. When computer A received the list, it recognized that computer E has a different set of neighbor (*i.e.*, the broadcast channel is in the large regime). Computer A selected computer D, which is a neighbor of computer E and sent it a condition repair request. When computer D received the condition repair request, it disconnected from one of its neighbors (other than computer E), which is computer G in this example. Computer D then connected to computer A. Figure 5D illustrates two computers that are not neighbors who now have empty ports. Computers E and G now have empty ports and are not currently neighbors. Therefore, computers E and G can connect to each other.

Figures 5E and 5F further illustrate the neighbors with empty ports condition. Figure 5E illustrates the neighbors with empty ports condition in the small regime. In this example, if computer E disconnected in an unplanned manner, then each computer broadcasts a port connection request when it detects the disconnect. When computer A receives the port connection request form computer B, it detects the neighbors with empty ports condition and sends a condition check message to computer B. Computer B recognizes that it has the same set of neighbors (computer C and D) as computer A and then sends a condition double check message to computer C. Computer C recognizes that the broadcast channel is in the small regime because is also has the same set of neighbors as computers A and B, computer C may then broadcast a message indicating that the broadcast channel is in the small regime.

Figure 5F illustrates the situation of Figure 5E when in the large regime. As discussed above, computer C receives the condition double check message from computer B. In this case, computer C recognizes that the broadcast channel is in the large regime because it has a set of neighbors that is different from computer B. The edges extending up from computer C and D indicate connections to other computers. Computer C then sends a condition check message to computer B. When computer B receives the condition check message, it sends a condition repair message to one of the neighbors of computer C. The computer that receives the condition repair message disconnects from one of its neighbors, other than computer C, and tries to connect to computer B and the neighbor from which it disconnected tries to connect to computer A.

Port Selection

As described above, the TCP/IP protocol designates ports above number 2056 as user ports. The broadcast technique uses five user port numbers on each computer: one external port and four internal ports. Generally, user ports cannot be statically allocated to an application program because other applications programs executing on the same computer may use conflicting port numbers. As a result, in one embodiment, the computers connected to the broadcast channel dynamically allocate their port numbers. Each computer could simply try to locate the lowest number unused port on that computer and use that port as the call-in port. A seeking computer, however, does not know in advance the call-in port number of the portal computers when the port numbers are dynamically allocated. Thus, a seeking computer needs to dial ports of a portal computer starting with the lowest port number when locating the call-in port of a portal computer. If the portal computer is connected to (or attempting to connect to) the broadcast channel, then the seeking computer would eventually find the call-in port. If the portal computer is not connected, then the seeking computer would eventually dial every user port. In addition, if each application program on a computer tried to allocate low-ordered port numbers, then a portal computer may end up with a high-numbered port for its call-in port because many of the low-ordered port numbers would be used by other application programs. Since the dialing of a port is a relatively slow process, it would take the seeking computer a long time to locate the call-in port of a portal computer. To minimize this time, the broadcast technique uses a port ordering algorithm to identify the port number order that a portal computer should use when finding an available port for its call-in port. In one embodiment, the broadcast technique uses a hashing algorithm to identify the port order. The algorithm preferably distributes the ordering of the port numbers randomly through out the user port number space and only selects each port number once. In addition, every time the algorithm is executed on any computer for a given channel type and channel instance, it generates the same port ordering. As described below, it is possible for a computer to be connected to multiple broadcast channels that are uniquely identified by channel type and channel instance. The algorithm may be "seeded" with channel type and channel instance in order to generate a unique ordering of port numbers for each broadcast channel. Thus, a seeking computer will dial the ports of a portal computer in the same order as the portal computer used when allocating its call-in port.

If many computers are at the same time seeking connection to a broadcast channel through a single portal computer, then the ports of the portal computer may be busy when called by seeking computers. The seeking computers would typically need to keep on redialing a busy port. The process of locating a call-in port may be significantly slowed by such redialing. In one embodiment, each seeking computer may each reorder the first few port numbers generated by the hashing algorithm. For example, each seeking computer could randomly reorder the first eight port numbers generated by the hashing algorithm. The random ordering could also be weighted where the first port number generated by the hashing algorithm would have a 50% chance of being first in the reordering, the second port number would have a 25% chance of being first in the reordering, and so on. Because the seeking computers would use different orderings, the likelihood of finding a busy port is reduced. For example, if the first eight port numbers are randomly selected, then it is possible that eight seeking computers could be simultaneously dialing ports in different sequences which would reduce the chances of dialing a busy port.

Locating a Portal Computer

Each computer that can connect to the broadcast channel has a list of one or more portal computers through which it can connect to the broadcast channel. In one embodiment, each computer has the same set of portal computers. A seeking computer locates a portal computer that is connected to the broadcast channel by successively dialing the ports of each portal computer in the order specified by an algorithm. A seeking computer could select the first portal computer and then dial all its ports until a call-in port of a computer that is fully connected to the broadcast channel is found. If no call-in port is found, then the seeking computer would select the next portal computer and repeat the process until a portal computer with such a call-in port is found. A problem with such a seeking technique is that all user ports of each portal computer are dialed until a portal computer fully connected to the broadcast channel is found. In an alternate embodiment, the seeking computer selects a port number according to the algorithm and then dials each portal computer at that port number. If no acceptable call-in port to the broadcast channel is found, then the seeking computer selects the next port number and repeats the process. Since the call-in ports are likely allocated at lower-ordered port numbers, the seeking computer first dials the port numbers that are most likely to be call-in ports of the broadcast channel. The seeking computers may have a maximum search depth, that is the number of ports that it will dial when seeking a portal computer that is fully connected. If the seeking computer exhausts its search depth, then either the broadcast channel has not yet been established or, if the seeking computer is also a portal computer, it can then establish the broadcast channel with itself as the first fully connected computer.

When a seeking computer locates a portal computer that is itself not fully connected, the two computers do not connect when they first locate each other because the broadcast channel may already be established and accessible through a higher-ordered port number on another portal computer. If the two seeking computers were to connect to each other, then two disjoint broadcast channels would be formed. Each seeking computer can share its experience in trying to locate a portal computer with the other seeking computer. In particular, if one seeking computer has searched all the portal computers to a depth of eight, then the one seeking computer can share that it has searched to a depth of eight with another seeking computer. If that other seeking computer has searched to a depth of, for example, only four, it can skip searching through depths five through eight and that other seeking computer can advance its searching to a depth of nine.

In one embodiment, each computer may have a different set of portal computers and a different maximum search depth. In such a situation, it may be possible that two disjoint broadcast channels are formed because a seeking computer cannot locate a fully connected port computer at a higher depth. Similarly, if the set of portal computers are disjoint, then two separate broadcast channels would be formed.

Identifying Neighbors for a Seeking Computer

As described above, the neighbors of a newly connecting computer are preferably selected randomly from the set of currently connected computers. One advantage of the broadcast channel, however, is that no computer has global knowledge of the broadcast channel. Rather, each computer has local knowledge of itself and its neighbors. This limited local knowledge has the advantage that all the connected computers are peers (as far as the broadcasting is concerned) and the failure of any one computer (actually any three computers when in the 4-regular and 4-connect form) will not cause the broadcast channel to fail. This local knowledge makes it difficult for a portal computer to randomly select four neighbors for a seeking computer.

To select the four computers, a portal computer sends an edge connection request message through one of its internal connections that is randomly selected. The receiving computer again sends the edge connection request message through one of its internal connections that is randomly selected. This sending of the message corresponds to a random walk through the graph that represents the broadcast channel. Eventually, a receiving computer will decide that the message has traveled far enough to represent a randomly selected computer. That receiving computer will offer the internal connection upon which it received the edge connection request message to the seeking computer for edge pinning. Of course, if either of the computers at the end of the offered internal connection are already neighbors of the seeking computer, then the seeking computer cannot connect through that internal connection. The computer that decided that the message has traveled far enough will detect this condition of already being a neighbor and send the message to a randomly selected neighbor.

In one embodiment, the distance that the edge connection request message travels is established by the portal computer to be approximately twice the estimated diameter of the broadcast channel. The message includes an indication of the distance that it is to travel. Each receiving computer decrements that distance to travel before sending the message on. The computer that receives a message with a distance to travel that is zero is considered to be the randomly selected computer. If that randomly selected computer cannot connect to the seeking computer (*e.g.*, because it is already connected to it), then that randomly selected computer forwards the edge connection request to one of its neighbors with a new distance to travel. In one embodiment, the forwarding computer toggles the new distance to travel between zero and one to help prevent two computers from sending the message back and forth between each other.

Because of the local nature of the information maintained by each computer connected to the broadcast channel, the computers need not generally be aware of the diameter of the broadcast channel. In one embodiment, each message sent through the broadcast channel has a distance traveled field. Each computer that forwards a message increments the distance traveled field. Each computer also maintains an estimated diameter of the broadcast channel. When a computer receives a message that has traveled a distance that indicates that the estimated diameter is too small, it updates its estimated diameter and broadcasts an estimated diameter message. When a computer receives an estimated diameter message that indicates a diameter that is larger than its own estimated diameter, it updates its own estimated diameter. This estimated diameter is used to establish the distance that an edge connection request message should travel.

External Data Representation

The computers connected to the broadcast channel may internally store their data in different formats. For example, one computer may use 32-bit integers, and another computer may use 64-bit integers. As another example, one computer may use ASCII to represent text and another computer may use Unicode. To allow communications between heterogeneous computers, the messages sent over the broadcast channel may use the XDR ("eXternal Data Representation") format.

The underlying peer-to-peer communications protocol may send multiple messages in a single message stream. The traditional technique for retrieving messages from a stream has been to repeatedly invoke an operating system routine to retrieve the next message in the stream. The retrieval of each message may require two calls to the operating system: one to retrieve the size of the next message and the other to retrieve the number of bytes indicated by the retrieved size. Such calls to the operating system can, however, be very slow in comparison to the invocations of local routines. To overcome the inefficiencies of such repeated calls, the broadcast technique in one embodiment, uses XDR to identify the message boundaries in a stream of messages. The broadcast technique may request the operating system to provide the next, for example, 1,024 bytes from the stream. The broadcast technique can then repeatedly invoke the XDR routines to retrieve the messages and use the success or failure of each invocation to determine whether another block of 1,024 bytes needs to be retrieved from the operating system. The invocation of XDR routines do not involve system calls and are thus more efficient than repeated system calls.

M-Regular

In the embodiment described above, each fully connected computer has four internal connections. The broadcast technique can be used with other numbers of internal connections. For example, each computer could have 6, 8, or any even number of internal connections. As the number of internal connections increase, the diameter of the broadcast channel tends to decrease, and thus propagation time for a message tends to decrease. The time that it takes to connect a seeking computer to the broadcast channel may, however, increase as the number of internal connections increases. When the number of internal connectors is even, then the broadcast channel can be maintained as m-regular and m-connected (in the steady state). If the number of internal connections is odd, then when the broadcast channel has an odd number of computers connected, one of the computers will have less than that odd number of internal connections. In such a situation, the broadcast network is neither m-regular nor m-connected. When the next computer connects to the broadcast channel, it can again become m-regular and m-connected. Thus, with an odd number of internal connections, the broadcast channel toggles between being and not being m-regular and m-connected.

Components

Figure 6 is a block diagram illustrating components of a computer that is connected to a broadcast channel. The above description generally assumed that there was only one broadcast channel and that each computer had only one connection to that broadcast channel. More generally, a network of computers may have multiple broadcast channels, each computer may be connected to more than one broadcast channel, and each computer can have multiple connections to the same broadcast channel. The broadcast channel is well suited for computer processes (*e.g.*, application programs) that execute collaboratively, such as network meeting programs. Each computer process can connect to one or more broadcast channels. The broadcast channels can be identified by channel type (*e.g.*, application program name) and channel instance that represents separate broadcast channels for that channel type. When a process attempts to connect to a broadcast channel, it seeks a process currently connected to that broadcast channel that is executing on a portal computer. The seeking process identifies the broadcast channel by channel type and channel instance.

Computer 600 includes multiple application programs 601 executing as separate processes. Each application program interfaces with a broadcaster component 602 for each broadcast channel to which it is connected. The broadcaster component may be implement as an object that is instantiated within the process space of the application program. Alternatively, the broadcaster component may execute as a separate process or thread from the application program. In one embodiment, the broadcaster component provides functions (*e.g.*, methods of class) that can be invoked by the application programs. The primary functions provided may include a connect function that an application program invokes passing an indication of the broadcast channel to which the application program wants to connect. The application program may provide a callback routine that the broadcaster component invokes to notify the application program that the connection has been completed, that is the process enters the fully connected state. The broadcaster component may also provide an acquire message function that the application program can invoke to retrieve the next message that is broadcast on the broadcast channel. Alternatively, the application program may provide a callback routine (which may be a virtual function provided by the application program) that the broadcaster component invokes to notify the application program that a broadcast message has been received. Each broadcaster component allocates a call-in port using the hashing algorithm. When calls are answered at the call-in port, they are transferred to other ports that serve as the external and internal ports.

The computers connecting to the broadcast channel may include a central processing unit, memory, input devices (*e.g.*, keyboard and pointing device), output devices (*e.g.*, display devices), and storage devices (*e.g.*, disk drives). The memory and storage devices are computer-readable medium that may contain computer instructions that implement the broadcaster component. In addition, the data structures and message structures may be stored or transmitted via a signal transmitted on a computer-readable media, such as a communications link.

Figure 7 is a block diagram illustrating the sub-components of the broadcaster component in one embodiment. The broadcaster component includes a connect component 701, an external dispatcher 702, an internal dispatcher 703 for each internal connection, an acquire message component 704 and a broadcast component 712. The application program may provide a connect callback component 710 and a receive response component 711 that are invoked by the broadcaster component. The application program invokes the connect component to establish a connection to a designated broadcast channel. The connect component identifies the external port and installs the external dispatcher for handling messages that are received on the external port. The connect component invokes the seek portal computer component 705 to identify a portal computer that is connected to the broadcast channel and invokes the connect request component 706 to ask the portal computer (if fully connected) to select neighbor processes for the newly connecting process. The external dispatcher receives external messages, identifies the type of message, and invokes the appropriate handling routine 707. The internal dispatcher receives the internal messages, identifies the type of message, and invokes the appropriate handling routine 708. The received broadcast messages are stored in the broadcast message queue 709. The acquire message component is invoked to retrieve messages from the broadcast queue. The broadcast component is invoked by the application program to broadcast messages in the broadcast channel.

A Distributed Conferencing System

In one embodiment, a conferencing system is implemented using the broadcast channel. Each participant in a conference connects to the conference's broadcast channel, and a participant is designated as the speaker. The conferencing application program may include a speaker component and an attendee component. The speaker component broadcasts the conference events on the broadcast channel. Each attendee component receives the conference events and displays the results of the conference events. For example, the speaker may present slides at the conference along with a description of each slide. Each attendee may receive an electronic copy of the slides in advance of the conference. At the scheduled time for the conference, the speaker and each attendee joins the conference by connecting to the broadcast channel of the conference. The speaker component allows the speaker to indicate when to display which slide. When a new slide is displayed, the speaker component broadcasts a new slide message. When the attendee component receives the new slide message, it displays the new slide to the participant. Also, the speaker component may allow the speaker to draw on a slide using a stylus or other pointing device. The speaker component then broadcasts draw messages on the broadcast channel so the attendee component can display the drawing to the attendees. The conferencing system may also use speech-to-text and text-to-speech to distribute the speaker's comments to all attendees.

The conferencing system may provide a directory web site where participants can locate and sign up for a conference of interest. The directory may provide a hierarchical categorization of scheduled conferences. When a user decides to sign up for a conference, the web server may download the broadcaster component and the conferencing application program to the attendee's computer, if not already stored on the attendee's computer. The web server will also download the channel type and channel instance associated with the broadcast channel for the conference along with the identification of the portal computers for the broadcast channel. The web server may also download the slides or other content to be displayed to the attendees during the conference.

The conferencing system may allow an entity to schedule conferences using the web site. For example, a software company may want to schedule a conference to announce a new product. The creation of the conference would entail the generation of a channel type and channel instance, the specification of a security level (e.g., encrypted messages), the specification of attendee qualifications, the providing of a description and scheduled time of the conference, the specification of the content to be distributed to the attendees, and so on. The speaker at a conference may not want to publicize the actual content (e.g., slides) in advance. In such a situation, the content can be encrypted when distributed to the attendees, and a key to decrypt the content can be distributed by the speaker during the conference. For example, each slide for the software company's announcement can be encrypted with a different key, and the appropriate key can be broadcast with each new slide message.

The conferencing system may allow attendees to broadcast comments on the broadcast channel. The times when an attendee can broadcast comments may be controlled by the speaker. For example, the speaker component may broadcast a comments allowed message and a comments not allowed message to delimit the times when comments will be allowed. Comments broadcast outside those times may be ignored. Alternatively, the attendees may be allowed to broadcast comments at any time, but the other attendees ignore those comments until the speaker broadcasts an approval message indicating that the attendee component can display a certain comment.

The conferencing system may allow each attendee to connect to and disconnect from the conference broadcast channel as this wish during the conference. In addition, the conferencing system may allow multiple speakers to share the "podium." The speakers can pass a speakers token between them to indicate who is currently speaking and thus in control of the conference. An attendee who joins the conference late may be able to synchronize with the conference by accessing a conference monitoring web server. The monitoring web server may be connected to the conference broadcast channel and monitor the current state of the conference. When an attendee joins late, the monitoring web server can provide the attendee with the current state of the conference. From then on, the attendee can listen on the broadcast channel to follow the progress of the conference. In addition, the attendee component may allow the attendee to view parts of the presentation other than that which is currently being presented. In this way, an attendee can refer back to or ahead to other portions of the presentation.

The following tables list messages sent by the broadcaster components.

EXTERNAL MESSAGES

Message Type	Description
seeking_connection_call	Indicates that a seeking process would like to know whether the receiving process is fully connected to the broadcast channel

connection_request_call	Indicates that the sending process would like the receiving process to initiate a connection of the sending process to the broadcast channel
edge_proposal_call	Indicates that the sending process is proposing an edge through which the receiving process can connect to the broadcast channel (*i.e.*, edge pinning)
port_connection_call	Indicates that the sending process is proposing a port through which the receiving process can connect to the broadcast channel
connected_stmt	Indicates that the sending process is connected to the broadcast channel
condition_repair_stmt	Indicates that the receiving process should disconnect from one of its neighbors and connect to one of the processes involved in the neighbors with empty port condition

INTERNAL MESSAGES

Message Type	Description
broadcast_stmt	Indicates a message that is being broadcast through the broadcast channel for the application programs
connection_port_search_stmt	Indicates that the designated process is looking for a port through which it can connect to the broadcast channel
connection_edge_search_call	Indicates that the requesting process is looking for an edge through which it can connect to the broadcast channel
connection_edge_search_resp	Indicates whether the edge between this process and the sending neighbor has been accepted by the requesting party
diameter_estimate_stmt	Indicates an estimated diameter of the broadcast channel
diameter_reset_stmt	Indicates to reset the estimated diameter to indicated diameter
disconnect_stmt	Indicates that the sending neighbor is disconnecting from the broadcast channel
condition_check_stmt	Indicates that neighbors with empty port condition have been detected
condition_double_check_stmt	Indicates that the neighbors with empty ports have the same set of neighbors
shutdown_stmt	Indicates that the broadcast channel is being shutdown

Flow Diagrams

Figures 8-34 are flow diagrams illustrating the processing of the broadcaster component in one embodiment. Figure 8 is a flow diagram illustrating the processing of the connect routine in one embodiment. This routine is passed a channel type (*e.g.*, application name) and channel instance (*e.g.*, session identifier), that identifies the broadcast channel to which this process wants to connect. The routine is also passed auxiliary information that includes the list of portal computers and a connection callback routine. When the connection is established, the connection callback routine is invoked to notify the application program. When this process invokes this routine, it is in the seeking connection state. When a portal computer is located that is connected and this routine connects to at least one neighbor, this process enters the partially connected state, and when the process eventually connects to four neighbors, it enters the fully connected state. When in the small regime, a fully connected process may have less than four neighbors. In block 801, the routine opens the call-in port through which the process is to communicate with other processes when establishing external and internal connections. The port is selected as the first available port using the hashing algorithm described above. In block 802, the routine sets the connect time to the current time. The connect time is used to identify the instance of the process that is connected through this external port. One process may connect to a broadcast channel of a certain channel type and channel instance using one call-in port and then disconnects, and another process may then connect to that same broadcast channel using the same call-in port. Before the other process becomes fully connected, another process may try to communicate with it thinking it is the fully connected old process. In such a case, the connect time can be used to identify this situation. In block 803, the routine invokes the seek portal computer routine passing the channel type and channel instance. The seek portal computer routine attempts to locate a portal computer through which this process can connect to the broadcast channel for the passed type and instance. In decision block 804, if the seek portal computer routine is successful in locating a fully connected process on that portal computer, then the routine continues at block 805, else the routine returns an unsuccessful indication. In decision block 805, if no portal computer other than the portal computer on which the process is executing was located, then this is the first process to fully connect to broadcast channel and the routine continues at block 806, else the routine continues at block 808. In block 806, the routine invokes the achieve connection routine to change the state of this process to fully connected. In block 807, the routine installs the external dispatcher for processing messages received through this process' external port for the passed channel type and channel instance. When a message is received through that external port, the external dispatcher is invoked. The routine then returns. In block 808, the routine installs an external dispatcher. In block 809, the routine invokes the connect request routine to initiate the process of identifying neighbors for the seeking computer. The routine then returns.

Figure 9 is a flow diagram illustrating the processing of the seek portal computer routine in one embodiment. This routine is passed the channel type and channel instance of the broadcast channel to which this process wishes to connect. This routine, for each search depth (*e.g.*, port number), checks the portal computers at that search depth. If a portal computer is located at that search depth with a process that is fully connected to the broadcast channel, then the routine returns an indication of success. In blocks 902-911, the routine loops selecting each search depth until a process is located. In block 902, the routine selects the next search depth using a port number ordering algorithm. In decision block 903, if all the search depths have already been selected during this execution of the loop, that is for the currently selected depth, then the routine returns a failure indication, else the routine continues at block 904. In blocks 904-911, the routine loops selecting each portal computer and determining whether a process of that portal computer is connected to (or attempting to connect to) the broadcast channel with the passed channel type and channel instance. In block 904, the routine selects the next portal computer. In decision block 905, if all the portal computers have already been selected, then the routine loops to block 902 to select the next search depth, else the routine continues at block 906. In block 906, the routine dials the selected portal computer through the port represented by the search depth. In decision block 907, if the dialing was successful, then the routine continues at block 908, else the routine loops to block 904 to select the next portal computer. The dialing will be successful if the dialed port is the call-in port of the broadcast channel of the passed channel type and channel instance of a process executing on that portal computer. In block 908, the routine invokes a contact process routine, which contacts the answering process of the portal computer through the dialed port and determines whether that process is fully connected to the broadcast channel. In block 909, the routine hangs up on the selected portal computer. In decision block 910, if the answering process is fully connected to the broadcast channel, then the routine returns a success indicator, else the routine continues at block 911. In block 911, the routine invokes the check for external call routine to determine whether an external call has been made to this process as a portal computer and processes that call. The routine then loops to block 904 to select the next portal computer.

Figure 10 is a flow diagram illustrating the processing of the contact process routine in one embodiment. This routine determines whether the process of the selected portal computer that answered the call-in to the selected port is fully connected to the broadcast channel. In block 1001, the routine sends an external message (*i.e.*, seeking_connection_call) to the answering process indicating that a seeking process wants to know whether the answering process is fully connected to the broadcast channel. In block 1002, the routine receives the external response message from the answering process. In decision block 1003, if the external response message is successfully received (*i.e.*, seeking_connection_resp), then the routine continues at block 1004, else the routine returns. Wherever the broadcast component requests to receive an external message, it sets a time out period. If the external message is not received within that time out period, the broadcaster component checks its own call-in port to see if another process is calling it. In particular, the dialed process may be calling the dialing process, which may result in a deadlock situation. The broadcaster component may repeat the receive request several times. If the expected message is not received, then the broadcaster component handles the error as appropriate. In decision block 1004, if the answering process indicates in its response message that it is fully connected to the broadcast channel, then the routine continues at block 1005, else the routine continues at block 1006. In block 1005, the routine adds the selected portal computer to a list of connected portal computers and then returns. In block 1006, the routine adds the answering process to a list of fellow seeking processes and then returns.

Figure 11 is a flow diagram illustrating the processing of the connect request routine in one embodiment. This routine requests a process of a portal computer that was identified as being fully connected to the broadcast channel to initiate the connection of this process to the broadcast channel. In decision block 1101, if at least one process of a portal computer was located that is fully connected to the broadcast channel, then the routine continues at block 1103, else the routine continues at block 1102. A process of the portal computer may no longer be in the list if it recently disconnected from the broadcast channel. In one embodiment, a seeking computer may always search its entire search depth and find multiple portal computers through which it can connect to the broadcast channel. In block

1102, the routine restarts the process of connecting to the broadcast channel and returns. In block 1103, the routine dials the process of one of the found portal computers through the call-in port. In decision block 1104, if the dialing is successful, then the routine continues at block 1105, else the routine continues at block 1113. The dialing may be unsuccessful if, for example, the dialed process recently disconnected from the broadcast channel. In block 1105, the routine sends an external message to the dialed process requesting a connection to the broadcast channel (*i.e.*, connection_request_call). In block 1106, the routine receives the response message (*i.e.*, connection_request_resp). In decision block 1107, if the response message is successfully received, then the routine continues at block 1108, else the routine continues at block 1113. In block 1108, the routine sets the expected number of holes (*i.e.*, empty internal connections) for this process based on the received response. When in the large regime, the expected number of holes is zero. When in the small regime, the expected number of holes varies from one to three. In block 1109, the routine sets the estimated diameter of the broadcast channel based on the received response. In decision block 1111, if the dialed process is ready to connect to this process as indicated by the response message, then the routine continues at block 1112, else the routine continues at block 1113. In block 1112, the routine invokes the add neighbor routine to add the answering process as a neighbor to this process. This adding of the answering process typically occurs when the broadcast channel is in the small regime. When in the large regime, the random walk search for a neighbor is performed. In block 1113, the routine hangs up the external connection with the answering process computer and then returns.

Figure 12 is a flow diagram of the processing of the check for external call routine in one embodiment. This routine is invoked to identify whether a fellow seeking process is attempting to establish a connection to the broadcast channel through this process. In block 1201, the routine attempts to answer a call on the call-in port. In decision block 1202, if the answer is successful, then the routine continues at block 1203, else the routine returns. In block 1203, the routine receives the external message from the external port. In decision block 1204, if the type of the message indicates that a seeking process is calling (*i.e.*, seeking_connection_call), then the routine continues at block 1205, else the routine returns. In block 1205, the routine sends an external message (*i.e.*, seeking_connection_resp) to the other seeking process indicating that this process is also is seeking a connection. In decision block 1206, if the sending of the external message is successful, then the routine continues at block 1207, else the routine returns. In block 1207, the routine adds the other seeking process to a list of fellow seeking processes and then returns. This list may be used if this process can find no process that is fully connected to the broadcast channel. In which case, this process may check to see if any fellow seeking process were successful in connecting to the broadcast channel. For example, a fellow seeking process may become the first process fully connected to the broadcast channel.

Figure 13 is a flow diagram of the processing of the achieve connection routine in one embodiment. This routine sets the state of this process to fully connected to the broadcast channel and invokes a callback routine to notify the application program that the process is now fully connected to the requested broadcast channel. In block 1301, the routine sets the connection state of this process to fully connected. In block 1302, the routine notifies fellow seeking processes that it is fully connected by sending a connected external message to them (*i.e.*, connected_stmt). In block 1303, the routine invokes the connect callback routine to notify the application program and then returns.

Figure 14 is a flow diagram illustrating the processing of the external dispatcher routine in one embodiment. This routine is invoked when the external port receives a message. This routine retrieves the message, identifies the external message type, and invokes the appropriate routine to handle that message. This routine loops processing each message until all the received messages have been handled. In block 1401, the routine answers (*e.g.*, picks up) the external port and retrieves an external message. In decision block 1402, if a message was retrieved, then the routine continues at block 1403, else the routine hangs up on the external port in block 1415 and returns. In decision block 1403, if the message type is for a process seeking a connection (*i.e.*, seeking_connection_call), then the routine invokes the handle seeking connection call routine in block 1404, else the routine continues at block 1405. In decision block 1405, if the message type is for a connection request call (*i.e.*, connection_request_call), then the routine invokes the handle connection request call routine in block 1406, else the routine continues at block 1407. In decision block 1407, if the message type is edge proposal call (*i.e.*, edge_proposal_call), then the routine invokes the handle edge proposal call routine in block 1408, else the routine continues at block 1409. In decision block 1409, if the message type is port connect call (*i.e.*, port_connect_call), then the routine invokes the handle port connection call routine in block 1410, else the routine continues at block 1411. In decision block 1411, if the message type is a connected statement (*i.e.*, connected_stmt), the routine invokes the handle connected statement in block 1112, else the routine continues at block 1212. In decision block 1412, if the message type is a condition repair statement (*i.e.*, condition_repair_stmt), then the routine invokes the handle condition repair routine in block 1413, else the routine loops to block 1414 to process the next message. After each handling routine is invoked, the routine loops to block 1414. In block 1414, the routine hangs up on the external port and continues at block 1401 to receive the next message.

Figure 15 is a flow diagram illustrating the processing of the handle seeking connection call routine in one embodiment. This routine is invoked when a seeking process is calling to identify a portal computer through which it can connect to the broadcast channel. In decision block 1501, if this process is currently fully connected to the broadcast channel identified in the message, then the routine continues at block 1502, else the routine continues at block 1503. In block 1502, the routine sets a message to indicate that this process is fully connected to the broadcast channel and continues at block 1505. In block 1503, the routine sets a message to indicate that this process is not fully connected. In block 1504, the routine adds the identification of the seeking process to a list of fellow seeking processes. If this process is not fully connected, then it is attempting to connect to the broadcast channel. In block 1505, the routine sends the external message response (*i.e.*, seeking_connection_resp) to the seeking process and then returns.

Figure 16 is a flow diagram illustrating processing of the handle connection request call routine in one embodiment. This routine is invoked when the calling process wants this process to initiate the connection of the process to the broadcast channel. This routine either allows the calling process to establish an internal connection with this process (*e.g.*, if in the small regime) or starts the process of identifying a process to which the calling process can connect. In decision block 1601, if this process is currently fully connected to the broadcast channel, then the routine continues at block 1603, else the routine hangs up on the external port in block 1602 and returns. In block 1603, the routine sets the number of holes that the calling process should expect in the response message. In block 1604, the routine sets the estimated diameter in the response message. In block 1605, the routine indicates whether this process is ready to connect to the calling process. This process is ready to connect when the number of its holes is greater than zero and the calling process is not a neighbor of this process. In block 1606, the routine sends to the calling process an external message that is responsive to the connection request call (*i.e.*, connection_request_resp). In block 1607, the routine notes the number of holes that the calling process needs to fill as indicated in the request message. In decision block 1608, if this process is ready to connect to the calling process, then the routine continues at block 1609, else the routine continues at block 1611. In block 1609, the routine invokes the add neighbor routine to add the calling process as a neighbor. In block 1610, the routine decrements the number of holes that the calling process needs to fill and continues at block 1611. In block 1611, the routine hangs up on the external port. In decision block 1612, if this process has no holes or the estimated diameter is greater than one (*i.e.*, in the large regime), then the routine continues at block 1613, else the routine continues at block 1616. In blocks 1613-1615, the routine loops forwarding a request for an edge through which to connect to the calling process to the broadcast channel. One request is forwarded for each pair of holes of the calling process that needs to be filled. In decision block 1613, if the number of holes of the calling process to be filled is greater than or equal to two, then the routine continues at block 1614, else the routine continues at block 1616. In block 1614, the routine invokes the forward connection edge search routine. The invoked routine is passed to an indication of the calling process and the random walk distance. In one embodiment, the distance is twice in the estimated diameter of the broadcast channel. In block 1614, the routine decrements the holes left to fill by two and loops to block 1613. In decision block 1616, if there is still a hole to fill, then the routine continues at block 1617, else the routine returns. In block 1617, the routine invokes the fill hole routine passing the identification of the calling process. The fill hole routine broadcasts a connection port search statement (*i.e.*, connection_port_search_stmt) for a hole of a connected process through which the calling process can connect to the broadcast channel. The routine then returns.

Figure 17 is a flow diagram illustrating the processing of the add neighbor routine in one embodiment. This routine adds the process calling on the external port as a neighbor to this process. In block 1701, the routine identifies the calling process on the external port. In block 1702, the routine sets a flag to indicate that the neighbor has not yet received the broadcast messages from this process. This flag is used to ensure that there are no gaps in the messages initially sent to the new neighbor. The external port becomes the internal port for this connection. In decision block 1703, if this process is in the seeking connection state, then this process is connecting to its first neighbor and the routine continues at block 1704, else the routine continues at block 1705. In block 1704, the routine sets the connection state of this process to partially connected. In block 1705, the routine adds the calling process to the list of neighbors of this process. In block 1706, the routine installs an internal dispatcher for the new neighbor. The internal dispatcher is invoked when a message is received from that new neighbor through the internal port of that new neighbor. In decision block 1707, if this process buffered up messages while not fully connected, then the routine continues at block 1708, else the routine continues at block 1709. In one embodiment, a process that is partially connected may buffer the messages that it receives through an internal connection so that it can send these messages as it connects to new neighbors. In block 1708, the routine sends the buffered messages to the new neighbor through the internal port. In decision block 1709, if the number of holes of this process equals the expected number of holes, then this process is fully connected and the routine continues at block 1710, else the routine continues at block 1711. In block 1710, the routine invokes the achieve connected routine to indicate that this process is fully connected. In decision block 1711, if the number of holes for this process is zero, then the routine continues at block 1712, else the routine returns. In block 1712, the routine deletes any pending edges and then returns. A pending edge is an edge that has been proposed to this process for edge pinning, which in this case is no longer needed.

Figure 18 is a flow diagram illustrating the processing of the forward connection edge search routine in one embodiment. This routine is responsible for passing along a request to connect a requesting process to a randomly selected neighbor of this process through the internal port of the selected neighbor, that is part of the random walk. In decision block 1801, if the forwarding distance remaining is greater than zero, then the routine continues at block 1804, else the routine continues at block 1802. In decision block 1802, if the number of neighbors of this process is greater than one, then the routine continues at block 1804, else this broadcast channel is in the small regime and the routine continues at block 1803. In decision block 1803, if the requesting process is a neighbor of this process, then the routine returns, else the routine continues at block 1804. In blocks 1804-1807, the routine loops attempting to send a connection edge search call internal message (*i.e.*, connection_edge_search_call) to a randomly selected neighbor. In block 1804, the routine randomly selects a neighbor of this process. In decision block 1805, if all the neighbors of this process have already been selected, then the routine cannot forward the message and the routine returns, else the routine continues at block 1806. In block 1806, the routine sends a connection edge search call internal message to the selected neighbor. In decision block 1807, if the sending of the message is successful, then the routine continues at block 1808, else the routine loops to block 1804 to select the next neighbor. When the sending of an internal message is unsuccessful, then the neighbor may have disconnected from the broadcast channel in an unplanned manner. Whenever such a situation is detected by the broadcaster component, it attempts to find another neighbor by invoking the fill holes routine to fill a single hole or the forward connecting edge search routine to fill two holes. In block 1808, the routine notes that the recently sent connection edge search call has not yet been acknowledged and indicates that the edge to this neighbor is reserved if the remaining forwarding distance is less than or equal to one. It is reserved because the selected neighbor may offer this edge to the requesting process for edge pinning. The routine then returns.

Figure 19 is a flow diagram illustrating the processing of the handle edge proposal call routine. This routine is invoked when a message is received from a proposing process that proposes to connect an edge between the proposing process and one of its neighbors to this process for edge pinning. In decision block 1901, if the number of holes of this process minus the number of pending edges is greater than or equal to one, then this process still has holes to be filled and the routine continues at block 1902, else the routine continues at block 1911. In decision block 1902, if the proposing process or its neighbor is a neighbor of this process, then the routine continues at block 1911, else the routine continues at block 1903. In block 1903, the routine indicates that the edge is pending between this process and the proposing process. In decision block 1904, if a proposed neighbor is already pending as a proposed neighbor, then the routine continues at block 1911, else the routine continues at block 1907. In block 1907, the routine sends an edge proposal response as an external message to the proposing process (*i.e.*, edge_proposal_resp) indicating that the proposed edge is accepted. In decision block 1908, if the sending of the message was successful, then the routine continues at block 1909, else the routine returns. In block 1909, the routine adds the edge as a pending edge. In block 1910, the routine invokes the add neighbor routine to add the proposing process on the external port as a neighbor. The routine then returns. In block 1911, the routine sends an external message (*i.e.*, edge_proposal_resp) indicating that this proposed edge is not accepted. In decision block 1912, if the number of holes is odd, then the routine continues at block 1913, else the routine returns. In block 1913, the routine invokes the fill hole routine and then returns.

Figure 20 is a flow diagram illustrating the processing of the handle port connection call routine in one embodiment. This routine is invoked when an external message is received then indicates that the sending process wants to connect to one hole of this process. In decision block 2001, if the number of holes of this process is greater than zero, then the routine continues at block 2002, else the routine continues at block 2003. In decision block 2002, if the sending process is not a neighbor, then the routine continues at block 2004, else the routine continues to block 2003. In block 2003, the routine sends a port connection response external message (*i.e.*, port_connection_resp) to the sending process that indicates that it is not okay to connect to this process. The routine then returns. In block 2004, the routine sends a port connection response external message to the sending process that indicates that is okay to connect this process. In decision block 2005, if the sending of the message was successful, then the routine continues at block 2006, else the routine continues at block 2007. In block 2006, the routine invokes the add neighbor routine to add the sending process as a neighbor of this process and then returns. In block 2007, the routine hangs up the external connection. In block 2008, the routine invokes the connect request routine to request that a process connect to one of the holes of this process. The routine then returns.

Figure 21 is a flow diagram illustrating the processing of the fill hole routine in one embodiment. This routine is passed an indication of the requesting process. If this process is requesting to fill a hole, then this routine sends an internal message to other processes. If another process is requesting to fill a hole, then this routine invokes the routine to handle a connection port search request. In block 2101, the routine initializes a connection port search statement internal message (*i.e.*, connection_port_search_stmt). In decision block 2102, if this process is the requesting process, then the routine continues at block 2103, else the routine continues at block 2104. In block 2103, the routine distributes the message to the neighbors of this process through the internal ports and then returns. In block 2104, the routine invokes the handle connection port search routine and then returns.

Figure 22 is a flow diagram illustrating the processing of the internal dispatcher routine in one embodiment. This routine is passed an indication of the neighbor who sent the internal message. In block 2201, the routine receives the internal message. This routine identifies the message type and invokes the appropriate routine to handle the message. In block 2202, the routine assesses whether to change the estimated diameter of the broadcast channel based on the information in the received message. In decision block 2203, if this process is the originating process of the message or the message has already been received (*i.e.*, a duplicate), then the routine ignores the message and continues at block 2208, else the routine continues at block 2203A. In decision block 2203A, if the process is partially connected, then the routine continues at block 2203B, else the routine continues at block 2204. In block 2203B, the routine adds the message to the pending connection buffer and continues at block 2204. In decision blocks 2204-2207, the routine decodes the message type and invokes the appropriate routine to handle the message. For example, in decision block 2204, if the type of the message is broadcast statement (*i.e.*, broadcast_stmt), then the routine invokes the handle broadcast message routine in block 2205. After invoking the appropriate handling routine, the routine continues at block 2208. In decision block 2208, if the partially connected buffer is full, then the routine continues at block 2209, else the routine continues at block 2210. The broadcaster component collects all its internal messages in a buffer while partially connected so that it can forward the messages as it connects to new neighbors. If, however, that buffer becomes full, then the process assumes that it is now fully connected and that the expected number of connections was too high, because the broadcast channel is now in the small regime. In block 2209, the routine invokes the achieve connection routine and then continues in block 2210. In decision block 2210, if the application program message queue is empty, then the routine returns, else the routine continues at block 2212. In block 2212, the routine invokes the receive response routine passing the acquired message and then returns. The received response routine is a callback routine of the application program.

Figure 23 is a flow diagram illustrating the processing of the handle broadcast message routine in one embodiment. This routine is passed an indication of the originating process, an indication of the neighbor who sent the broadcast message, and the broadcast message itself. In block 2301, the routine performs the out of order processing for this message. The broadcaster component queues messages from each originating process until it can send them in sequence number order to the application program. In block 2302, the routine invokes the distribute broadcast message routine to forward the message to the neighbors of this process. In decision block 2303, if a newly connected neighbor is waiting to receive messages, then the routine continues at block 2304, else the routine returns. In block 2304, the routine sends the messages in the correct order if possible for each originating process and then returns.

Figure 24 is a flow diagram illustrating the processing of the distribute broadcast message routine in one embodiment. This routine sends the broadcast message to each of the neighbors of this process, except for the neighbor who sent the message to this process. In block 2401, the routine selects the next neighbor other than the neighbor who sent the message. In decision block 2402, if all such neighbors have already been selected, then the routine returns. In block 2403, the routine sends the message to the selected neighbor and then loops to block 2401 to select the next neighbor.

Figure 26 is a flow diagram illustrating the processing of the handle connection port search statement routine in one embodiment. This routine is passed an indication of the neighbor that sent the message and the message itself. In block 2601, the routine invokes the distribute internal message which sends the message to each of its neighbors other than the sending neighbor. In decision block 2602, if the number of holes of this process is greater than zero, then the routine continues at block 2603, else the routine returns. In decision block 2603, if the requesting process is a neighbor, then the routine continues at block 2605, else the routine continues at block 2604. In block 2604, the routine invokes the court neighbor routine and then returns. The court neighbor routine connects this process to the requesting process if possible. In block 2605, if this process has one hole, then the neighbors with empty ports condition exists and the routine continues at block 2606, else the routine returns. In block 2606, the routine generates a condition check message (*i.e.*, condition_check) that includes a list of this process' neighbors. In block 2607, the routine sends the message to the requesting neighbor.

Figure 27 is a flow diagram illustrating the processing of the court neighbor routine in one embodiment. This routine is passed an indication of the prospective neighbor for this process. If this process can connect to the prospective neighbor, then it sends a port connection call external message to the prospective neighbor and adds the prospective neighbor as a neighbor. In decision block 2701, if the prospective neighbor is already a neighbor, then the routine returns, else the routine continues at block 2702. In block 2702, the routine dials the prospective neighbor. In decision block 2703, if the number of holes of this process is greater than zero, then the routine continues at block 2704, else the routine continues at block 2706. In block 2704, the routine sends a port connection call external message (*i.e.*, port_connection_call) to the prospective neighbor and receives its response (*i.e.*, port_connection_resp). Assuming the response is successfully received, in block 2705, the routine adds the prospective neighbor as a neighbor of this process by invoking the add neighbor routine. In block 2706, the routine hangs up with the prospect and then returns.

Figure 28 is a flow diagram illustrating the processing of the handle connection edge search call routine in one embodiment. This routine is passed a indication of the neighbor who sent the message and the message itself. This routine either forwards the message to a neighbor or proposes the edge between this process and the sending neighbor to the requesting process for edge pinning. In decision block 2801, if this process is not the requesting process or the number of holes of the requesting process is still greater than or equal to two, then the routine continues at block 2802, else the routine continues at block 2813. In decision block 2802, if the forwarding distance is greater than zero, then the random walk is not complete and the routine continues at block 2803, else the routine continues at block 2804. In block 2803, the routine invokes the forward connection edge search routine passing the identification of the requesting process and the decremented forwarding distance. The routine then continues at block 2815. In decision block 2804, if the requesting process is a neighbor or the edge between this process and the sending neighbor is reserved because it has already been offered to a process, then the routine continues at block 2805, else the routine continues at block 2806. In block 2805, the routine invokes the forward connection edge search routine passing an indication of the requesting party and a toggle indicator that alternatively indicates to continue the random walk for one or two more computers. The routine then continues at block 2815. In block 2806, the routine dials the requesting process via the call-in port. In block 2807, the routine sends an edge proposal call external message (*i.e.*, edge_proposal_call) and receives the response (*i.e.*, edge_proposal_resp). Assuming that the response is successfully received, the routine continues at block 2808. In decision block 2808, if the response indicates that the edge is acceptable to the requesting process, then the routine continues at block 2809, else the routine continues at block 2812. In block 2809, the routine reserves the edge between this process and the sending neighbor. In block 2810, the routine adds the requesting process as a neighbor by invoking the add neighbor routine. In block 2811, the routine removes the sending neighbor as a neighbor. In block 2812, the routine hangs up the external port and continues at block 2815. In decision block 2813, if this process is the requesting process and the number of holes of this process equals one, then the routine continues at block 2814, else the routine continues at block 2815. In block 2814, the routine invokes the fill hole routine. In block 2815, the routine sends an connection edge search response message (*i.e.*, connection_edge_search_response) to the sending neighbor indicating acknowledgement and then returns. The graphs are sensitive to parity. That is, all possible paths starting from a node and ending at that node will have an even length unless the graph has a cycle whose length is odd. The broadcaster component uses a toggle indicator to vary the random walk distance between even and odd distances.

Figure 29 is a flow diagram illustrating the processing of the handle connection edge search response routine in one embodiment. This routine is passed as indication of the requesting process, the sending neighbor, and the message. In block 2901, the routine notes that the connection edge search response (*i.e.*, connection_edge_search_resp) has been received and if the forwarding distance is less than or equal to one unreserves the edge between this process and the sending neighbor. In decision block 2902, if the requesting process indicates that the edge is acceptable as indicated in the message, then the routine continues at block 2903, else the routine returns. In block 2903, the routine reserves the edge between this process and the sending neighbor. In block 2904, the routine removes the sending neighbor as a neighbor. In block 2905, the routine invokes the court neighbor routine to connect to the requesting process. In decision block 2906, if the invoked routine was unsuccessful, then the routine continues at block 2907, else the routine returns. In decision block 2907, if the number of holes of this process is greater than zero, then the routine continues at block 2908, else the routine returns. In block 2908, the routine invokes the fill hole routine and then returns.

Figure 30 is a flow diagram illustrating the processing of the broadcast routine in one embodiment. This routine is invoked by the application program to broadcast a message on the broadcast channel. This routine is passed the message to be broadcast. In decision block 3001, if this process has at least one neighbor, then the routine continues at block 3002, else the routine returns since it is the only process connected to be broadcast channel. In block 3002, the routine generates an internal message of the broadcast statement type (*i.e.*, broadcast _stmt). In block 3003, the routine sets the sequence number of the message. In block 3004, the routine invokes the distribute internal message routine to broadcast the message on the broadcast channel. The routine returns.

Figure 31 is a flow diagram illustrating the processing of the acquire message routine in one embodiment. The acquire message routine may be invoked by the application program or by a callback routine provided by the application program. This routine returns a message. In block 3101, the routine pops the message from the message queue of the broadcast channel. In decision block 3102, if a message was retrieved, then the routine returns an indication of success, else the routine returns indication of failure.

Figures 32-34 are flow diagrams illustrating the processing of messages associated with the neighbors with empty ports condition. Figure 32 is a flow diagram illustrating processing of the handle condition check message in one embodiment. This message is sent by a neighbor process that has one hole and has received a request to connect to a hole of this process. In decision block 3201, if the number of holes of this process is equal to one, then the routine continues at block 3202, else the neighbors with empty ports condition does not exist any more and the routine returns. In decision block 3202, if the sending neighbor and this process have the same set of neighbors, the routine continues at block 3203, else the routine continues at block 3205. In block 3203, the routine initializes a condition double check message (*i.e.*, condition_double_check) with the list of neighbors of this process. In block 3204, the routine sends the message internally to a neighbor other than sending neighbor. The routine then returns. In block 3205, the routine selects a neighbor of the sending process that is not also a neighbor of this process. In block 3206, the routine sends a condition repair message (*i.e.*, condition_repair_stmt) externally to the selected process. In block 3207, the routine invokes the add neighbor routine to add the selected neighbor as a neighbor of this process and then returns.

Figure 33 is a flow diagram illustrating processing of the handle condition repair statement routine in one embodiment. This routine removes an existing neighbor and connects to the process that sent the message. In decision block 3301, if this process has no holes, then the routine continues at block 3302, else the routine continues at block 3304. In block 3302, the routine selects a neighbor that is not involved in the neighbors with empty ports condition. In block 3303, the routine removes the selected neighbor as a neighbor of this process. Thus, this process that is executing the routine now has at least one hole. In block 3304, the routine invokes the add neighbor routine to add the process that sent the message as a neighbor of this process. The routine then returns.

Figure 34 is a flow diagram illustrating the processing of the handle condition double check routine. This routine determines whether the neighbors with empty ports condition really is a problem or whether the broadcast channel is in the small regime. In decision block 3401, if this process has one hole, then the routine continues at block 3402, else the routine continues at block 3403. If this process does not have one hole, then the set of neighbors of this process is not the same as the set of neighbors of the sending process. In decision block 3402, if this process and the sending process have the same set of neighbors, then the broadcast channel is not in the small regime and the routine continues at block 3403, else the routine continues at block 3406. In decision block 3403, if this process has no holes, then the routine returns, else the routine continues at block 3404. In block 3404, the routine sets the estimated diameter for this process to one. In block 3405, the routine broadcasts a diameter reset internal message (*i.e.*, diameter_reset) indicating that the estimated diameter is one and then returns. In block 3406, the routine creates a list of neighbors of this process. In block 3407, the routine sends the condition check message (*i.e.*, condition_check_stmt) with the list of neighbors to the neighbor who sent the condition double check message and then returns.

From the above description, it will be appreciated that although specific embodiments of the technology have been described, various modifications may be made without deviating from the spirit and scope of the invention. For example, the communications on the broadcast channel may be encrypted. Also, the channel instance or session identifier may be a very large number (*e.g.*, 128 bits) to help prevent an unauthorized user to maliciously tap into a broadcast channel. The portal computer may also enforce security and not allow an unauthorized user to connect to the broadcast channel. Accordingly, the invention is not limited except by the claims.

CLAIMS

1. A computer network for providing a conferencing system for a plurality of participants, each participant having connections to at least three neighbor participants, wherein an originating participant sends data to the other participants by sending the data through each of its connections to its neighbor participants and wherein each participant sends data that it receives from a neighbor participant to its other neighbor participants.

2. The computer network of claim 1 wherein each participant is connected to 4 other participants.

3. The computer network of claim 1 wherein each participant is connected to an even number of other participants.

4. The computer network of claim 1 wherein the network is m-regular, where m is the number of neighbor participants of each participant.

5. The computer network of claim 1 wherein the network is m-connected, where m is the number of neighbor participants of each participant.

6. The computer network of claim 1 wherein the network is m-regular and m-connected, where m is the number of neighbor participants of each participant.

7. The computer network of claim 1 wherein all the participants are peers.

8. The computer network of claim 1 wherein the connections are peer-to-peer connections.

9. The computer network of claim 1 wherein the connections are TCP/IP connections.

10. The computer network of claim 1 wherein each participant is a process executing on a computer.

11. The computer network of claim 1 wherein a computer hosts more than one participant.

12. The computer network of claim 1 wherein each participant sends to each of its neighbors only one copy of the data.

13. The computer network of claim 1 wherein the interconnections of participants form a broadcast channel for a topic of interest.

14. A distributed conferencing system comprising:

a plurality of broadcast channels, each broadcast channel for conducting a conference;

means for identifying a broadcast channel for a conference of interest; and

means for connecting to the identified broadcast channel.

15. The distributed conferencing system of claim 14 wherein means for identifying a conference of interest includes accessing a web server that maps conferences to corresponding broadcast channel.

16. The distributed conferencing system of claim 14 wherein a broadcast channel is formed by attendee computers and a speaker computer that are each interconnected to at least three other computers.

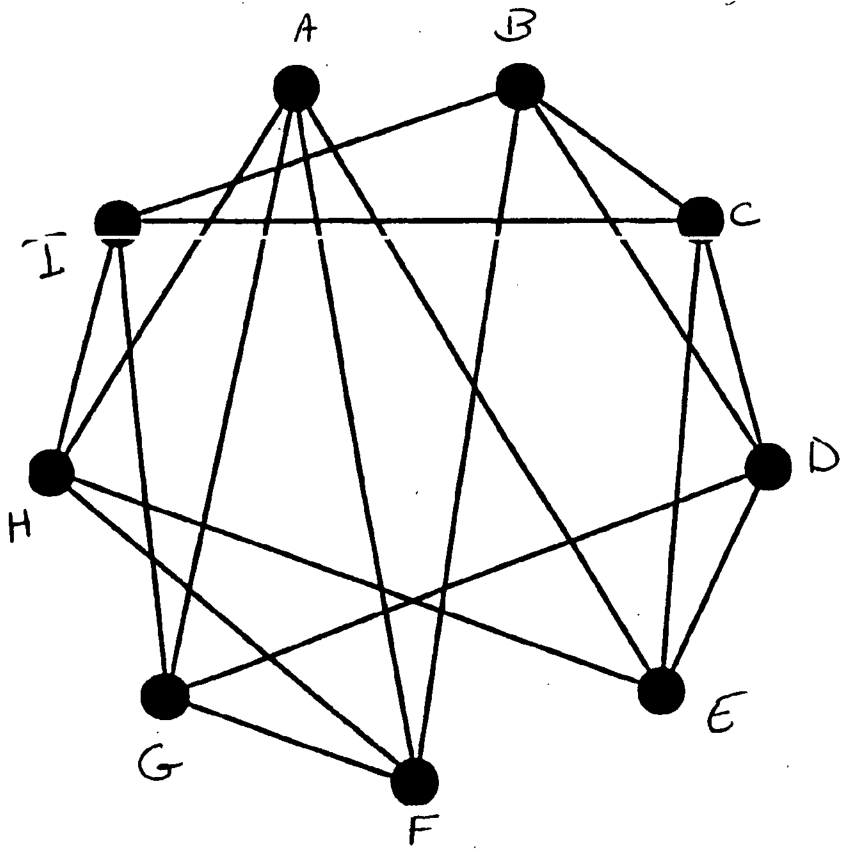


Fig 1

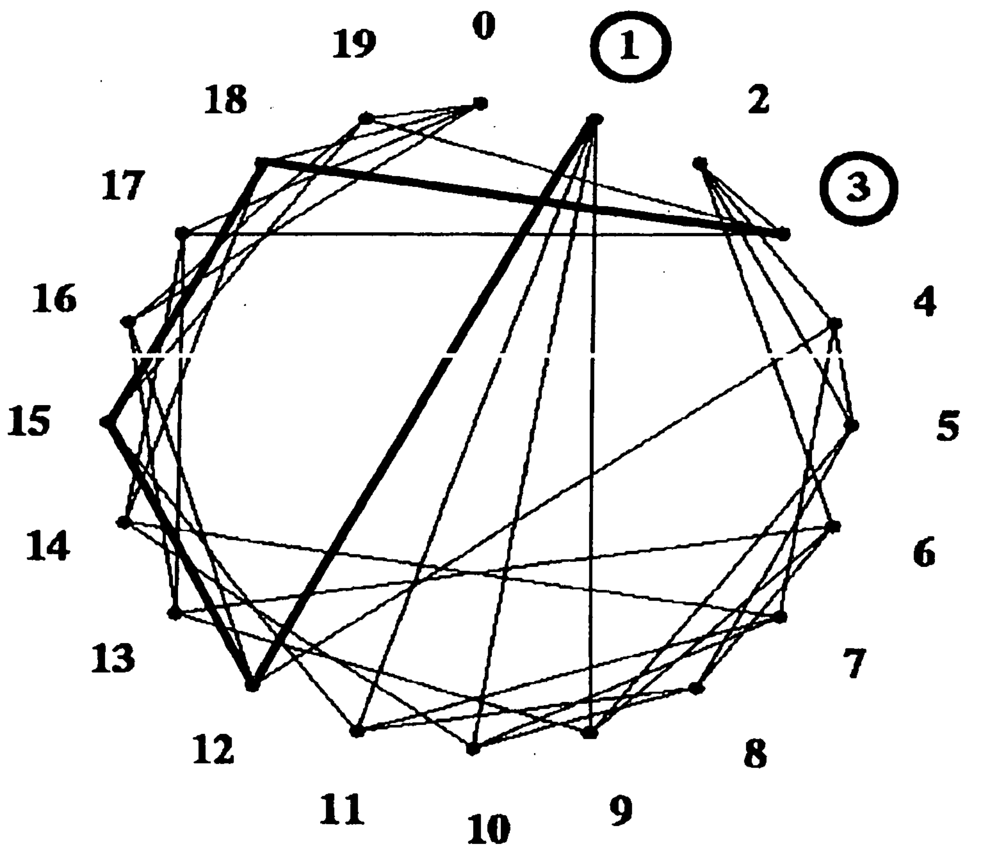

0
①
2
③
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

Fig 2

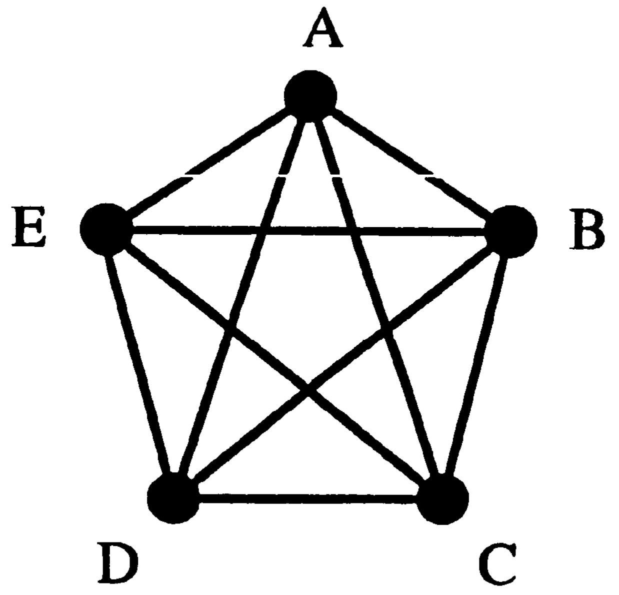


Fig 3A

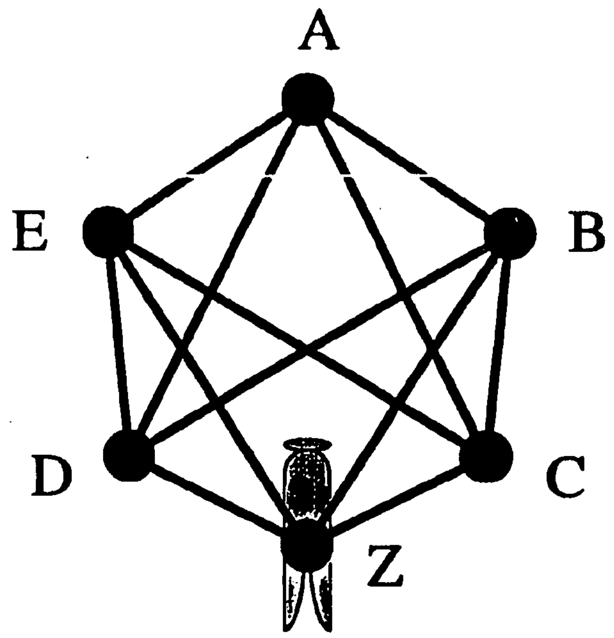


Fig 3B

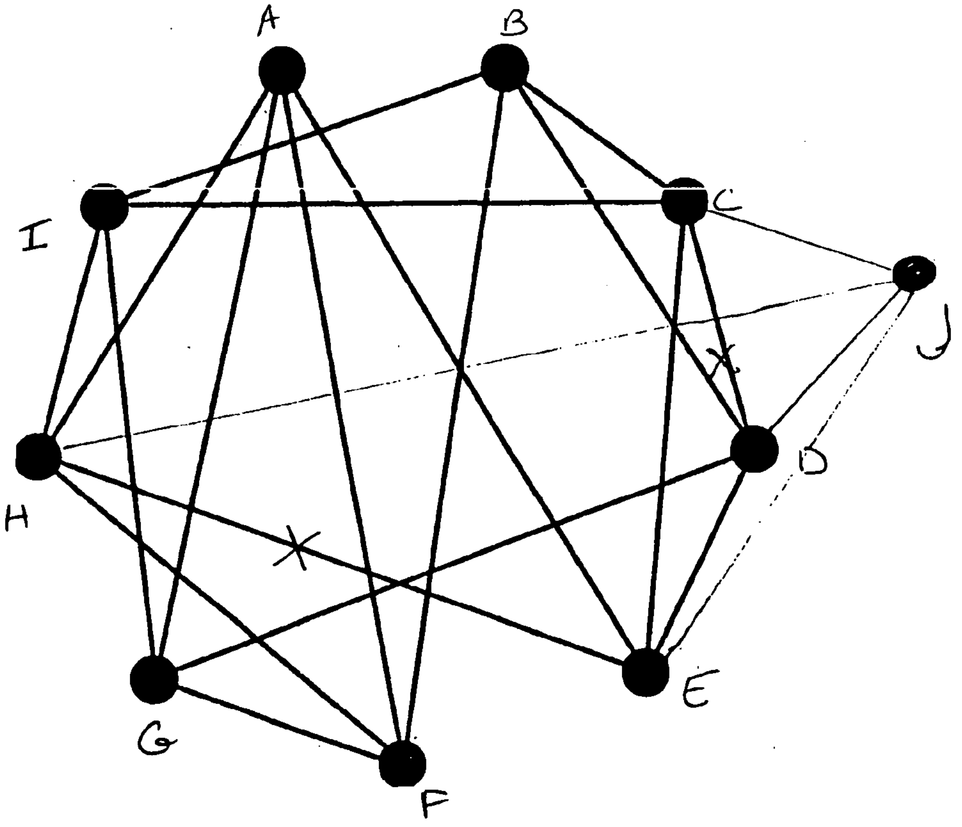

A
B
C
J
D
E
F
G
H
I
X
X

Fig 4A

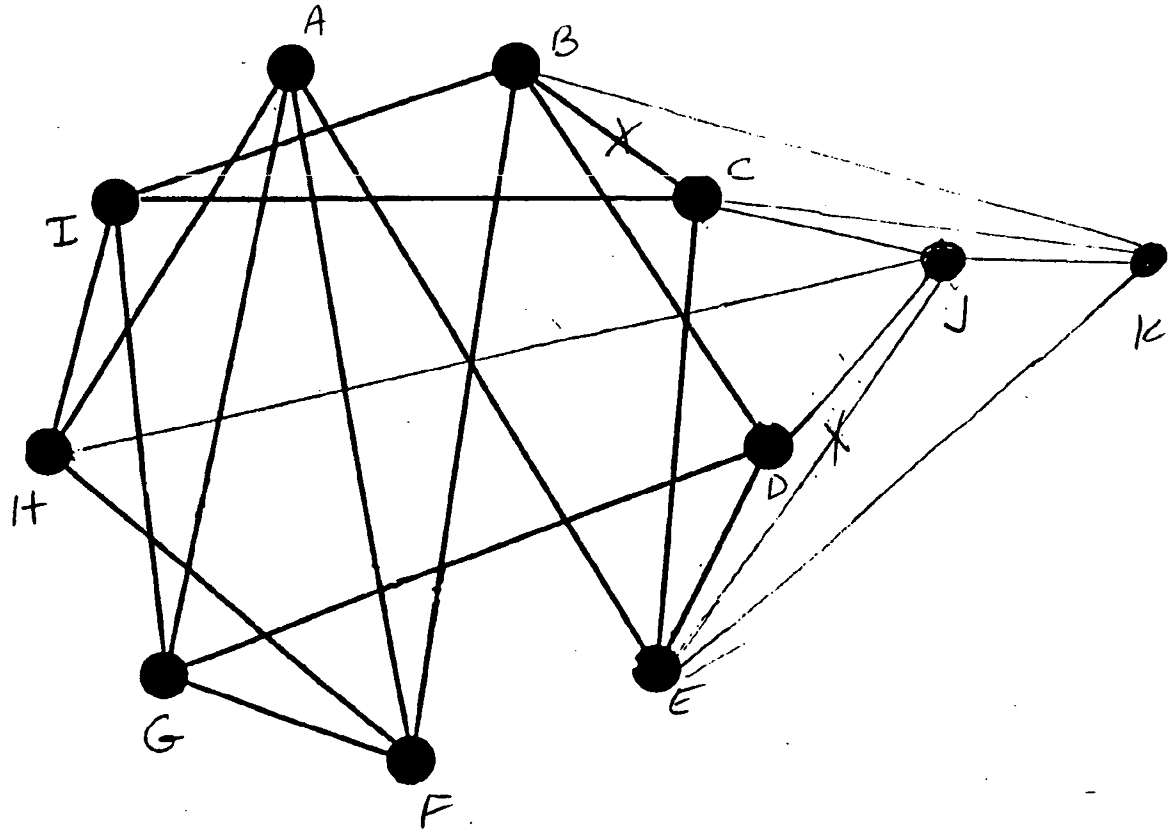

A
B
C
I
J
K
H
D
X
X
G
E
F

Fig 4B

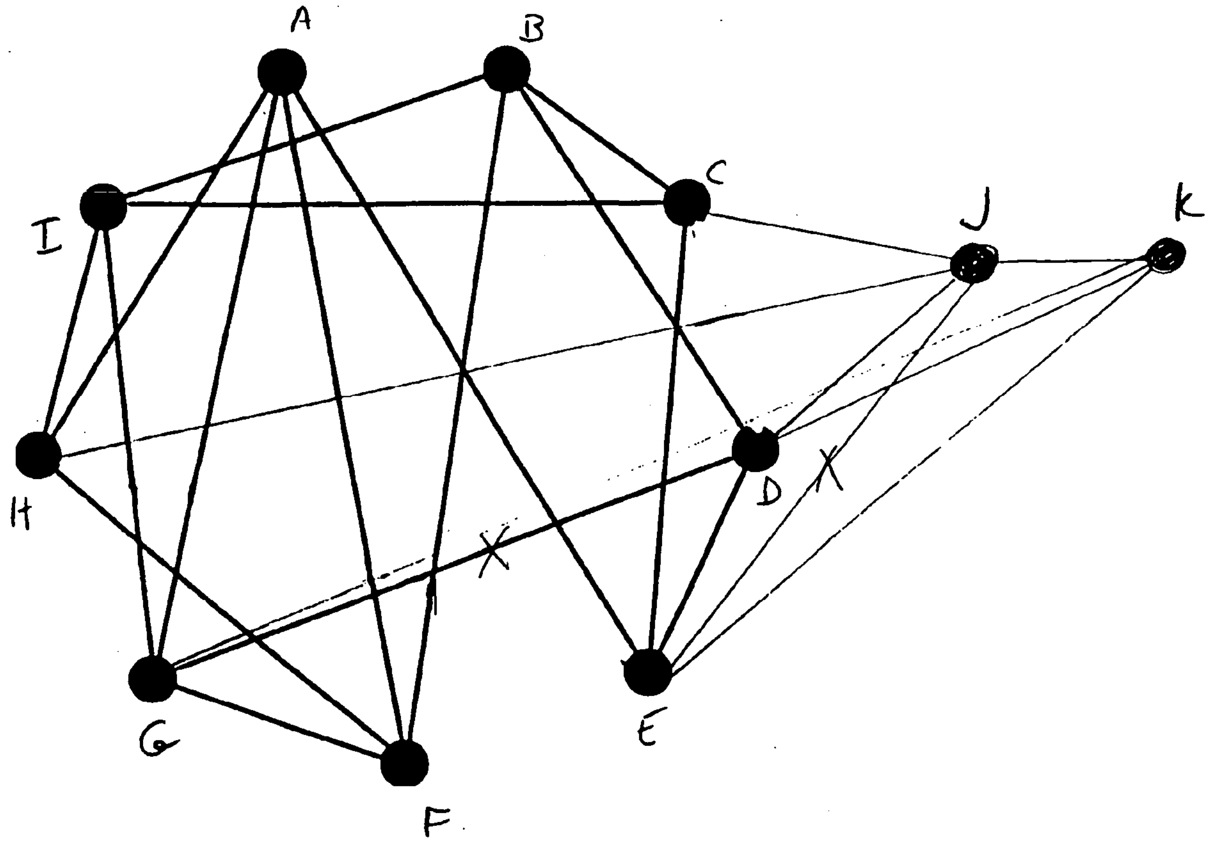

A
B
C
J
K
I
H
D
X
X
G
E
F

Fig 4C

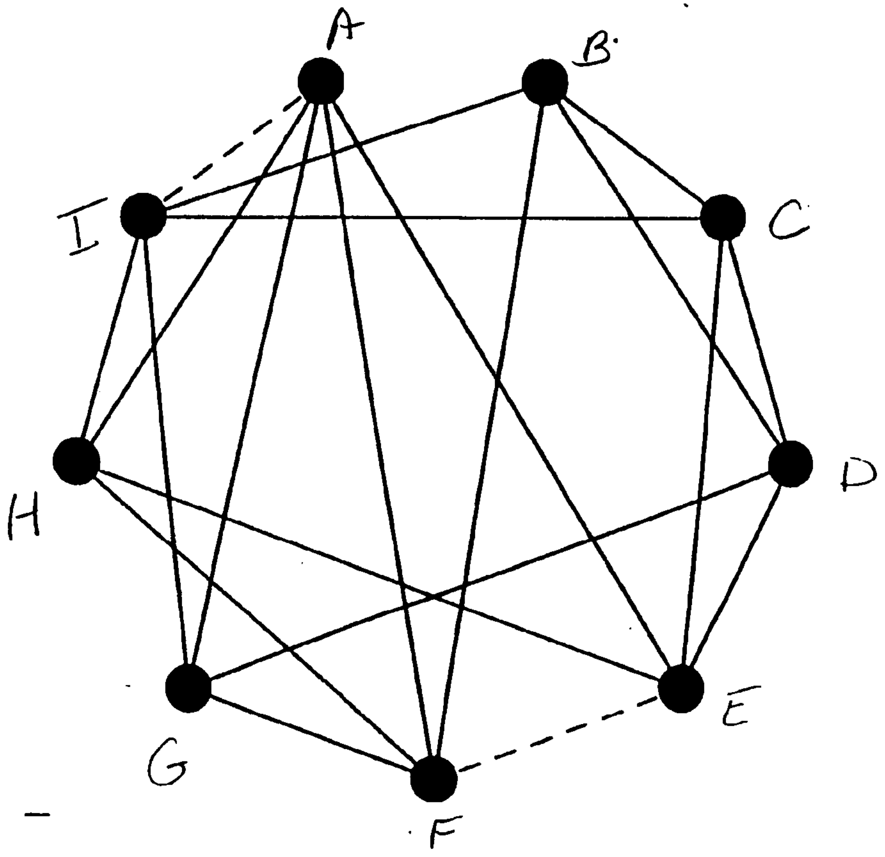


Fig 5A

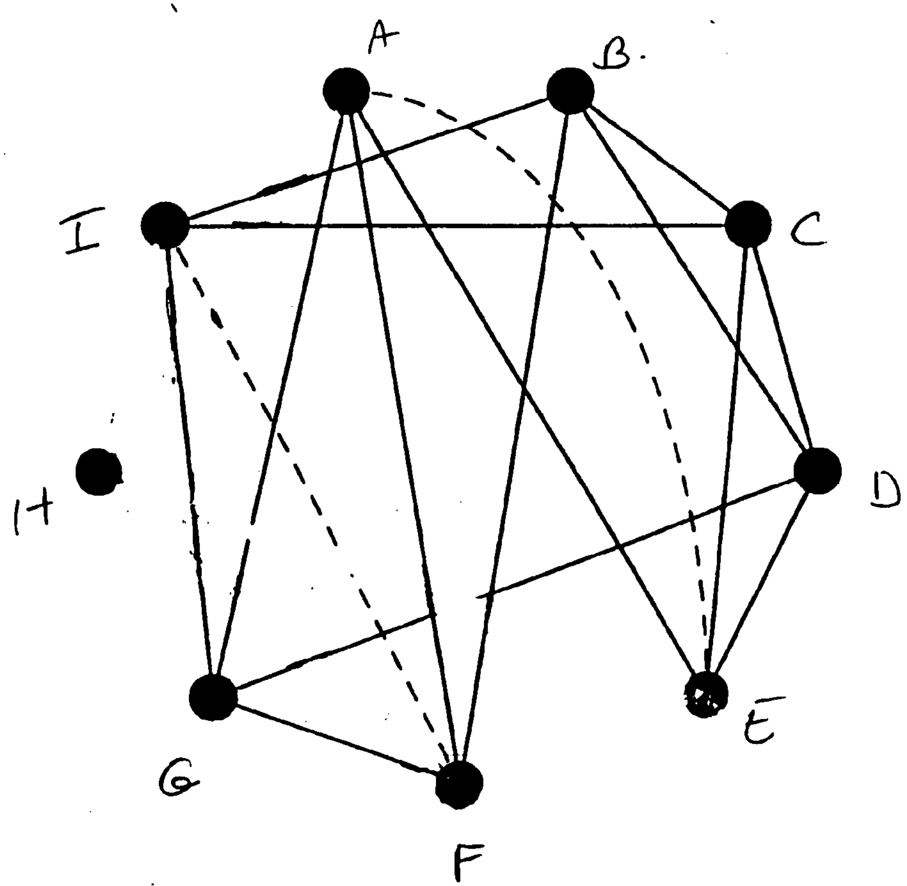


Fig 5B

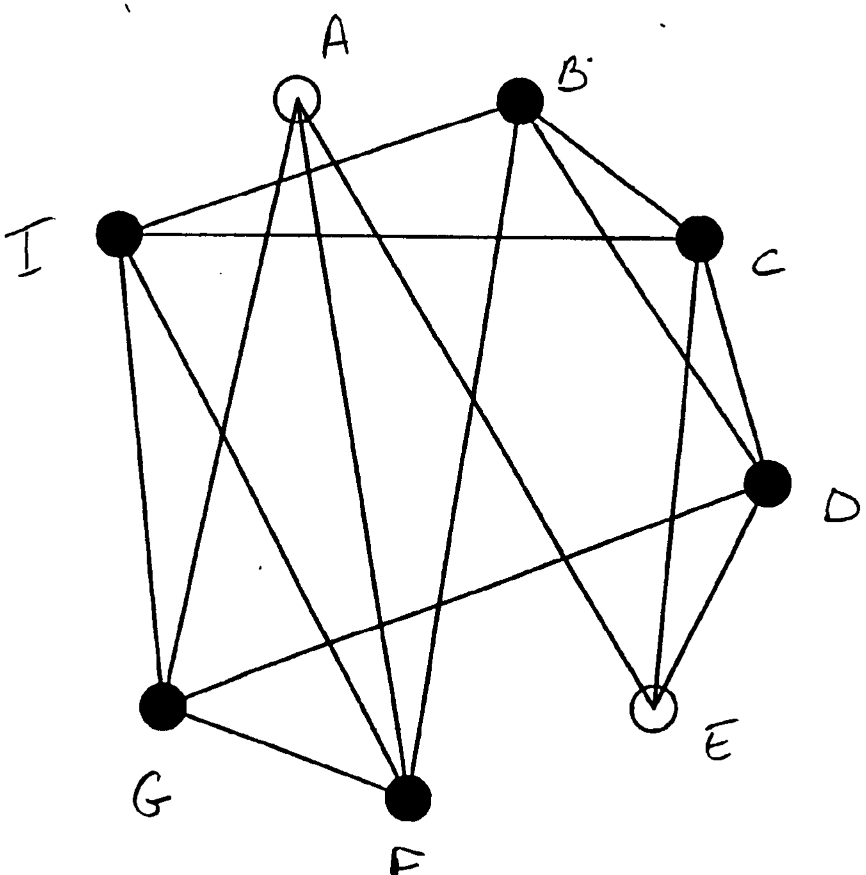


Fig 5C

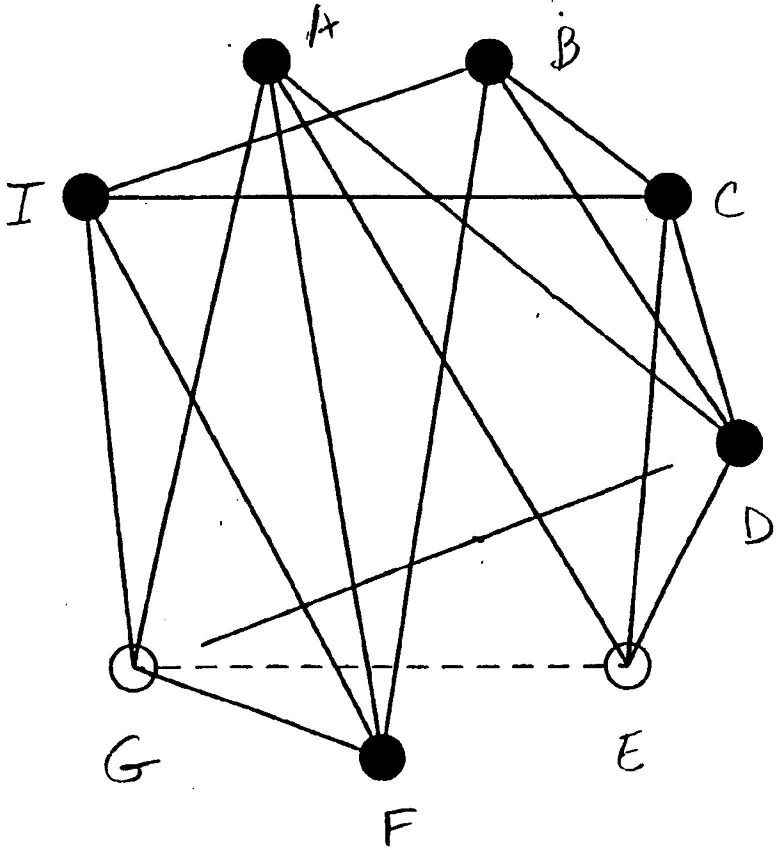


Fig 5D

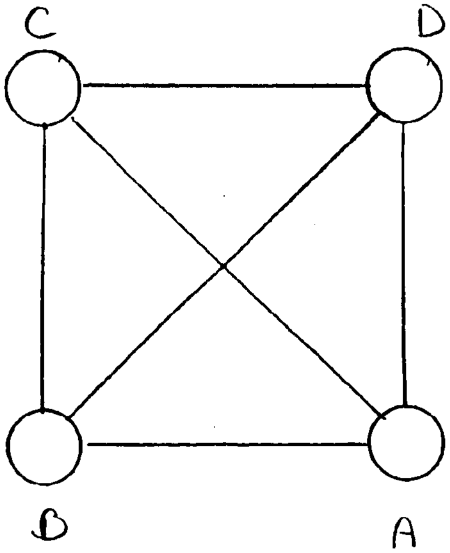


Fig 5E

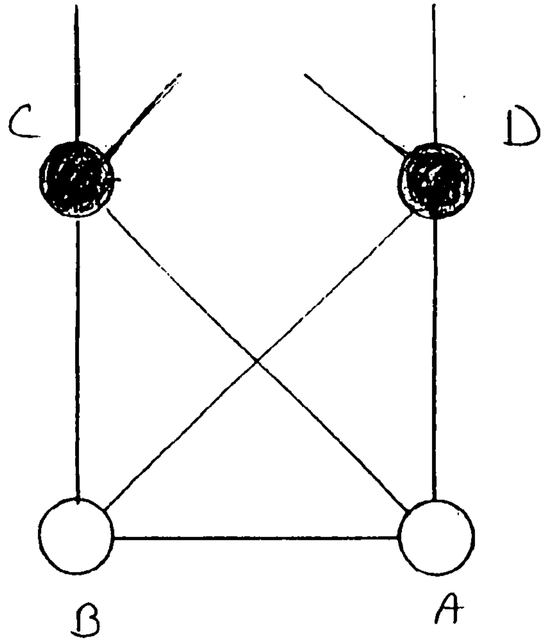


Fig 5F

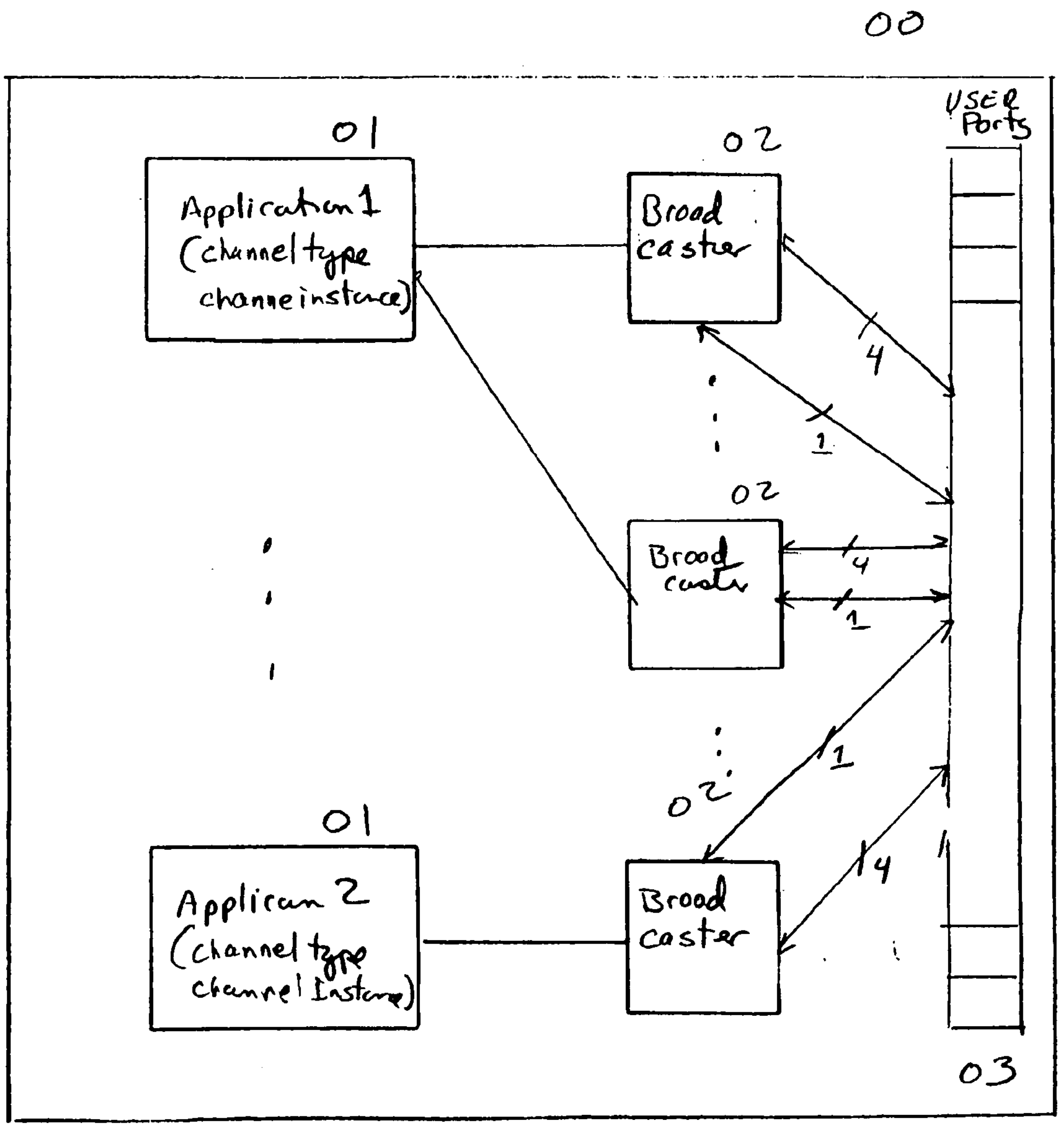

USER Ports
Application 1 (channel type channe instance)
Broad caster
Broad caster
Broad caster
Applican 2 (channel type channel Instance)

Fig 6

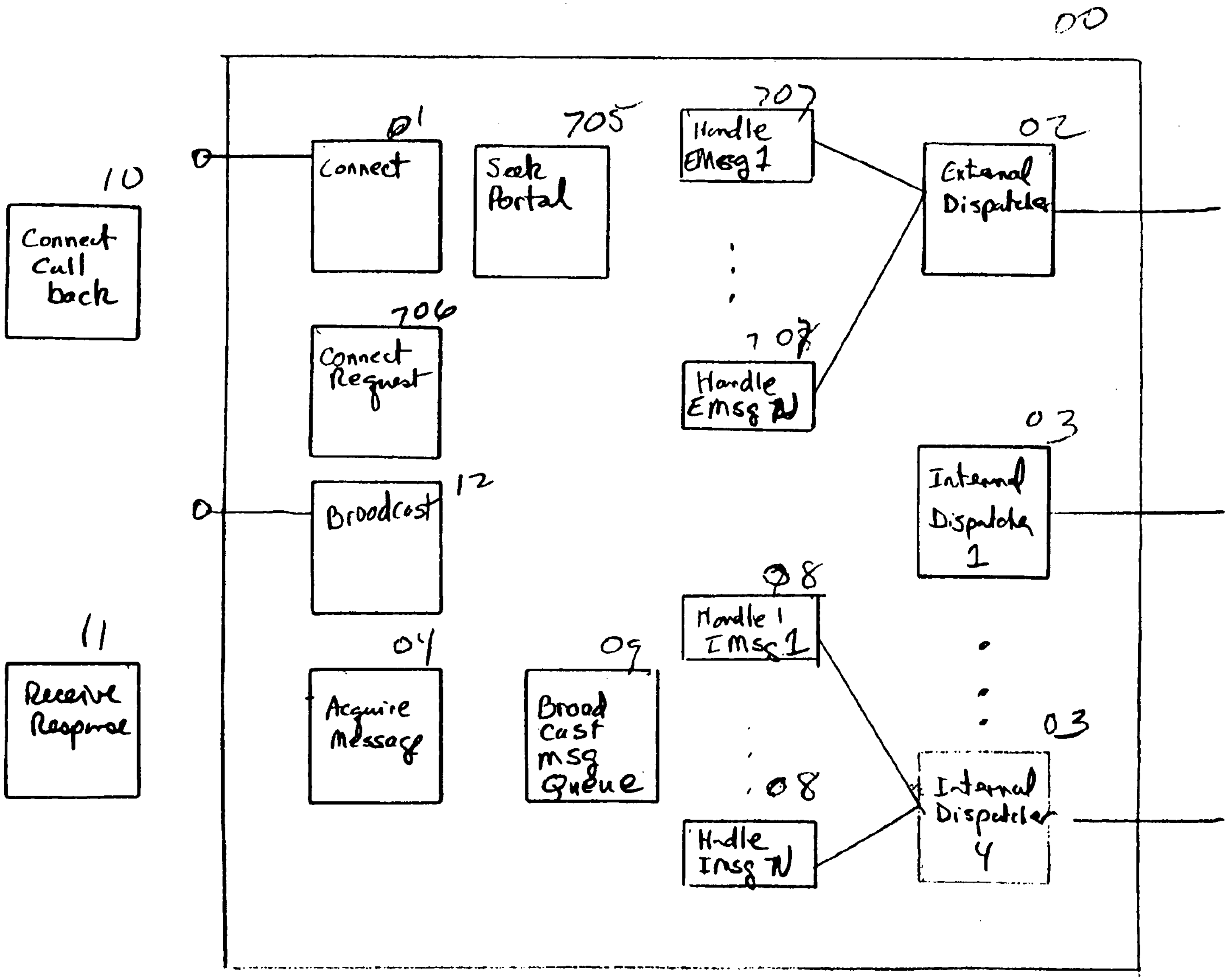

00
10
Connect Call back
11
Receive Response
01
Connect
705
Seek Portal
706
Connect Request
12
Broadcost
04
Acquire Message
09
Broad cast msg Queue
707
Handle EMsg 1
707
Handle EMsg N
08
Handle IMsg 1
08
Handle IMsg N
02
External Dispatcher
03
Internal Dispatcher 1
03
Internal Dispatcher 4

Figure 7

p2

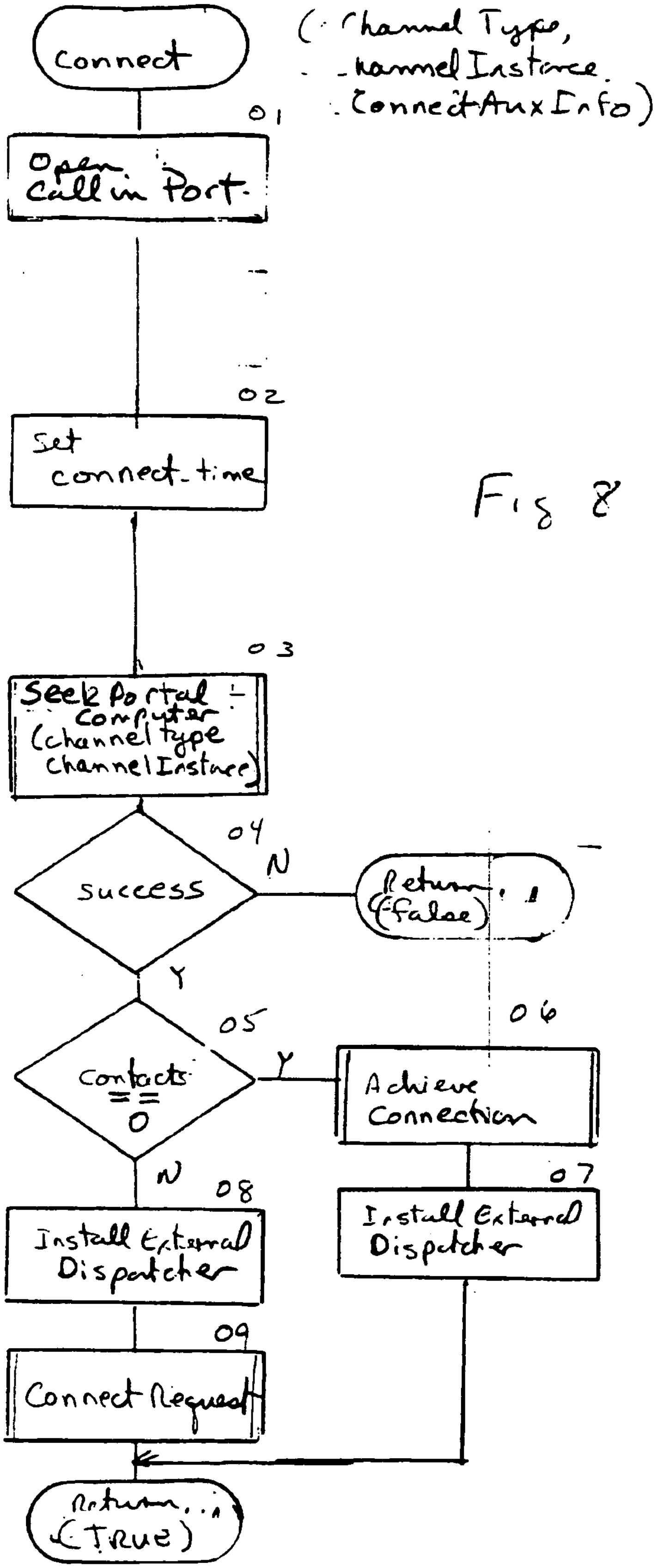

connect
(ChannelType,
ChannelInstance,
ConnectAuxInfo)
01
Open Call in Port.
02
Set connect_time
03
Seek Portal Computer (channel type channelInstance)
04
success
N
Return (false)
Y
05
Contacts == 0
Y
06
Achieve Connection
07
Install External Dispatcher
N
08
Install External Dispatcher
09
Connect Request
Return (TRUE)

Fig 8

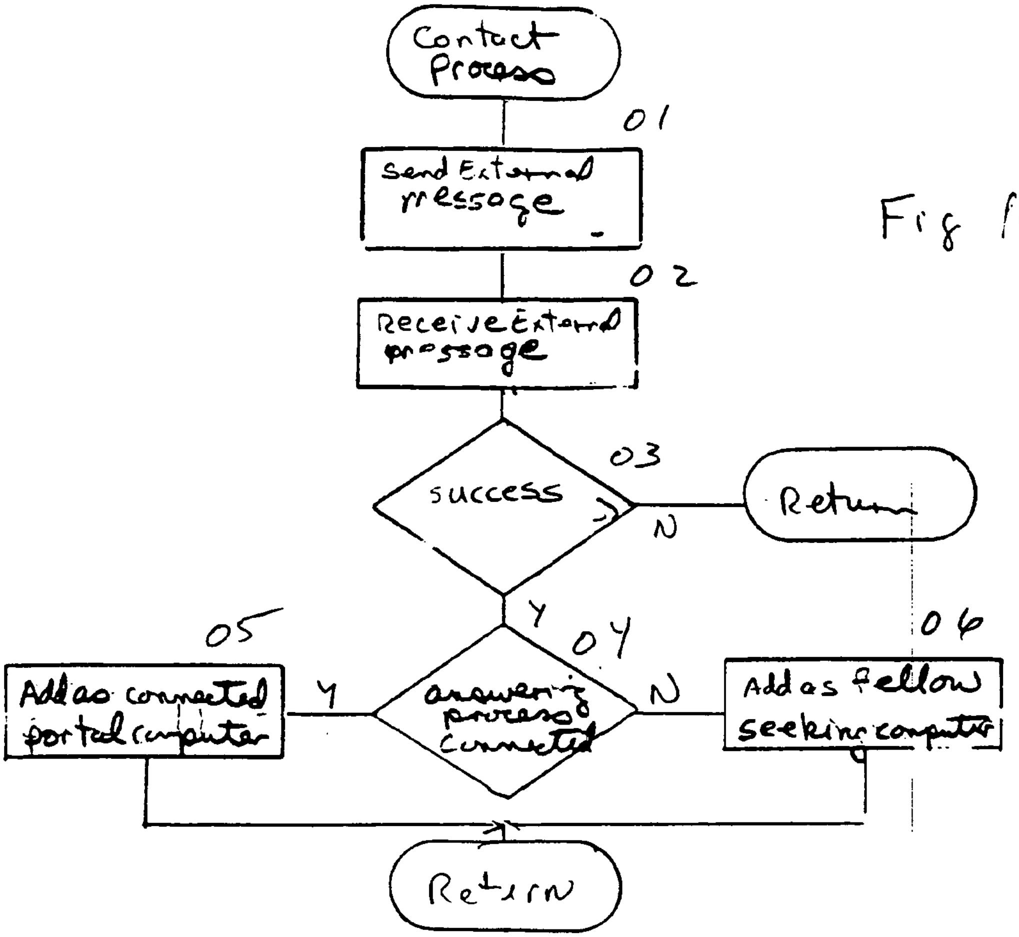
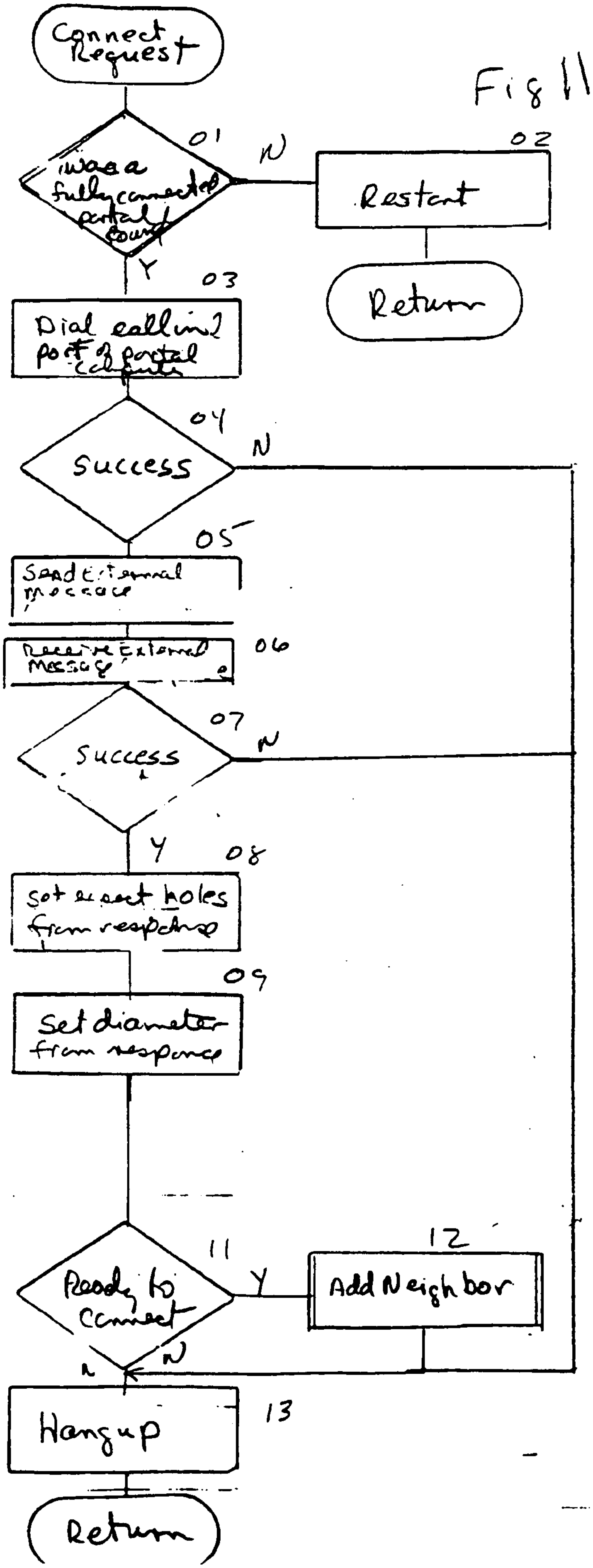

Contact Process
01
Send External message
02
Receive External message
03
success
N
Return
Y
04
answering process connected
05
Y
Add as connected ported computer
N
06
Add as fellow seeking computer
Return

Fig 10

p23

Fig 11


Connect Request
01
was a fully connected portal court
N
02
Restart
Return
Y
03
Dial calling port of portal computer
04
success
N
05
Send External message
Receive External Message
06
07
success
N
Y
08
set exact holes from response
09
Set diameter from response
11
Ready to Connect
Y
12
Add Neighbor
N
Hangup
13
Return

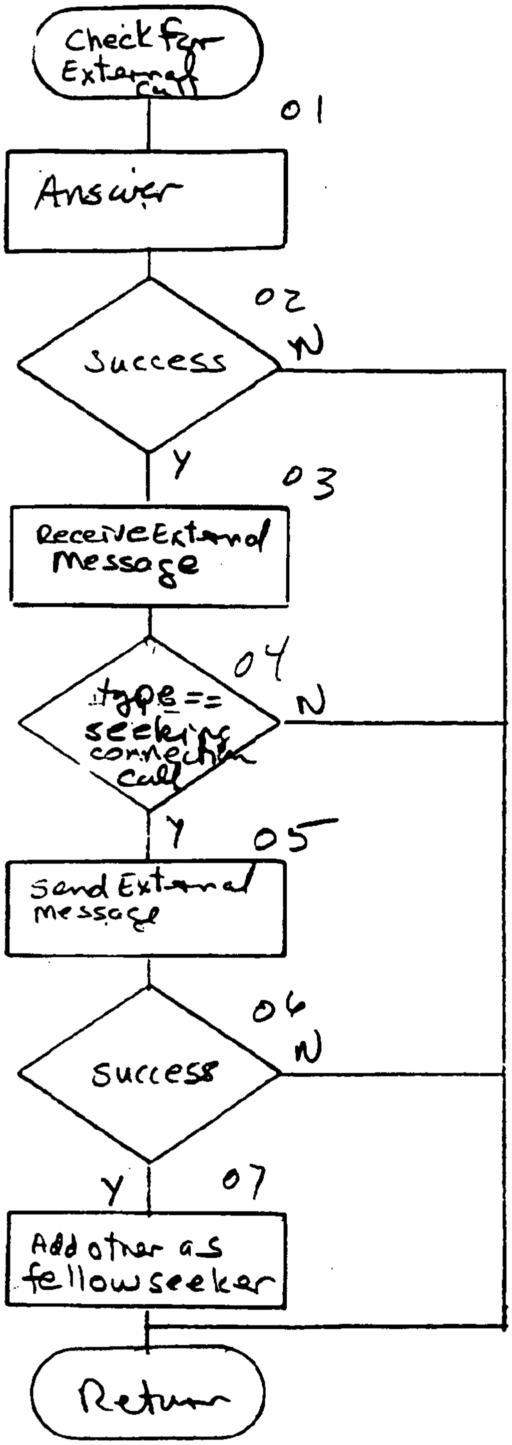

Check for External call
01
Answer
02
Success
N
Y
03
Receive External message
04
type == seeking connection call
N
Y
05
Send External message
06
N
Success
Y
07
Add other as fellow seeker
Return

Fig 12

Achieve
Nirvana.
22

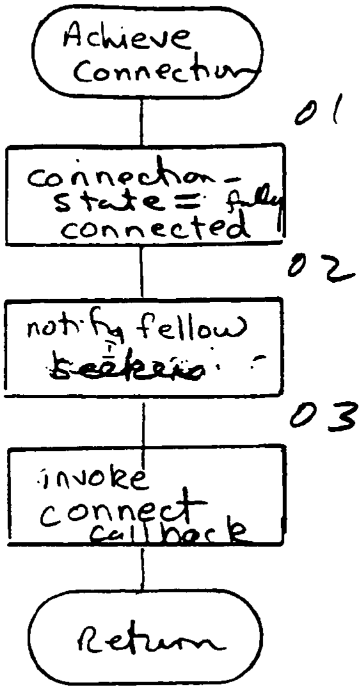
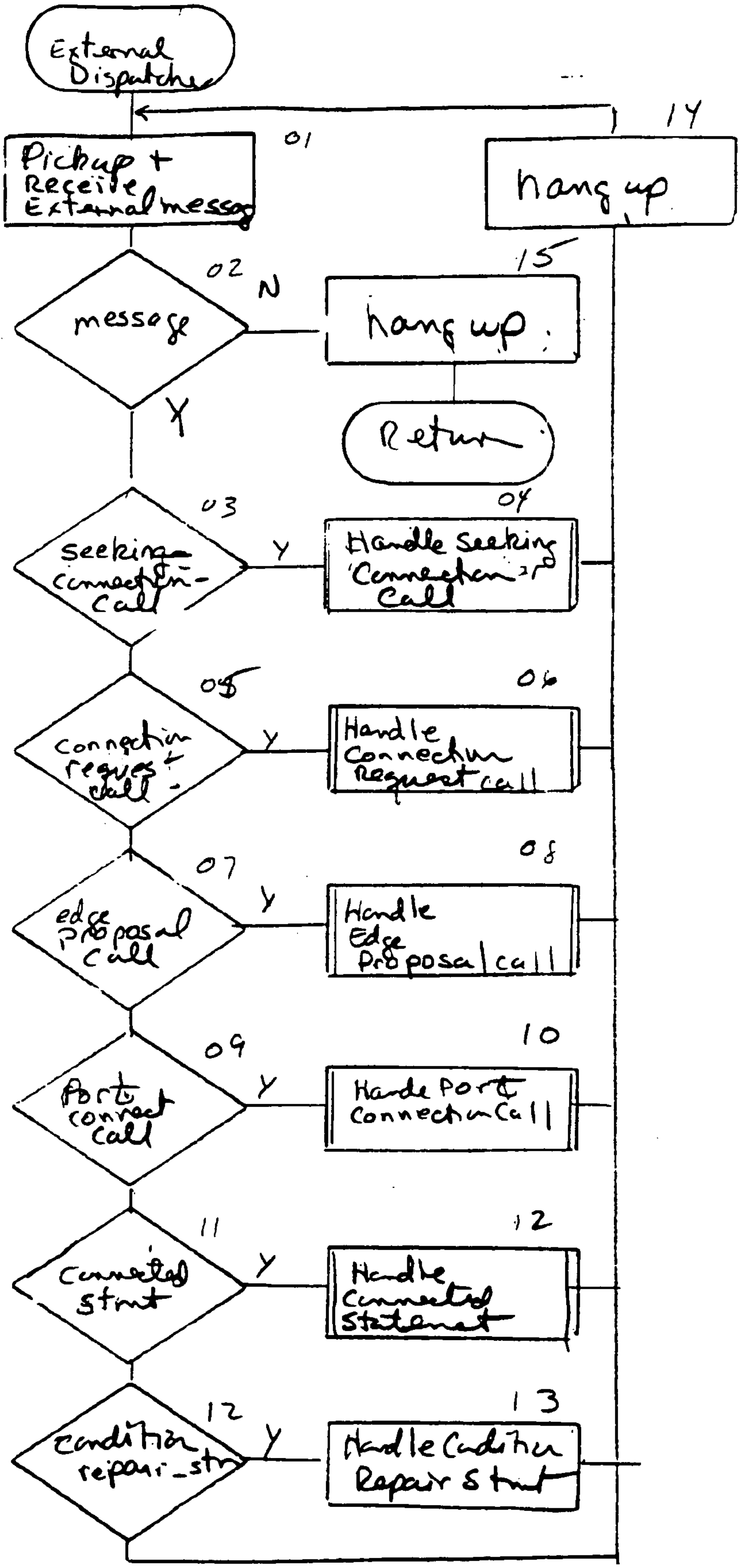

Achieve Connection
01
connection-state = fully connected
02
notify fellow
~~seekers~~
03
invoke connect callback
Return

Fig 13

p7

Fig 14


External Dispatcher
Pickup + Receive External message
01
message
02
N
hang up
15
Return
Y
14
hang up
seeking connection call
03
Y
Handle Seeking Connection Call
04
connection request call
05
Y
Handle Connection Request call
06
edge proposal call
07
Y
Handle Edge Proposal call
08
Port connect call
09
Y
Handle Port Connection Call
10
Connected stmt
11
Y
Handle Connected Statement
12
condition repair_stmt
12
Y
Handle Condition Repair Stmt
13

P 7

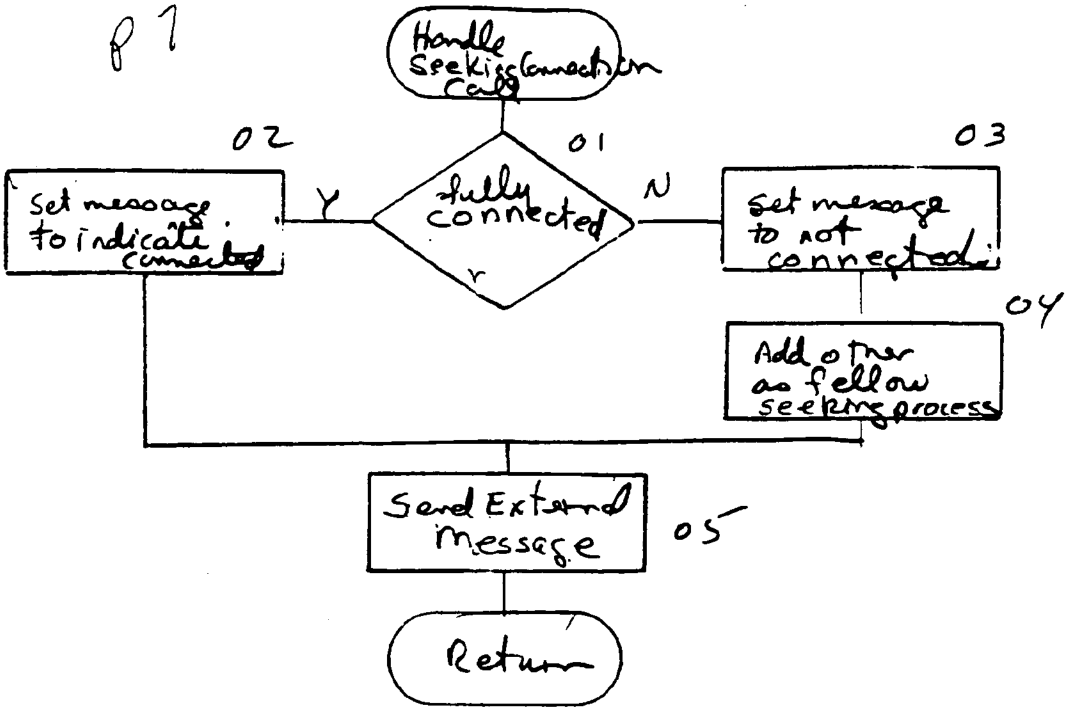
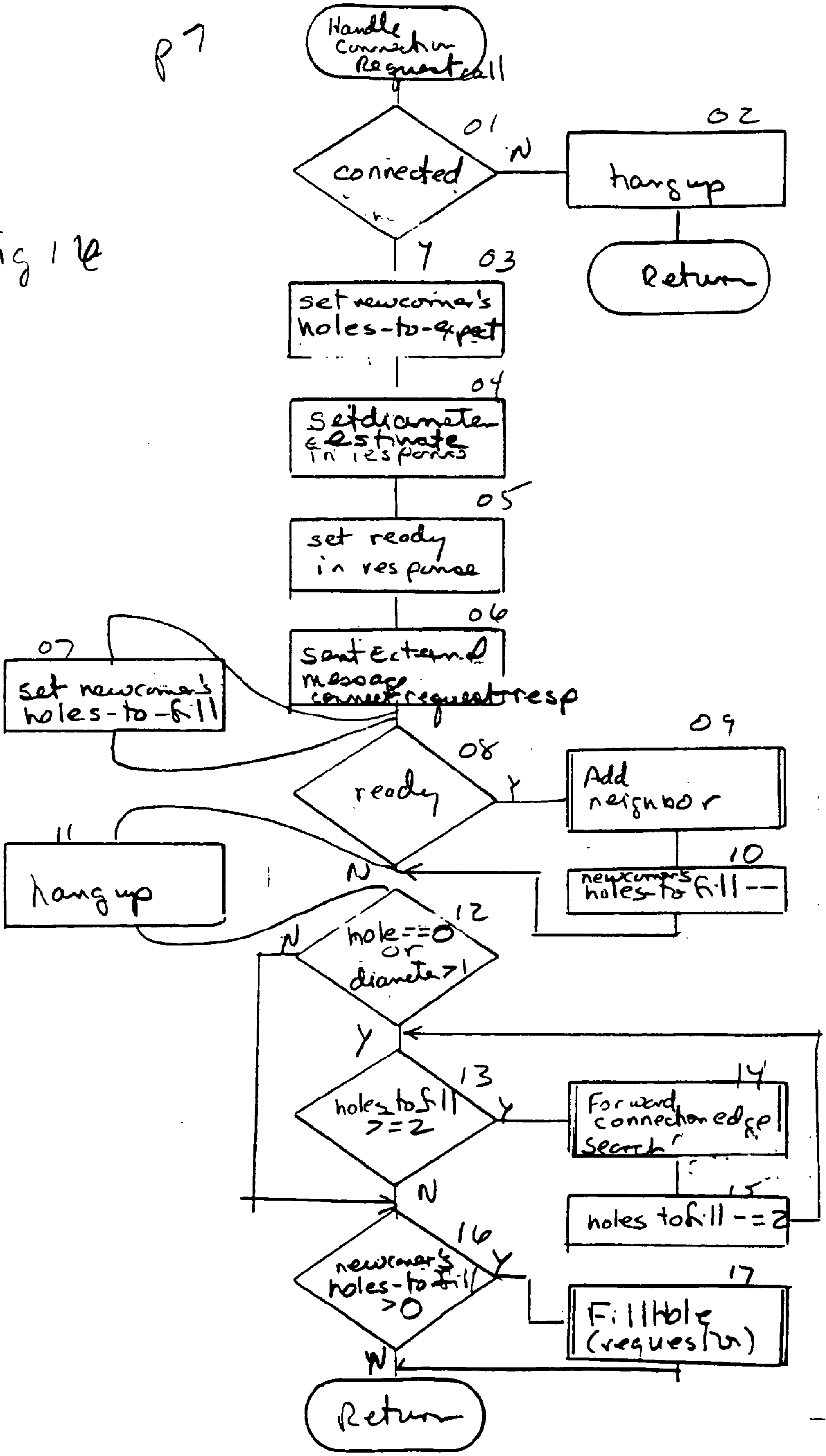

Handle Seeking Connection Call
01
fully connected
Y
N
02
Set message to indicate connected
03
set message to not connected
04
Add other as fellow seeking process
Send External Message
05
Return

Fig 15

p7

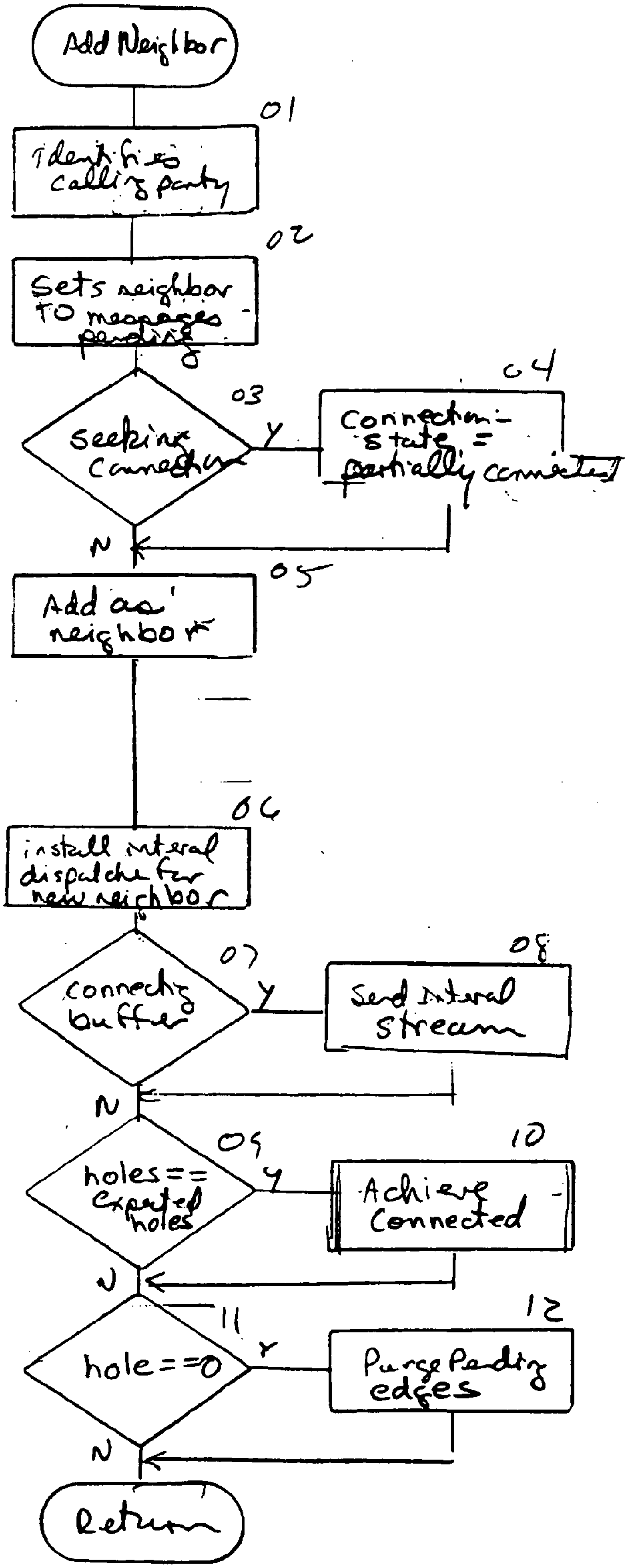
Fig 1e



Handle Connection Request call
01
connected
N
02
hangup
Return
Y
03
set newcomer's holes-to-expect
04
set diameter estimate in response
05
set ready in response
06
send External message connect request resp
07
set newcomer's holes-to-fill
08
ready
Y
09
Add neighbor
10
newcomer's holes-to-fill --
11
hangup
N
12
hole==0 or diameter>1
N
Y
13
holes to fill >=2
Y
14
Forward connection edge search
15
holes to fill -=2
N
16
newcomer's holes-to-fill >0
Y
17
FillHole (requester)
N
Return

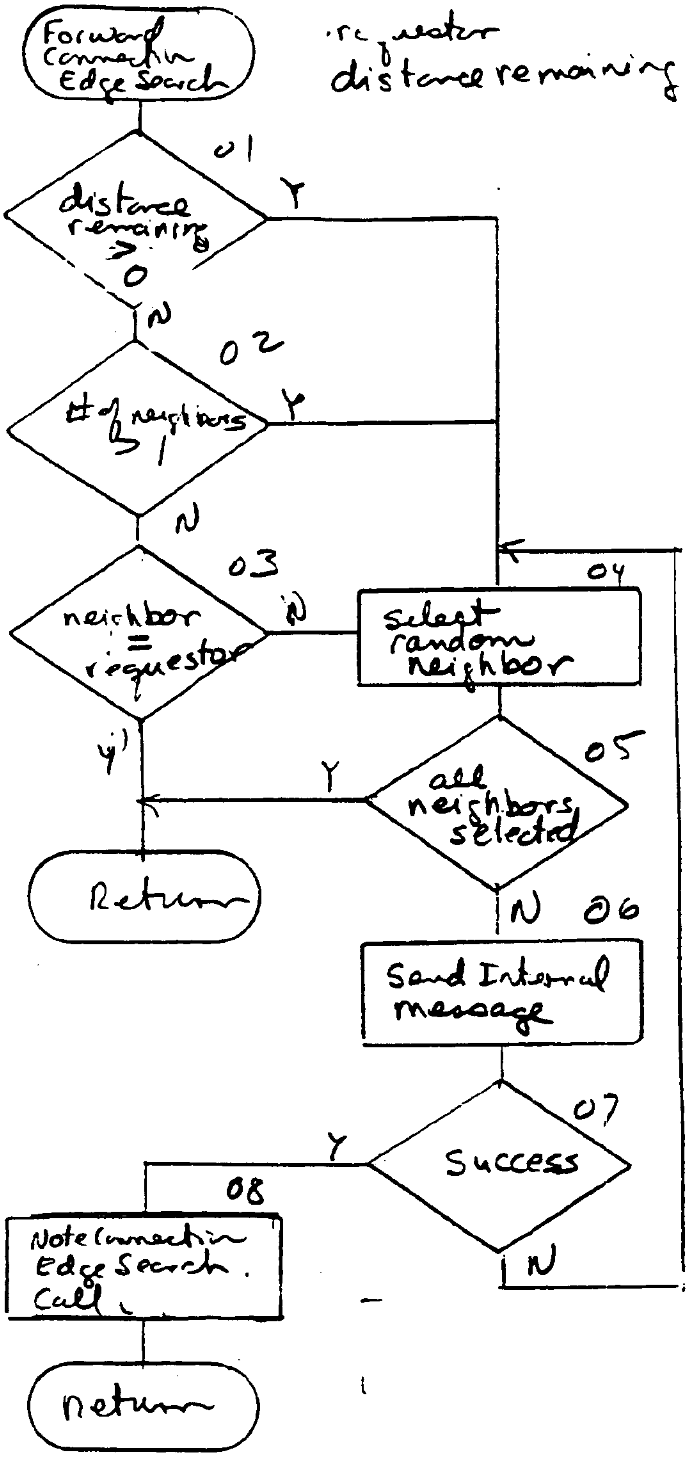
p9

Fig17

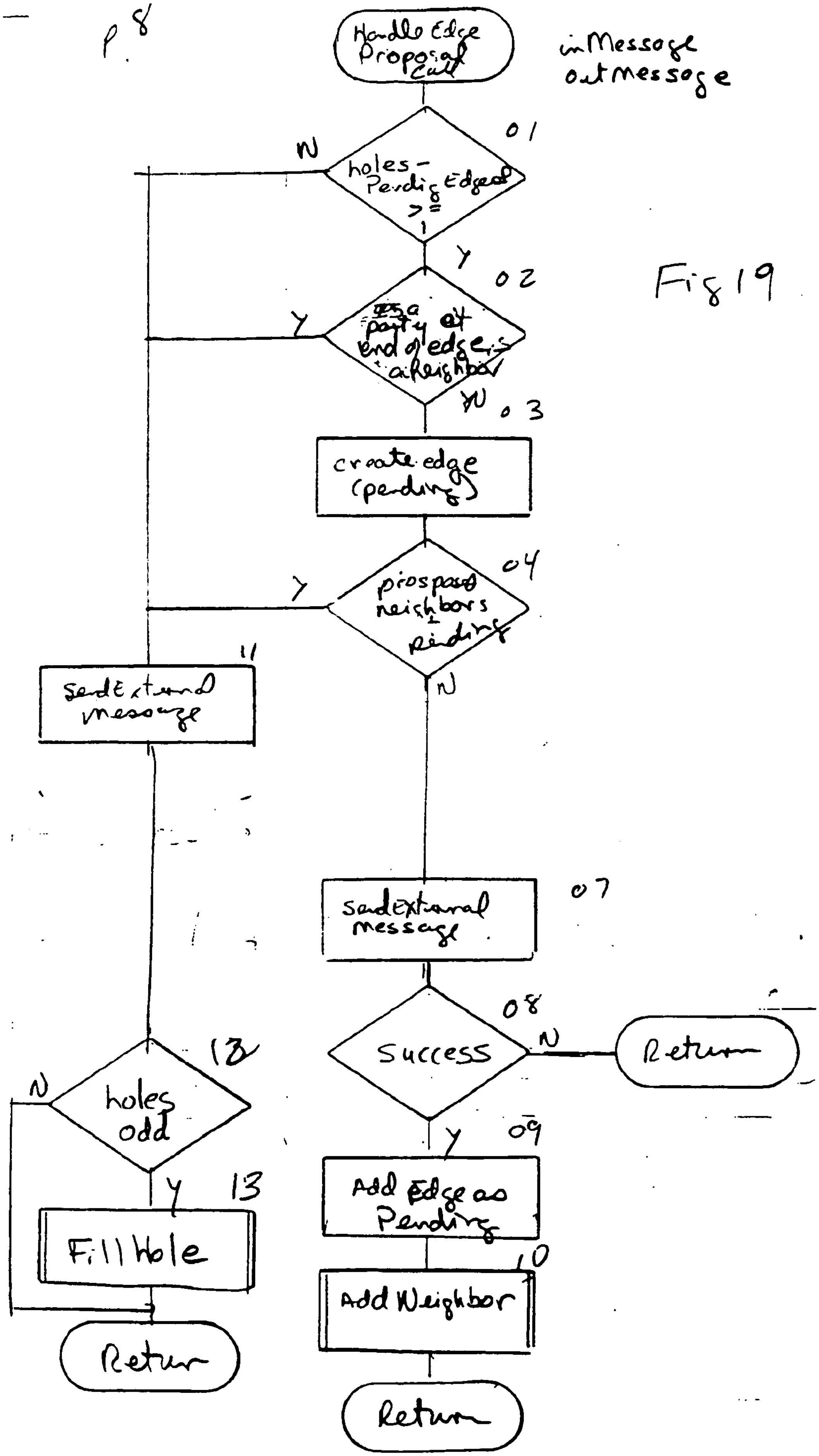

Add Neighbor
01
identifies calling party
02
Sets neighbor TO messages pending
03
seeking Connection
Y
04
Connection-state = partially connected
N
05
Add as neighbor
06
install internal dispatche for new neighbor
07
connecting buffer
Y
08
Send Internal stream
N
09
holes == expected holes
Y
10
Achieve Connected
N
11
hole==0
Y
12
Purge Pending edges
N
Return

15

Fig 18


requester
distance remaining
Forward Connection Edge Search
01
distance remaining > 0
Y
N
02
# of neighbors > 1
Y
N
03
neighbor = requestor
N
Y
04
select random neighbor
05
all neighbors selected
Y
N
06
Send Internal message
07
Success
Y
N
08
Note Connection Edge Search Call
Return
return

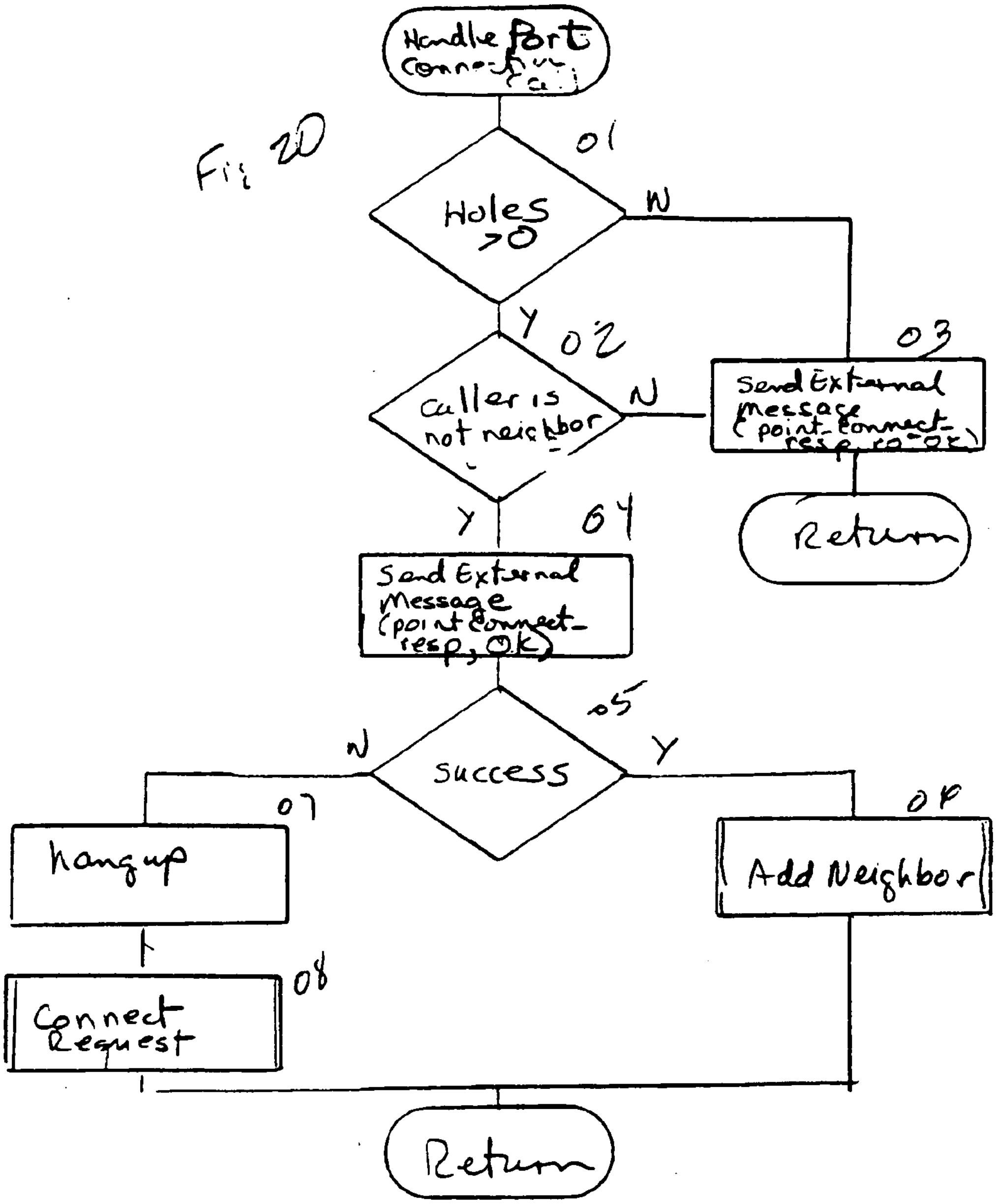
p 8



Handle Edge Proposal Call
inMessage
outMessage
01
holes-PendingEdges >= 1
N
Y
02
~~is a~~ party at end of edge is a Neighbor
Y
N
03
create edge (pending)
04
prospect neighbors pending
Y
N
11
sendExternal message
07
sendExternal message
08
Success
N
Return
Y
09
Add ~~Edge~~ Edge as Pending
10
Add Neighbor
Return
12
holes odd
N
Y
13
Fill Hole
Return

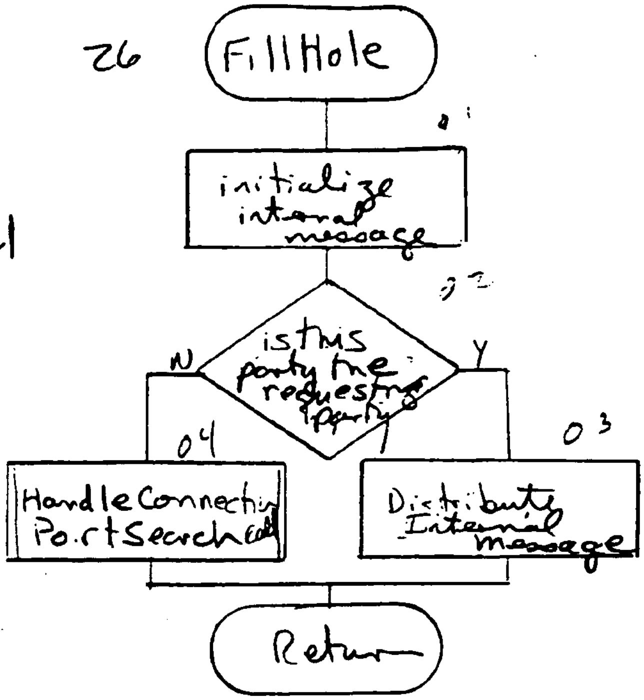
Fig 19

Fig 2D



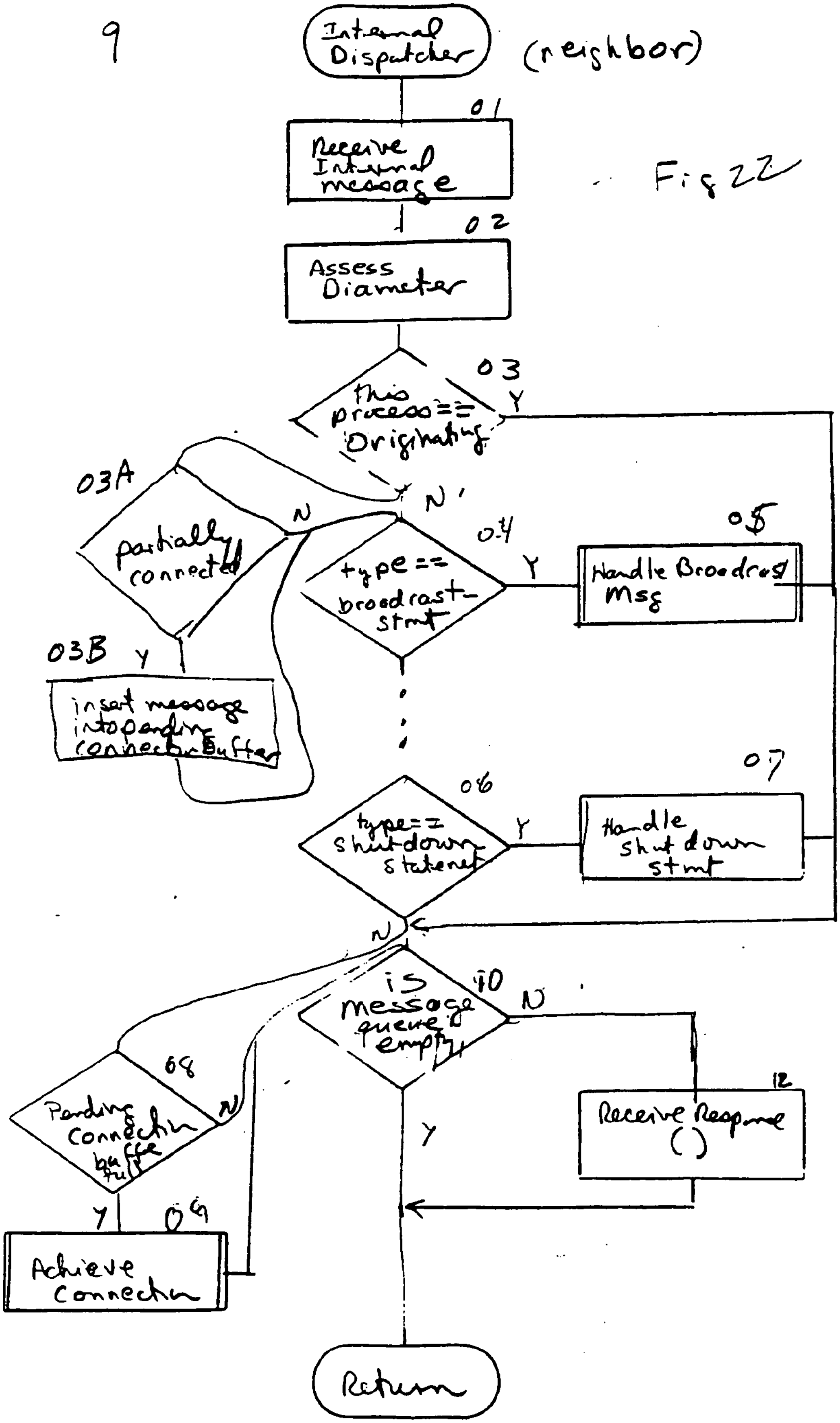
Handle Port Connection Call
01
Holes >0
N
Y
02
caller is not neighbor
N
03
Send External message (point_connect_resp, no-OK)
Return
Y
04
Send External Message (point connect_resp, OK)
05
success
N
Y
07
hangup
08
Connect Request
0P
Add Neighbor
Return

Fig 21


26
FillHole
initialize internal message
02
is this party the requesting party
N
Y
04
03
Handle Connection Port Search call
Distribute Internal message
Return

9

Fig 22


Internal Dispatcher
(neighbor)
01
Receive Internal message
02
Assess Diameter
03
this process == Originating
Y
N
03A
partially connected
N
Y
03B
insert message into pending connectionBuffer
04
type == broadcast_stmt
Y
05
Handle Broadcast Msg
06
type == shutdown statement
Y
07
Handle shutdown stmt
N
10
is message queue empty
N
Y
12
Receive Response ( )
08
Pending connection buffer full
N
Y
09
Achieve Connection
Return

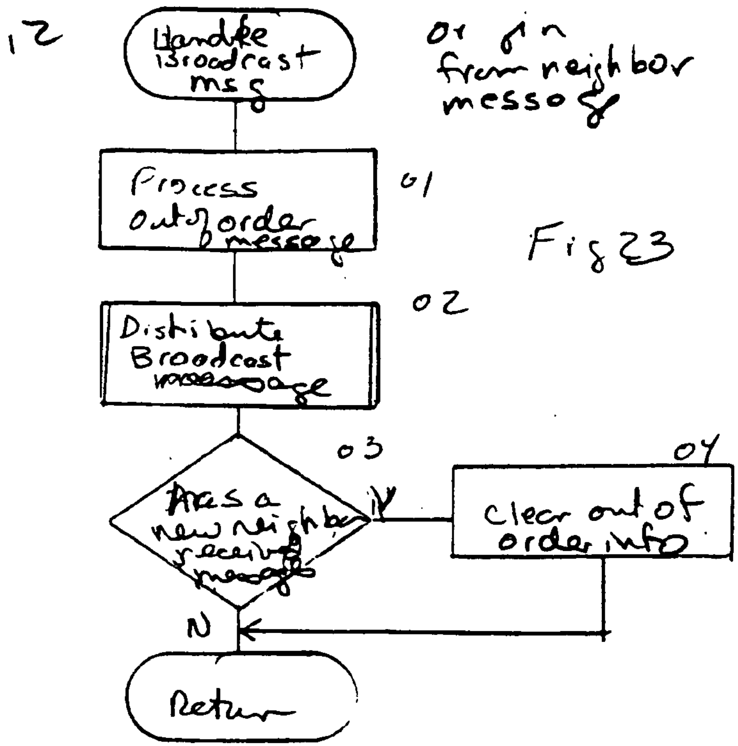
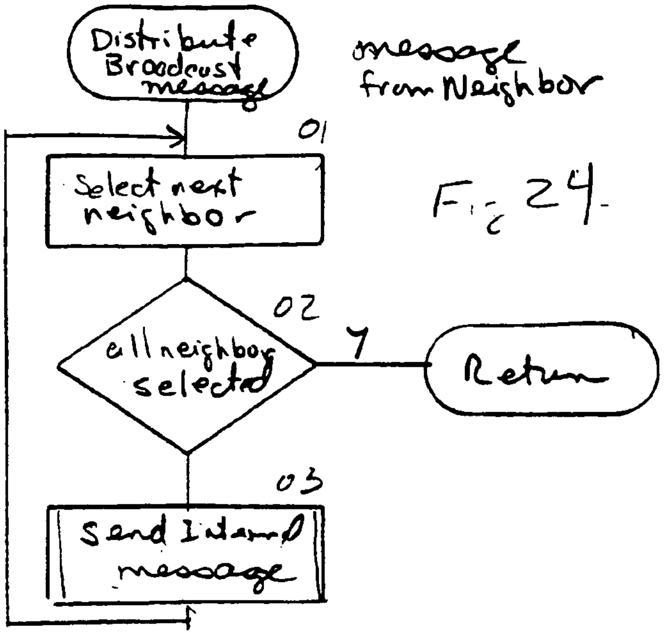

12
Handle Broadcast msg
or gin from neighbor messag
01
Process out of order message
02
Distribute Broadcast message
03
Has a new neighbor receive message
Y
04
Clear out of order info
N
Return

Fig 23

20

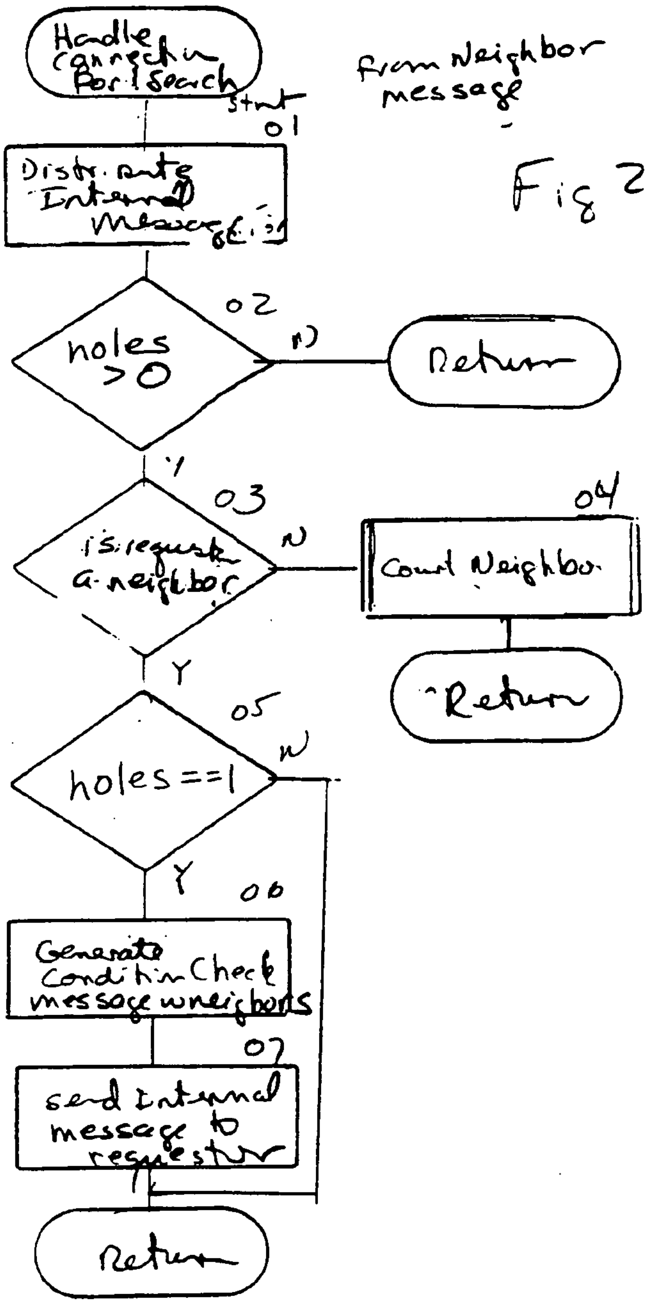


Distribute Broadcast message
01
Select next neighbor
02
all neighbor selected
Y
Return
03
Send Internal message

message from Neighbor

Fig 24.

13


Handle connection Port Search
start
01
Distribute Internal Message (E.3)
02
holes > 0
N
Return
Y
03
is requester a neighbor
N
04
Count Neighbor
Return
Y
05
holes == 1
N
Y
06
Generate Condition Check message w neighbors
07
send Internal message to requester
Return

from Neighbor message

Fig 26

21

21

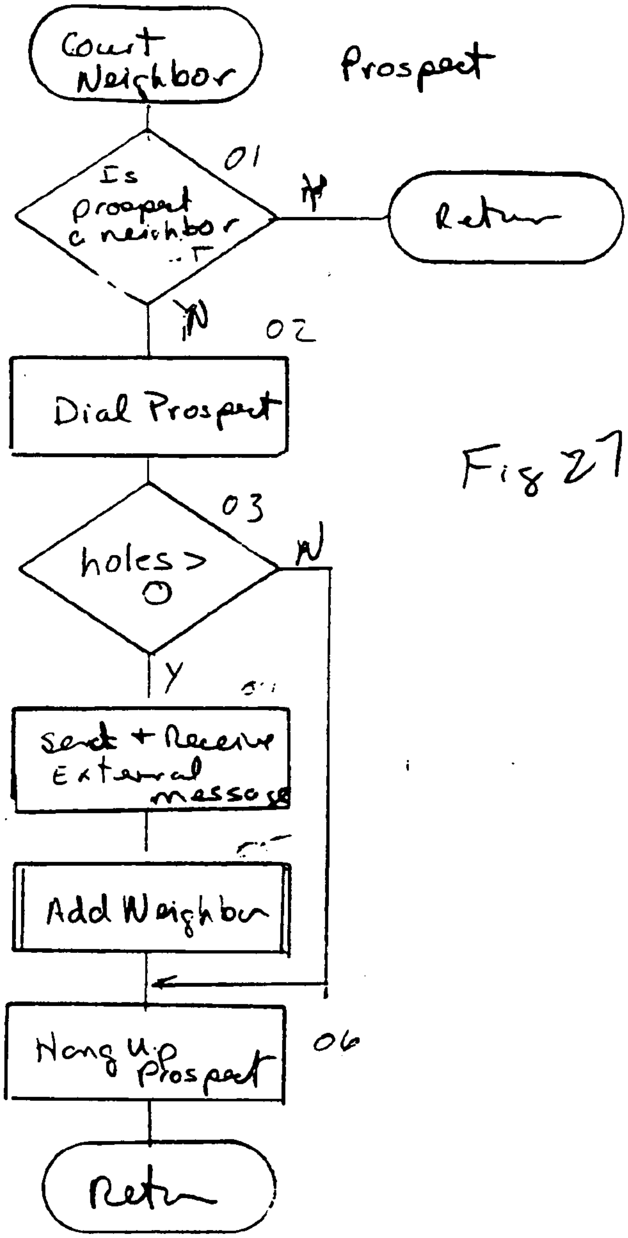
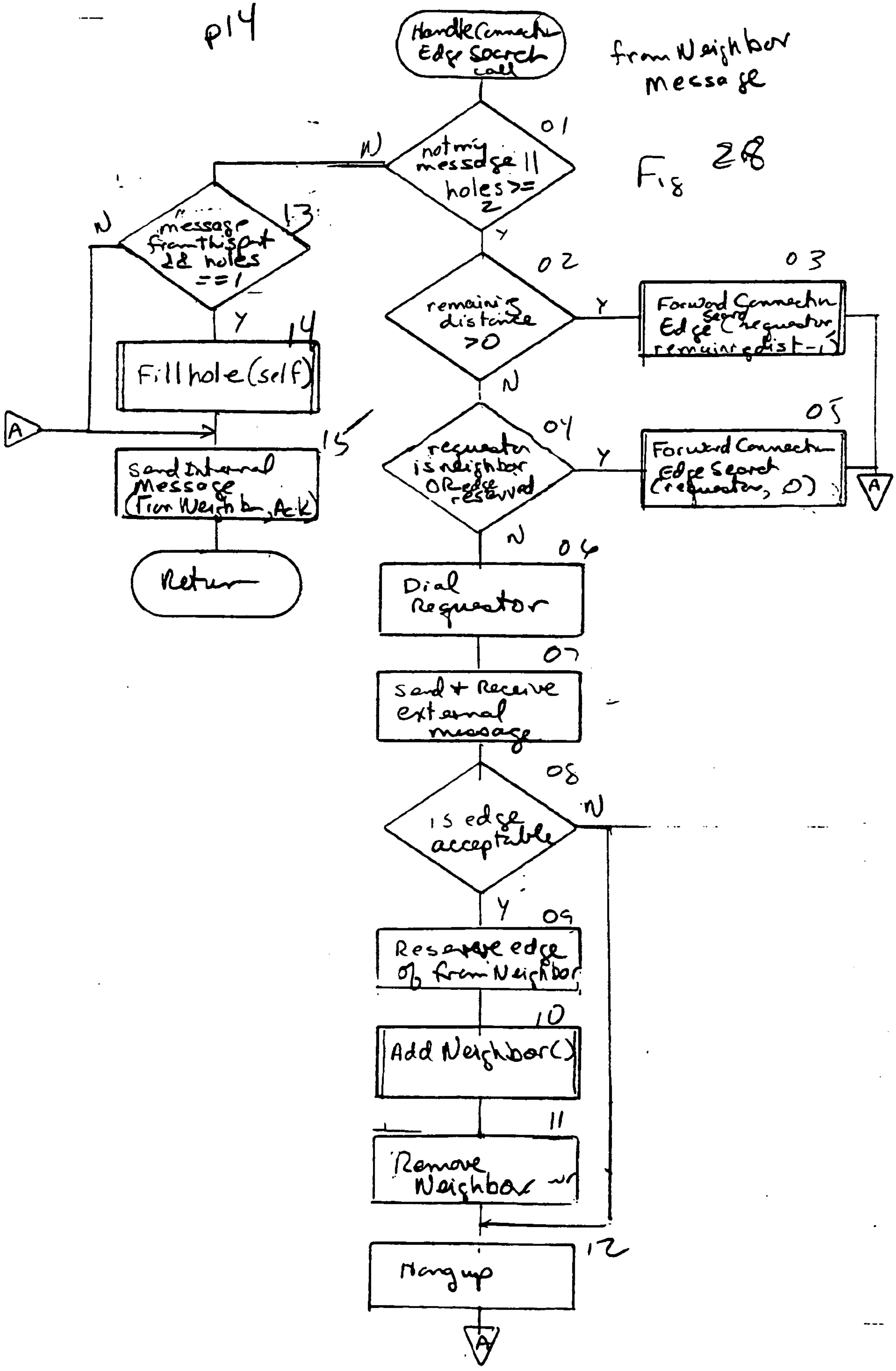

Count Neighbor
Prospect
01
Is prospect a neighbor
Y
Return
N
02
Dial Prospect
03
holes > 0
N
Y
Send + Receive External message
Add Neighbor
Hang up Prospect
06
Return

Fig 27

p14

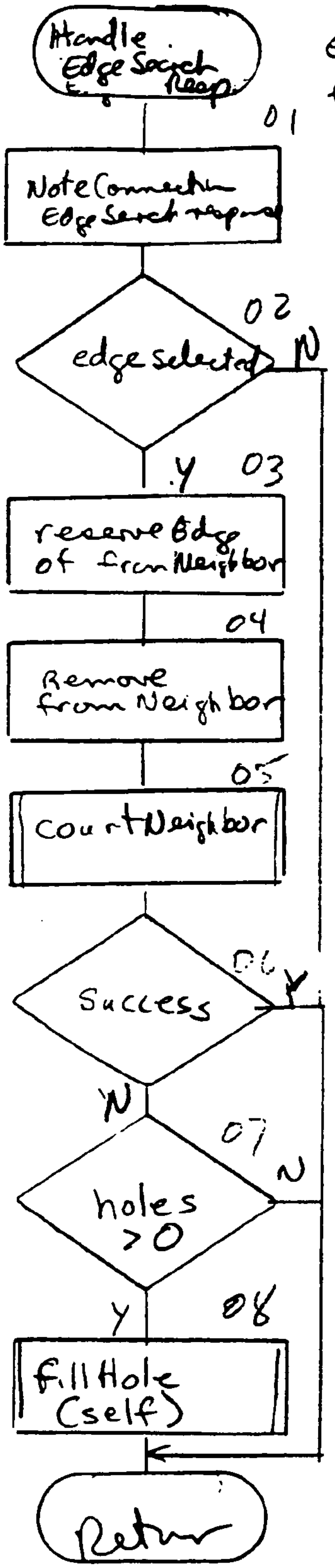
from Neighbor message

Fig 28



Handle Connection Edge Search call
01
not my message || holes >= 2
N
Y
13
" message from this port && holes == 1
N
Y
14
Fill hole (self)
A
15
Send Internal Message (from Neighbor, Ack)
Return
02
remaining distance > 0
Y
03
Forward Connection Edge Search (requestor, remaining dist - 1)
N
04
requestor is neighbor OR edge reserved
Y
05
Forward Connection Edge Search (requestor, 0)
A
N
06
Dial Requestor
07
Send + Receive external message
08
is edge acceptable
N
Y
09
Reserve edge of from Neighbor
10
Add Neighbor()
11
Remove Neighbor
12
Hang up
A

5


Handle EdgeSearch Resp
01
NoteConnection EdgeSearch response
02
edgeSelected
N
Y
03
reserveEdge of fromNeighbor
04
Remove fromNeighbor
05
countNeighbor
06
Success
Y
N
07
holes >0
N
Y
08
fillHole (self)
Return

origin fromNeighbor message

Fig 29

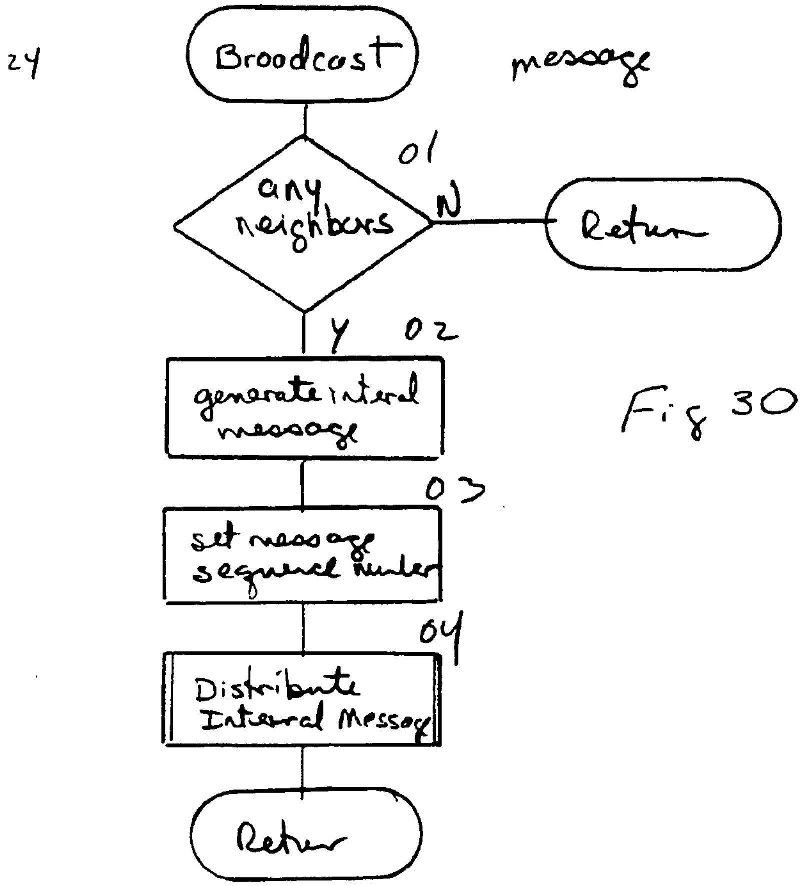

24
Broodcast
message
01
any neighbors
N
Return
Y
02
generate internal message
03
set message sequence number
04
Distribute Internal Message
Return

Fig 30

3

message

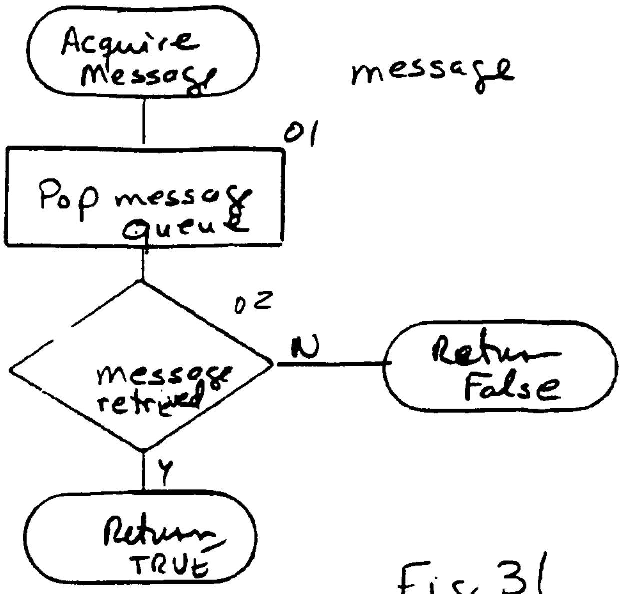
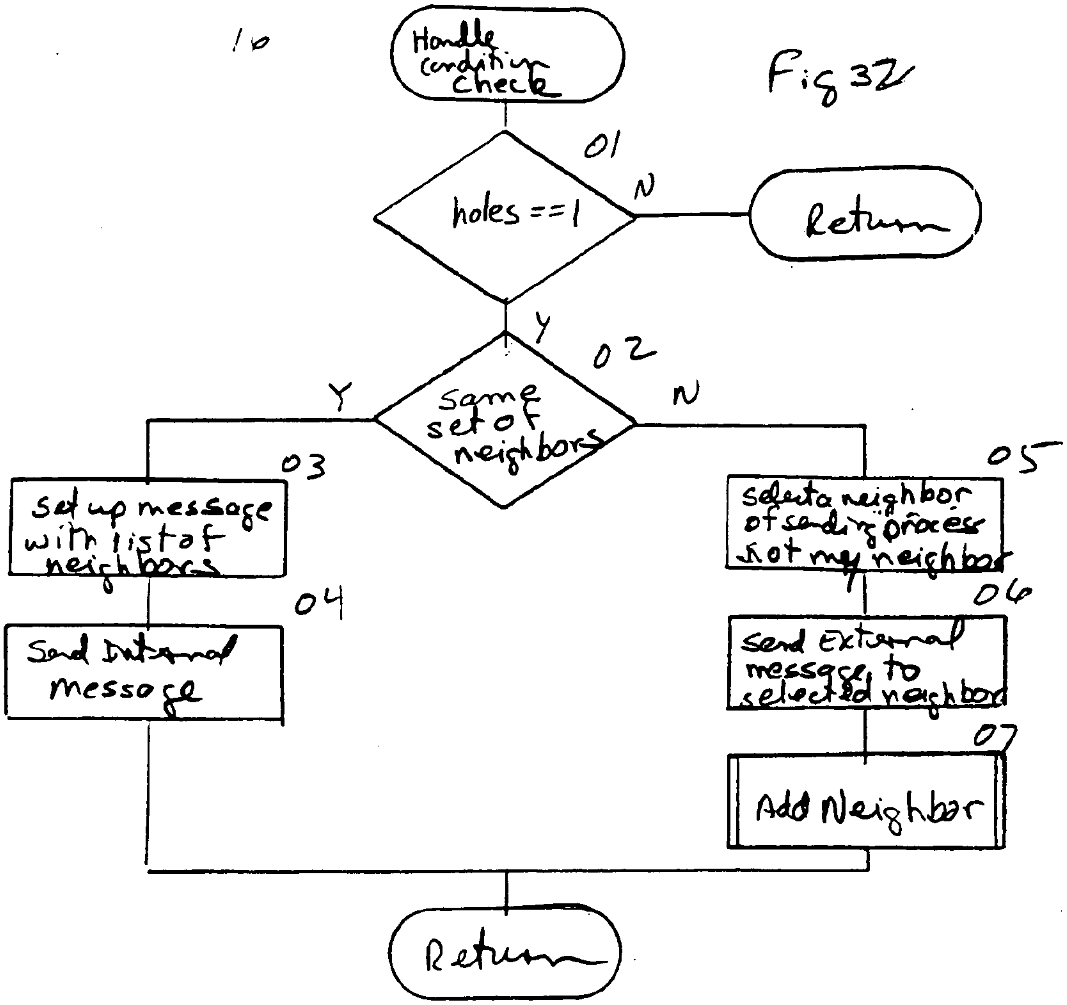

Acquire Message
01
Pop message queue
02
message retrieved
N
Return False
Y
Return TRUE

Fig 31

10



Handle condition Check
01
holes ==1
N
Return
Y
02
Same set of neighbors
Y
N
03
set up message with list of neighbors
04
Send Internal message
05
select a neighbor of sending process not my neighbor
06
send External message to selected neighbor
07
Add Neighbor
Return

Fig 32

23

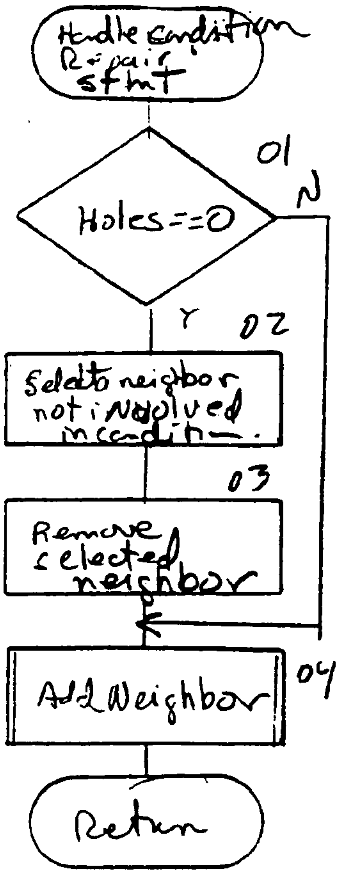

Handle condition Repair sfmt
01
Holes==0
N
Y
02
Select a neighbor not involved in condition.
03
Remove selected neighbor
04
Add Neighbor
Return

Fig 33

17

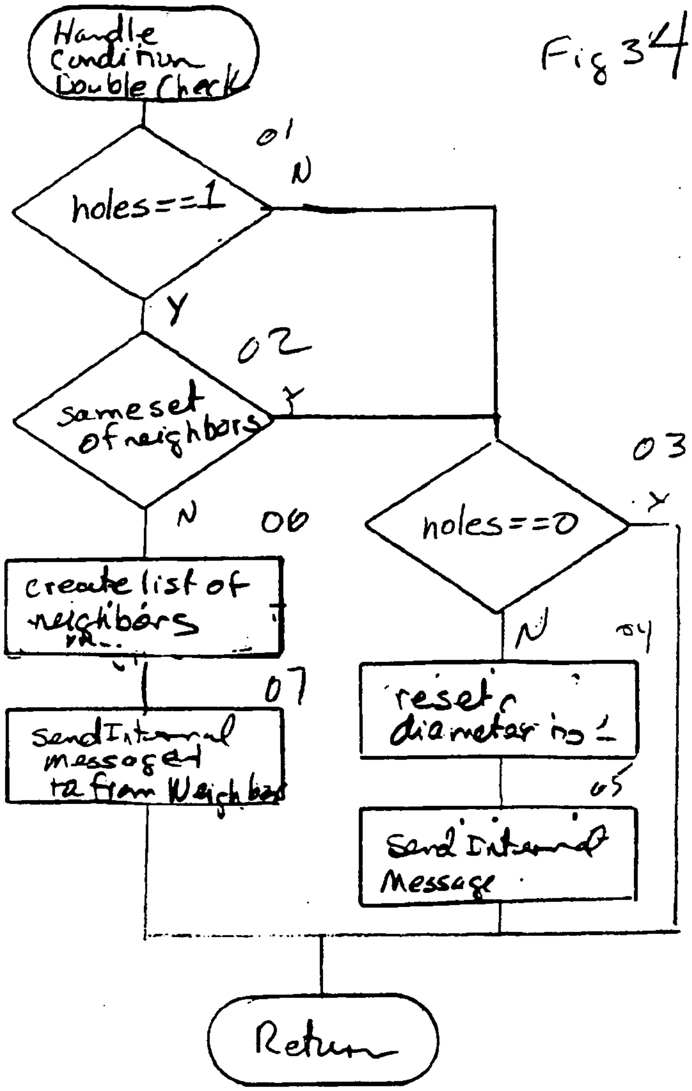

Handle Condition Double Check
01
holes==1
N
Y
02
same set of neighbors
Y
N
03
holes==0
Y
N
06
create list of neighbors
07
send Internal message to from Neighbors
04
reset diameter to 1
05
Send Internal Message
Return

Fig 34

Internet RFC/STD/FYI/BCP Archives

RFC1832

[Index | Search | What's New | Comments | Help]

Network Working Group   R. Srinivasan
Request for Comments: 1832   Sun Microsystems
Category: Standards Track   August 1995

XDR: External Data Representation Standard

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

ABSTRACT

This document describes the External Data Representation Standard (XDR) protocol as it is currently deployed and accepted.

TABLE OF CONTENTS

1. INTRODUCTION 2
2. BASIC BLOCK SIZE 2
3. XDR DATA TYPES 3
3.1 Integer 3
3.2 Unsigned Integer 4
3.3 Enumeration 4
3.4 Boolean 4
3.5 Hyper Integer and Unsigned Hyper Integer 4
3.6 Floating-point 5
3.7 Double-precision Floating-point 6
3.8 Quadruple-precision Floating-point 7
3.9 Fixed-length Opaque Data 8
3.10 Variable-length Opaque Data 8
3.11 String 9
3.12 Fixed-length Array 10
3.13 Variable-length Array 10
3.14 Structure 11
3.15 Discriminated Union 11
3.16 Void 12
3.17 Constant 12
3.18 Typedef 13

3.19 Optional-data 14
3.20 Areas for Future Enhancement 15
4. DISCUSSION 15
5. THE XDR LANGUAGE SPECIFICATION 17
5.1 Notational Conventions 17

5.2 Lexical Notes 17
5.3 Syntax Information 18
5.4 Syntax Notes 19
6. AN EXAMPLE OF AN XDR DATA DESCRIPTION 20
7. TRADEMARKS AND OWNERS 21
APPENDIX A: ANSI/IEEE Standard 754-1985 22
APPENDIX B: REFERENCES 24
Security Considerations 24
Author's Address 24

1. INTRODUCTION

XDR is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the SUN WORKSTATION*, VAX*, IBM-PC*, and Cray*. XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats. The language can only be used only to describe data; it is not a programming language. This language allows one to describe intricate data formats in a concise manner. The alternative of using graphical representations (itself an informal language) quickly becomes incomprehensible when faced with complexity. The XDR language itself is similar to the C language [1], just as Courier [4] is similar to Mesa. Protocols such as ONC RPC (Remote Procedure Call) and the NFS* (Network File System) use XDR to describe the format of their data.

The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8 bits of data. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning. For example, the Ethernet* standard suggests that bytes be encoded in "little-endian" style [2], or least significant bit first.

2. BASIC BLOCK SIZE

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through n-1. The bytes are read or written to some byte stream such that byte m always precedes byte m+1. If the n bytes needed to contain the data are not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of 4.

We include the familiar graphic box notation for illustration and comparison. In most illustrations, each box (delimited by a plus sign at the 4 corners and vertical bars and dashes) depicts a byte.

Ellipses (...) between boxes show zero or more additional bytes where required.

```
+--------+--------+...+--------+--------+...+--------+
| byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |   BLOCK
+--------+--------+...+--------+--------+...+--------+
|<-----------n bytes---------->|<------r bytes------>|
|<-----------n+r (where (n+r) mod 4 = 0)>----------->|
```

3. XDR DATA TYPES

Each of the sections that follow describes a data type defined in the XDR standard, shows how it is declared in the language, and includes a graphic illustration of its encoding.

For each data type in the language we show a general paradigm declaration. Note that angle brackets (< and >) denote variablelength sequences of data and square brackets ([and]) denote fixed-length sequences of data. "n", "m" and "r" denote integers. For the full language specification and more formal definitions of terms such as "identifier" and "declaration", refer to section 5: "The XDR Language Specification".

For some data types, more specific examples are included. A more extensive example of a data description is in section 6: "An Example of an XDR Data Description".

3.1 Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648,2147483647]. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. Integers are declared as follows:

```
int identifier;
```

```
  (MSB)                   (LSB)
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |                      INTEGER
+-------+-------+-------+-------+
<------------32 bits------------>
```

3.2. Unsigned Integer

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0,4294967295]. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. An unsigned integer is declared as follows:

```
unsigned int identifier;
```

```
  (MSB)                   (LSB)
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |             UNSIGNED INTEGER
+-------+-------+-------+-------+
<------------32 bits------------>
```

3.3 Enumeration

Enumerations have the same representation as signed integers. Enumerations are handy for describing subsets of the integers. Enumerated data is declared as follows:

```
enum { name-identifier = constant, ... } identifier;
```

For example, the three colors red, yellow, and blue could be described by an enumerated type:

```
enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

It is an error to encode as an enum any other integer than those that have been given assignments in the enum declaration.

3.4 Boolean

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are declared as follows:

```
bool identifier;
```

This is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;
```

3.5 Hyper Integer and Unsigned Hyper Integer

The standard also defines 64-bit (8-byte) numbers called hyper integer and unsigned hyper integer. Their representations are the obvious extensions of integer and unsigned integer defined above.

They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively. Their declarations:

```
hyper identifier; unsigned hyper identifier;

     (MSB)                                                   (LSB)
   +-------+-------+-------+-------+-------+-------+-------+-------+
   |byte 0 |byte 1 |byte 2 |byte 3 |byte 4 |byte 5 |byte 6 |byte 7 |
   +-------+-------+-------+-------+-------+-------+-------+-------+
   <----------------------------64 bits---------------------------->
                                              HYPER INTEGER
                                              UNSIGNED HYPER INTEGER
```

3.6 Floating-point

The standard defines the floating-point data type "float" (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision floating-point numbers [3]. The following three fields describe the single-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. 8 bits are devoted to this field. The exponent is biased by 127.

F: The fractional part of the number's mantissa, base 2. 23 bits are devoted to this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as follows:

```
float identifier;

+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |              SINGLE-PRECISION
S|   E   |          F          |         FLOATING-POINT NUMBER
+-------+-------+-------+-------+
1|<- 8 ->|<-------23 bits------>|
<------------32 bits------------>
```

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating-point number are 0 and 31. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 9, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow) [3]. According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be interpreted within XDR as anything other than "NaN".

3.7 Double-precision Floating-point

The standard defines the encoding for the double-precision floating-point data type "double" (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized double-precision floating-point numbers [3]. The standard encodes the following three fields, which describe the double-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. 11 bits are devoted to this field. The exponent is biased by 1023.

F: The fractional part of the number's mantissa, base 2. 52 bits are devoted to this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as follows:

```
double identifier;
```

```
+------+------+------+------+------+------+------+------+
|byte 0|byte 1|byte 2|byte 3|byte 4|byte 5|byte 6|byte 7|
S|    E   |                    F                        |
+------+------+------+------+------+------+------+------+
1|<--11-->|<-----------------52 bits------------------->|
<-----------------------64 bits------------------------->
                               DOUBLE-PRECISION FLOATING-POINT
```

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a double-precision floating-point number are 0 and 63. The beginning bit (and most significant bit) offsets of S, E , and F are 0, 1, and 12, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow) [3]. According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be interpreted within XDR as anything other than "NaN".

3.8 Quadruple-precision Floating-point

The standard defines the encoding for the quadruple-precision floating-point data type "quadruple" (128 bits or 16 bytes). The encoding used is designed to be a simple analog of of the encoding used for single and double-precision floating-point numbers using one form of IEEE double extended precision. The standard encodes the following three fields, which describe the quadruple-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. 15 bits are devoted to this field. The exponent is biased by 16383.

F: The fractional part of the number's mantissa, base 2. 112 bits are devoted to this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as follows:

```
quadruple identifier;
```

```
+------+------+------+------+------+------+-...--+------+
|byte 0|byte 1|byte 2|byte 3|byte 4|byte 5| ...  |byte15|
         S|    E          |                  F                      |
         +------+------+------+------+------+------+-...--+------+
         1|<----15---->|<-------------112 bits------------------>|
         <-----------------------128 bits------------------------>
                                      QUADRUPLE-PRECISION FLOATING-POINT
```

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a quadruple-precision floating-point number are 0 and 127. The beginning bit (and most significant bit) offsets of S, E , and F are 0, 1, and 16, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The encoding for signed zero, signed infinity (overflow), and denormalized numbers are analogs of the corresponding encodings for single and double-precision floating-point numbers [5], [6]. The "NaN" encoding as it applies to quadruple-precision floating-point numbers is system dependent and should not be interpreted within XDR as anything other than "NaN".

3.9 Fixed-length Opaque Data

At times, fixed-length uninterpreted data needs to be passed among machines. This data is called "opaque" and is declared as follows:

```
opaque identifier[n];
```

where the constant n is the (static) number of bytes necessary to contain the opaque data. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count of the opaque object a multiple of four.

```
         0        1     ...
     +--------+--------+...+--------+--------+...+--------+
     | byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |
     +--------+--------+...+--------+--------+...+--------+
     |<-----------n bytes---------->|<------r bytes------>|
     |<-----------n+r (where (n+r) mod 4 = 0)------------>|
                                              FIXED-LENGTH OPAQUE
```

3.10 Variable-length Opaque Data

The standard also provides for variable-length (counted) opaque data, defined as a sequence of n (numbered 0 through n-1) arbitrary bytes to be the number n encoded as an unsigned integer (as described below), and followed by the n bytes of the sequence.

Byte m of the sequence always precedes byte m+1 of the sequence, and byte 0 of the sequence always follows the sequence's length (count). If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Variable-length opaque data is declared in the following way:

```
opaque identifier<m>;
```

```
   or
      opaque identifier<>;
```

The constant m denotes an upper bound of the number of bytes that the sequence may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) - 1, the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that the maximum data transfer size is 8192 bytes, as follows:

```
      opaque filedata<8192>;

         0     1     2     3     4     5   ...
      +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
      |        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
      +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
      |<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
                              |<----n+r (where (n+r) mod 4 = 0)---->|
                                               VARIABLE-LENGTH OPAQUE
```

It is an error to encode a length greater than the maximum described in the specification.

3.11 String

The standard defines a string of n (numbered 0 through n-1) ASCII bytes to be the number n encoded as an unsigned integer (as described above), and followed by the n bytes of the string. Byte m of the string always precedes byte m+1 of the string, and byte 0 of the string always follows the string's length. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Counted byte strings are declared as follows:

```
      string object<m>;
   or
      string object<>;
```

The constant m denotes an upper bound of the number of bytes that a string may contain. If m is not specified, as in the second

declaration, it is assumed to be (2**32) - 1, the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, as follows:

```
      string filename<255>;

         0     1     2     3     4     5   ...
      +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
      |        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
      +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
      |<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
                              |<----n+r (where (n+r) mod 4 = 0)---->|
                                                               STRING
```

It is an error to encode a length greater than the maximum described in the specification.

3.12 Fixed-length Array

Declarations for fixed-length arrays of homogeneous elements are in the following form:

```
      type-name identifier[n];
```

Fixed-length arrays of elements numbered 0 through n-1 are encoded by individually encoding the elements of the array in their natural order, 0 through n-1. Each element's size is a multiple of four bytes. Though all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of type "string", yet each element will vary in its length.

```
      +---+---+---+---+---+---+---+---+...+---+---+---+---+
      |   element 0   |   element 1   |...|  element n-1  |
      +---+---+---+---+---+---+---+---+...+---+---+---+---+
      |<--------------------n elements------------------->|

                                          FIXED-LENGTH ARRAY
```

3.13 Variable-length Array

Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count n (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element n- 1. The declaration for variable-length arrays follows this form:

```
      type-name identifier<m>;
   or
      type-name identifier<>;
```

The constant m specifies the maximum acceptable element count of an array; if m is not specified, as in the second declaration, it is assumed to be (2**32) - 1.

```
        0  1  2  3
      +--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
      |     n     | element 0 | element 1 |...|element n-1|
      +--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
      |<-4 bytes->|<--------------n elements------------->|
                                                COUNTED ARRAY
```

It is an error to encode a value of n that is greater than the maximum described in the specification.

3.14 Structure

Structures are declared as follows:

```
      struct {
         component-declaration-A;
         component-declaration-B;
         ...
      } identifier;
```

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of four bytes, though the components may be different sizes.

```
      +-------------+-------------+...
      | component A | component B |...                     STRUCTURE
      +-------------+-------------+...
```

3.15 Discriminated Union

A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either "int", "unsigned int", or an enumerated type, such as "bool". The component types are called "arms" of the union, and are preceded by the value of the discriminant which implies their encoding. Discriminated unions are declared as follows:

```
      union switch (discriminant-declaration) {
      case discriminant-value-A:

         arm-declaration-A;
      case discriminant-value-B:
         arm-declaration-B;
      ...
      default: default-declaration;
      } identifier;
```

Each "case" keyword is followed by a legal value of the discriminant. The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of four bytes.

The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

```
        0   1   2   3
      +---+---+---+---+---+---+---+---+
      |  discriminant |  implied arm  |          DISCRIMINATED UNION
      +---+---+---+---+---+---+---+---+
      |<---4 bytes--->|
```

3.16 Void

An XDR void is a 0-byte quantity. Voids are useful for describing operations that take no data as input or no data as output. They are also useful in unions, where some arms may contain data and others do not. The declaration is simply as follows:

```
      void;
```

Voids are illustrated as follows:

```
        ++
         ||                                                   VOID
         ++
       --><-- 0 bytes
```

3.17 Constant

The data declaration for a constant follows this form:

```
const name-identifier = n;
```

"const" is used to define a symbolic name for a constant; it does not declare any data. The symbolic constant may be used anywhere a regular constant may be used. For example, the following defines a symbolic constant DOZEN, equal to 12.

```
const DOZEN = 12;
```

3.18 Typedef

"typedef" does not declare any data either, but serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration part of the typedef. For example, the following defines a new type called "eggbox" using an existing type called "egg":

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name would have in the typedef, if it was considered a variable. For example, the following two declarations are equivalent in declaring the variable "fresheggs":

```
eggbox  fresheggs; egg     fresheggs[DOZEN];
```

When a typedef involves a struct, enum, or union definition, there is another (preferred) syntax that may be used to define the same type. In general, a typedef of the following form:

```
typedef <<struct, union, or enum definition>> identifier;
```

may be converted to the alternative form by removing the "typedef" part and placing the identifier after the "struct", "union", or "enum" keyword, instead of at the end. For example, here are the two ways to define the type "bool":

```
typedef enum {    /* using typedef */
   FALSE = 0,
   TRUE = 1
} bool;

enum bool {       /* preferred alternative */
   FALSE = 0,
   TRUE = 1
};
```

The reason this syntax is preferred is one does not have to wait until the end of a declaration to figure out the name of the new type.

3.19 Optional-data

Optional-data is one kind of union that occurs so frequently that we give it a special syntax of its own for declaring it. It is declared as follows:

```
type-name *identifier;
```

This is equivalent to the following union:

```
union switch (bool opted) {
case TRUE:
   type-name element;
case FALSE:
   void;
} identifier;
```

It is also equivalent to the following variable-length array declaration, since the boolean "opted" can be interpreted as the length of the array:

```
type-name identifier<1>;
```

Optional-data is not so interesting in itself, but it is very useful for describing recursive data-structures such as linked-lists and trees. For example, the following defines a type "stringlist" that encodes lists of arbitrary length strings:

```
struct *stringlist {
   string item<>;
   stringlist next;
};
```

It could have been equivalently declared as the following union:

```
union stringlist switch (bool opted) {
case TRUE:
   struct {
      string item<>;
      stringlist next;
   } element;
case FALSE:
   void;
};
```

or as a variable-length array:

```
struct stringlist<1> {

   string item<>;
   stringlist next;
};
```

Both of these declarations obscure the intention of the stringlist type, so the optional-data declaration is preferred over both of them. The optional-data type also has a close correlation to how recursive data structures are represented in high-level languages such as Pascal or C by use of pointers. In fact, the syntax is the same as that of the C language for pointers.

3.20 Areas for Future Enhancement

The XDR standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. Also missing are packed (or binary-coded) decimals.

The intent of the XDR standard was not to describe every kind of data that people have ever sent or will ever want to send from machine to machine. Rather, it only describes the most commonly used data-types of high-level languages such as Pascal or C so that applications written in these languages will be able to communicate easily over some medium.

One could imagine extensions to XDR that would let it describe almost any existing protocol, such as TCP. The minimum necessary for this are support for different block sizes and byte-orders. The XDR discussed here could then be considered the 4-byte big-endian member of a larger XDR family.

4. DISCUSSION

(1) Why use a language for describing data? What's wrong with diagrams?

There are many advantages in using a data-description language such as XDR versus using diagrams. Languages are more formal than diagrams and lead to less ambiguous descriptions of data. Languages are also easier to understand and allow one to think of other issues instead of the low-level details of bit-encoding. Also, there is a close analogy between the types of XDR and a high-level language such as C or Pascal. This makes the implementation of XDR encoding and decoding modules an easier task. Finally, the language specification itself is an ASCII string that can be passed from machine to machine to perform on-the-fly data interpretation.

(2) Why is there only one byte-order for an XDR unit?

Supporting two byte-orderings requires a higher level protocol for determining in which byte-order the data is encoded. Since XDR is not a protocol, this can't be done. The advantage of this, though, is that data in XDR format can be written to a magnetic tape, for example, and any machine will be able to interpret it, since no higher level protocol is necessary for determining the byte-order.

(3) Why is the XDR byte-order big-endian instead of little-endian? Isn't this unfair to little-endian machines such as the VAX(r), which has to convert from one form to the other?

Yes, it is unfair, but having only one byte-order means you have to be unfair to somebody. Many architectures, such as the Motorola 68000* and IBM 370*, support the big-endian byte-order.

(4) Why is the XDR unit four bytes wide?

There is a tradeoff in choosing the XDR unit size. Choosing a small size such as two makes the encoded data small, but causes alignment problems for machines that aren't aligned on these boundaries. A large size such as eight means the data will be aligned on virtually every machine, but causes the encoded data to grow too big. We chose four as a compromise. Four is big enough to support most architectures efficiently, except for rare machines such as the eight-byte aligned Cray*. Four is also small enough to keep the encoded data restricted to a reasonable size.

(5) Why must variable-length data be padded with zeros?

It is desirable that the same data encode into the same thing on all machines, so that encoded data can be meaningfully compared or checksummed. Forcing the padded bytes to be zero ensures this.

(6) Why is there no explicit data-typing?

Data-typing has a relatively high cost for what small advantages it may have. One cost is the expansion of data due to the inserted type fields. Another is the added cost of interpreting these type fields and acting accordingly. And most protocols already know what type they expect, so data-typing supplies only redundant information. However, one can still get the benefits of data-typing using XDR. One way is to encode two things: first a string which is the XDR data description of the encoded data, and then the encoded data itself. Another way is to assign a value to all the types in XDR, and then define a universal type which takes this value as its discriminant and for each value, describes the corresponding data type.

5. THE XDR LANGUAGE SPECIFICATION

5.1 Notational Conventions

This specification uses an extended Back-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

(1) The characters '|', '(', ')', '[', ']', '"', and '*' are special. (2) Terminal symbols are strings of any characters surrounded by double quotes. (3) Non-terminal symbols are strings of non-special characters. (4) Alternative items are separated by a vertical bar ("|"). (5) Optional items are enclosed in brackets. (6) Items are grouped together by enclosing them in parentheses. (7) A '*' following an item means 0 or more occurrences of that item.

For example, consider the following pattern:

```
      "a " "very" (", " "very")* [" cold " "and "]  " rainy "
      ("day" | "night")
```

An infinite number of strings match this pattern. A few of them are:

```
         "a very rainy day"
         "a very, very rainy day"
         "a very cold and  rainy day"
         "a very, very, very cold and  rainy night"
```

5.2 Lexical Notes

(1) Comments begin with '/*' and terminate with '*/'. (2) White space serves to separate items and is otherwise ignored. (3) An identifier is a letter followed by an optional sequence of letters, digits or underbar ('_'). The case of identifiers is not ignored. (4) A constant is a sequence of one or more decimal digits, optionally preceded by a minus-sign ('-').

5.3 Syntax Information

```
      declaration:
           type-specifier identifier
         | type-specifier identifier "[" value "]"
         | type-specifier identifier "<" [ value ] ">"
         | "opaque" identifier "[" value "]"
         | "opaque" identifier "<" [ value ] ">"
         | "string" identifier "<" [ value ] ">"
         | type-specifier "*" identifier
         | "void"

      value:
           constant
         | identifier

      type-specifier:
           [ "unsigned" ] "int"
         | [ "unsigned" ] "hyper"
         | "float"
         | "double"
         | "quadruple"
         | "bool"
         | enum-type-spec
         | struct-type-spec
         | union-type-spec
         | identifier

      enum-type-spec:
         "enum" enum-body

      enum-body:
         "{"
            ( identifier "=" value )
            ( "," identifier "=" value )*
         "}"

      struct-type-spec:
         "struct" struct-body

      struct-body:
         "{"
```

```
            ( declaration ";" )
            ( declaration ";" )*
         "}"

      union-type-spec:
         "union" union-body

      union-body:
         "switch" "(" declaration ")" "{"
            ( "case" value ":" declaration ";" )
            ( "case" value ":" declaration ";" )*
            [ "default" ":" declaration ";" ]
         "}"

      constant-def:
         "const" identifier "=" constant ";"

      type-def:
           "typedef" declaration ";"
         | "enum" identifier enum-body ";"
         | "struct" identifier struct-body ";"
         | "union" identifier union-body ";"

      definition:
           type-def
         | constant-def

      specification:
           definition *
```

5.4 Syntax Notes

(1) The following are keywords and cannot be used as identifiers: "bool", "case", "const", "default", "double", "quadruple", "enum", "float", "hyper", "opaque", "string", "struct", "switch", "typedef", "union", "unsigned" and "void".

(2) Only unsigned constants may be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a "const" definition.

(3) Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.

(4) Similarly, variable names must be unique within the scope of struct and union declarations. Nested struct and union declarations create new scopes.

(5) The discriminant of a union must be of a type that evaluates to an integer. That is, "int", "unsigned int", "bool", an enumerated type or any typedefed type that evaluates to one of these is legal. Also, the case values must be one of the legal values of the discriminant. Finally, a case value may not be specified more than once within the scope of a union declaration.

6. AN EXAMPLE OF AN XDR DATA DESCRIPTION

Here is a short XDR data description of a thing called a "file", which might be used to transfer files from one machine to another.

```
      const MAXUSERNAME = 32;     /* max length of a user name */
      const MAXFILELEN = 65535;   /* max length of a file      */
      const MAXNAMELEN = 255;     /* max length of a file name */

      /*
       * Types of files:
       */
      enum filekind {
         TEXT = 0,       /* ascii data */
         DATA = 1,       /* raw data   */
         EXEC = 2        /* executable */
      };

      /*
       * File information, per kind of file:
       */
      union filetype switch (filekind kind) {
      case TEXT:
         void;                           /* no extra information */
      case DATA:
         string creator<MAXNAMELEN>;     /* data creator         */
      case EXEC:
         string interpretor<MAXNAMELEN>; /* program interpretor  */
      };

      /*
       * A complete file:
       */
      struct file {
         string filename<MAXNAMELEN>; /* name of file    */
         filetype type;               /* info about file */
         string owner<MAXUSERNAME>;   /* owner of file   */
         opaque data<MAXFILELEN>;     /* file data       */
      };
```

Suppose now that there is a user named "john" who wants to store his lisp program "sillyprog" that contains just the data "(quit)". His file would be encoded as follows:

```
    OFFSET  HEX BYTES       ASCII    COMMENTS
    ------  ---------       -----    --------
     0      00 00 00 09     ....     -- length of filename = 9
     4      73 69 6c 6c     sill     -- filename characters
     8      79 70 72 6f     ypro     -- ... and more characters ...
    12      67 00 00 00     g...     -- ... and 3 zero-bytes of fill
    16      00 00 00 02     ....     -- filekind is EXEC = 2
    20      00 00 00 04     ....     -- length of interpretor = 4
    24      6c 69 73 70     lisp     -- interpretor characters
    28      00 00 00 04     ....     -- length of owner = 4
    32      6a 6f 68 6e     john     -- owner characters
    36      00 00 00 06     ....     -- length of file data = 6
    40      28 71 75 69     (qui     -- file data bytes ...
    44      74 29 00 00     t)..     -- ... and 2 zero-bytes of fill
```

7. TRADEMARKS AND OWNERS

SUN WORKSTATION	Sun Microsystems, Inc.
VAX	Digital Equipment Corporation
IBM-PC	International Business Machines Corporation
Cray	Cray Research
NFS	Sun Microsystems, Inc.
Ethernet	Xerox Corporation.
Motorola 68000	Motorola, Inc.
IBM 370	International Business Machines Corporation

APPENDIX A: ANSI/IEEE Standard 754-1985

The definition of NaNs, signed zero and infinity, and denormalized numbers from [3] is reproduced here for convenience. The definitions for quadruple-precision floating point numbers are analogs of those for single and double-precision floating point numbers, and are defined in [3].

In the following, 'S' stands for the sign bit, 'E' for the exponent, and 'F' for the fractional part. The symbol 'u' stands for an undefined bit (0 or 1).

For single-precision floating point numbers:

Type	S (1 bit)	E (8 bits)	F (23 bits)
signalling NaN	u	255 (max)	.0uuuuu---u (with at least one 1 bit)
quiet NaN	u	255 (max)	.1uuuuu---u
negative infinity	1	255 (max)	.000000---0
positive infinity	0	255 (max)	.000000---0
negative zero	1	0	.000000---0
positive zero	0	0	.000000---0

For double-precision floating point numbers:

Type	S (1 bit)	E (11 bits)	F (52 bits)
signalling NaN	u	2047 (max)	.0uuuuu---u (with at least one 1 bit)
quiet NaN	u	2047 (max)	.1uuuuu---u
negative infinity	1	2047 (max)	.000000---0
positive infinity	0	2047 (max)	.000000---0
negative zero	1	0	.000000---0
positive zero	0	0	.000000---0

For quadruple-precision floating point numbers:

Type	S (1 bit)	E (15 bits)	F (112 bits)
signalling NaN	u	32767 (max)	.0uuuuu---u (with at least one 1 bit)
quiet NaN	u	32767 (max)	.1uuuuu---u
negative infinity	1	32767 (max)	.000000---0
positive infinity	0	32767 (max)	.000000---0
negative zero	1	0	.000000---0
positive zero	0	0	.000000---0

Subnormal numbers are represented as follows:

Precision	Exponent	Value
Single	0	(-1)**S * 2**(-126) * 0.F
Double	0	(-1)**S * 2**(-1022) * 0.F
Quadruple	0	(-1)**S * 2**(-16382) * 0.F

APPENDIX B: REFERENCES

[1] Brian W. Kernighan & Dennis M. Ritchie, "The C Programming Language", Bell Laboratories, Murray Hill, New Jersey, 1978.

[2] Danny Cohen, "On Holy Wars and a Plea for Peace", IEEE Computer, October 1981.

[3] "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

[4] "Courier: The Remote Procedure Call Protocol", XEROX Corporation, XSIS 038112, December 1981.

[5] "The SPARC Architecture Manual: Version 8", Prentice Hall, ISBN 0-13-825001-4.

[6] "HP Precision Architecture Handbook", June 1987, 5954-9906.

[7] Srinivasan, R., "Remote Procedure Call Protocol Version 2", RFC 1831, Sun Microsystems, Inc., August 1995.

Security Considerations

Security issues are not discussed in this memo.

Author's Address

Raj Srinivasan
Sun Microsystems, Inc.
ONC Technologies
2550 Garcia Avenue
M/S MTV-5-40
Mountain View, CA 94043
USA

Phone: 415-336-2478
Fax: 415-336-6015
EMail: raj@eng.sun.com

Comments/Questions about this archive ? Send mail to rfc-admin@faqs.org

t.120

A Primer
on the
T.120
Series
Standard

DataBeam™

A PRIMER ON THE T.120 SERIES STANDARDS

The T.120 standard contains a series of communication and application protocols and services that provide support for real-time, multipoint data communications. These multipoint facilities are important building blocks for a whole new range of collaborative applications, including desktop data conferencing, multi-user applications, and multi-player gaming.

Broad in scope, T.120 is a comprehensive specification that solves several problems that have historically slowed market growth for applications of this nature. Perhaps most importantly, T.120 resolves complex technological issues in a manner that is acceptable to both the computing and telecommunications industries.

Broad vendor support means that end users will be able to choose from a variety of interoperable products.

Established by the International Telecommunications Union (ITU), T.120 is a family of open standards that was defined by leading data communication practitioners in the industry. Over 100 key international vendors, including Apple, AT&T, British Telecom, Cisco Systems, Intel, MCI, Microsoft, and PictureTel, have committed to implementing T.120-based products and services.

While T.120 has emerged as a critical element in the data communications landscape, the only information that currently exists on the topic is a weighty and complicated set of standards documents. This primer bridges this information gap by summarizing T.120's major benefits, fundamental architectural elements, and core capabilities.

KEY BENEFITS OF T.120

So why all the excitement about T.120? The bottom line is that it provides exceptional benefits to end users, vendors, and developers tasked with implementing realtime applications. The following list is a high-level overview of the major benefits associated with the T.120 standard:

Multipoint Data Delivery

T.120 provides an elegant abstraction for developers to create and manage a multipoint domain with ease. From an application perspective, data is seamlessly delivered to multiple parties in "realtime."

Interoperability

T.120 allows endpoint applications from multiple vendors to interoperate. T.120 also specifies how applications may interoperate with (or through) a variety of network bridging products and services that also support the T.120 standard.

Reliable Data Delivery

Error-corrected data delivery ensures that all endpoints will receive each data transmission.

T.120 BENEFITS

✔ Multipoint Data Delivery
✔ Interoperability
✔ Reliable Data Delivery
✔ Multicast Enabled Delivery
✔ Network Transparency
✔ Platform Independence
✔ Network Independence
✔ Support for Varied Topologies
✔ Application Independence
✔ Scalability
✔ Co-existence with Other Standards
✔ Extendability

Multicast Enabled Delivery

In mulilticast enabled networks, T.120 can employ reliable (ordered, guaranteed) and unreliable delivery services. Unreliable data delivery is also available without multicast. By using multicast, the T.120 infrastructure reduces network congestion and improves performance for the end user. The T.120 infrastructure can use both unicast and multicast simultaneously, providing a flexible solution for mixed unicast and multicast networks. The Multicast Adaptation Protocol (MAP) is expected to be ratified in early 1998.

Network Transparency

Applications are completely shielded from the underlying data transport mechanism being used. Whether the transport is a high-speed LAN or a simple dial-up modem, the application developer is only concerned with a single, consistent set of application services.

Platform Independence

Because the T.120 standard is completely free from any platform dependencies, it will readily take advantage of the inevitable advances in computing technology. In fact, DataBeam's customers have already ported the T.120 source code easily from Windows to a variety of environments, including OS/2, MAC/OS, several versions of UNIX, and other proprietary real-time operating systems.

Network Independence

The T.120 standard supports a broad range of transport options, including the Public Switched Telephone Networks (PSTN or POTS), Integrated Switched Digital Networks (ISDN), Packet Switched Digital Networks (PSDN), Circuit Switched Digital Networks (CSDN), and popular local area network protocols (such as TCP/IP and IPX via reference protocol). Furthermore, these vastly different network transports, operating at different speeds, can easily coexist in the same multipoint conference.

Support for Varied Topologies

Multipoint conferences can be set up with virtually no limitation on network topology. Star topologies, with a single Multipoint Control Unit (MCU) will be common early on. The standard also supports a wide variety of other topologies ranging from those with multiple, cascaded MCUs to topologies as simple as a daisy-chain. In complex multipoint conferences, topology may have a significant impact on efficiency and performance.

Application Independence

Although the driving market force behind T.120 was teleconferencing, its designers purposely sought to satisfy a much broader range of application needs. Today, T.120 provides a generic, real-time communications facility that can be used by many different applications. These applications include interactive gaming, virtual reality and simulations, real-time subscription news feeds, and process control applications.

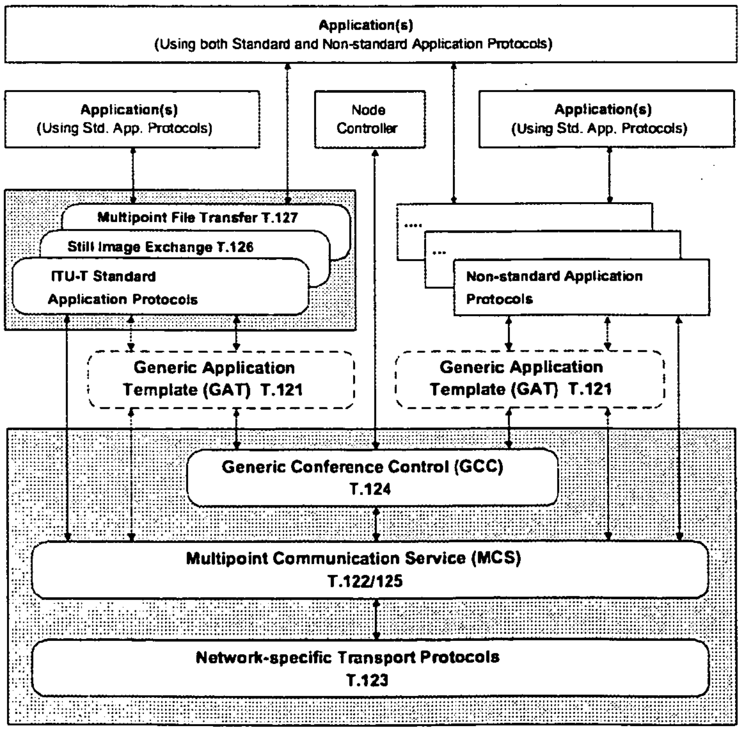
Scalability

T.120 is defined to be easily scalable from simple PC-based architectures to complex multi-processor environments characterized by their high performance. Resources for T.120 applications are plentiful, with practical limits imposed only by the confines of the specific platform running the software.

Co-existence with Other Standards

T.120 was designed to work alone or within the larger context of other ITU standards, such as the H.32x family of video conferencing standards. T.120 also supports and cross-references other important ITU standards, such as V.series modems.

FIGURE 1: MODEL OF ITU T.120 SERIES ARCHITECTURE


Application(s)
(Using both Standard and Non-standard Application Protocols)
Application(s)
(Using Std. App. Protocols)
Node
Controller
Application(s)
(Using Std. App. Protocols)
Multipoint File Transfer T.127
Still Image Exchange T.126
ITU-T Standard
Application Protocols
....
...
Non-standard Application
Protocols
Generic Application
Template (GAT) T.121
Generic Application
Template (GAT) T.121
Generic Conference Control (GCC)
T.124
Multipoint Communication Service (MCS)
T.122/125
Network-specific Transport Protocols
T.123

Extendability

The T.120 standard can be freely extended to include a variety of new capabilities, such as support for new transport stacks (like ATM or Frame Relay), improved security measures, and new application-level protocols.

ARCHITECTURAL OVERVIEW

The T.120 architecture relies on a multi-layered approach with defined protocols and service definitions between layers. Each layer presumes that all layers exist below. Figure 1 provides a graphical representation of the T.120 architecture.

The lower level layers (T.122, T.123, T.124, and T.125) specify an application-independent mechanism for providing multipoint data communication services to any application that can use these facilities. The upper level layers (T.126 and T.127) define protocols for specific conferencing applications, such as shared whiteboarding and multipoint file transfer. Applications using these standardized protocols can co-exist in the same conference with applications using proprietary protocols. In fact, a single application may even use a mix of standardized and non-standardized protocols.

COMPONENT OVERVIEW

The following overview describes the key characteristics and concepts behind each individual component of the T.120 standard. This overview starts at the bottom of the T.120 stack and progresses upward.

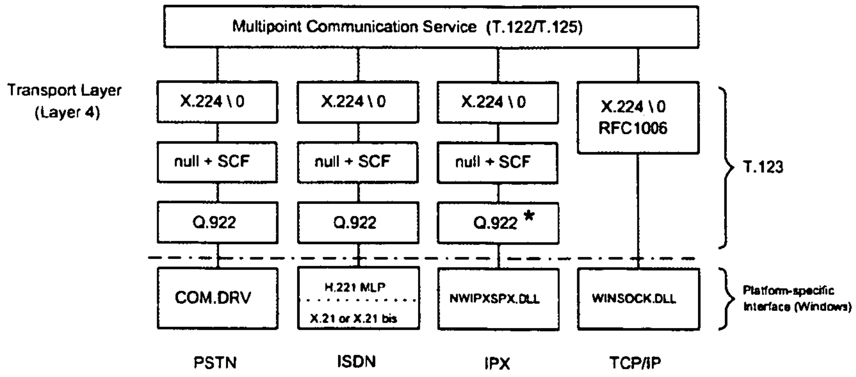
Transport Stacks - T.123

T.120 applications expect the underlying transport to provide reliable delivery of its Protocol Data Units (PDUs) and to segment and sequence that data. T.123 specifies transport profiles for each of the following:

- Public Switched Telephone Networks (PSTN)
- Integrated Switched Digital Networks (ISDN)
- Circuit Switched Digital Networks (CSDN)
- Packet Switched Digital Networks (PSDN)
- TCP/IP
- Novell Netware IPX (via reference profile)

As highlighted below in Figure 2, the T.123 layer presents a uniform OSI transport interface and services (X.214/X.224)

FIGURE 2: CROSS-SECTION OF T.123 TRANSPORTS (BASIC MODE PROFILES)


Multipoint Communication Service (T.122/T.125)
Transport Layer (Layer 4)
X.224 \ 0
X.224 \ 0
X.224 \ 0
X.224 \ 0 RFC1006
null + SCF
null + SCF
null + SCF
Q.922
Q.922
Q.922 *
T.123
COM.DRV
H.221 MLP
X.21 or X.21 bis
NWIPXSPX.DLL
WINSOCK.DLL
Platform-specific Interface (Windows)
PSTN
ISDN
IPX
TCP/IP

★ Subset of Q.922

to the MCS layer above. The T.123 layer includes built-in error correction facilities so application developers do not have to rely on special hardware facilities to perform this function.

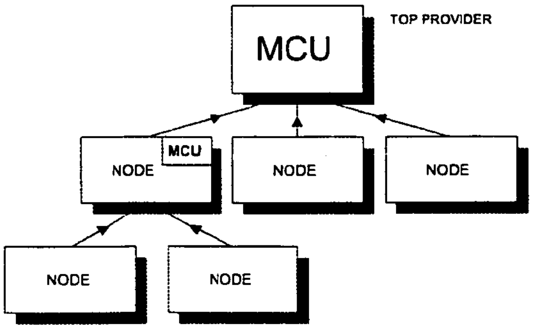
In a given computing environment, a transport stack typically plugs into a local facility that provides an interface to the specific transport connection. For example, in the Windows environment, DataBeam's transport stacks plug into COMM.DRV for modem communications, WINSOCK.DLL for TCP/IP and UDP/IP communications, and NWIPXSPX. DLL for Novell IPX communications support.

The MCU is a logical construct whose role may be served by a node on a desktop or by special-purpose equipment within the network.

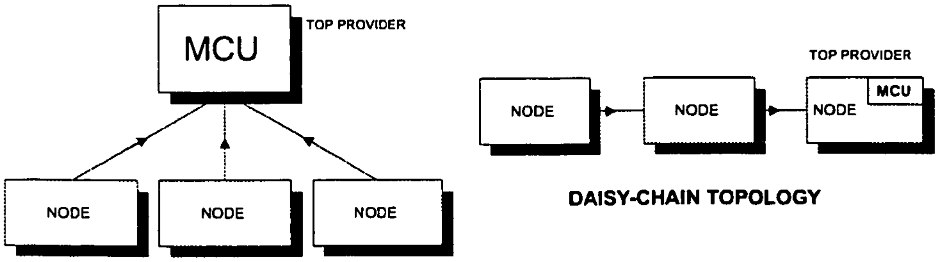
The Multicast Adaptation Protocol (MAP) service layer is a new extension to MCS. MAP manages unicast- and multicast-based transports. MAP can be used with any transport where multicast is available, such as IP networks. While multicast provides unreliable delivery, many applications using T.120 require reliable services. Developers can incorporate a variety of multicast error correction schemes into MAP, thereby selecting the scheme most closely aligned with their application.

In 1996, the ITU is expected to adopt extensions to support important new transport facilities, such as Asynchronous Transfer Mode (ATM) and H.324 POTS videophone. It is necessary to note that developers can easily produce a proprietary transport stack (supporting, for example, AppleTalk) that transparently uses the services above T.123. An important function of MCUs or T.120-enabled bridges, routers, or gateways is to provide transparent interworking across different network boundaries.

FIGURE 3: EXAMPLES OF VALID MCS TOPOLOGIES



CASCADED MCU TOPOLOGY



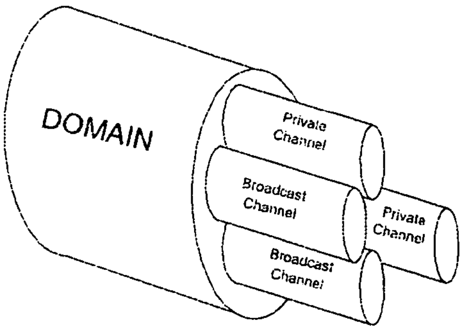
DAISY-CHAIN TOPOLOGY

TRADITIONAL STAR TOPOLOGY

Multipoint Communication Service (MCS) - T.122, T.125

T.122 defines the multipoint services available to the developer, while T.125 specifies the data transmission protocol. Together they form MCS, the multipoint "engine" of the T.120 conference. MCS relies on T.123 to deliver the data. (Use of MCS is entirely independent of the actual T.123 transport stack(s) that is loaded.)

FIGURE 4: CHANNEL DIAGRAM



MCS is a powerful tool that can be used to solve virtually any multipoint application design requirement. MCS is an elegant abstraction of a complex organism. Learning to use MCS effectively is the key to successfully developing real-time applications.

How MCS Works

In a conference, multiple endpoints (or MCS *nodes*) are logically connected together to form what T.120 refers to as a *domain*. Domains generally equate to the concept of a conference. An application may actually be *attached* to multiple domains simultaneously. For example, the chairperson of a large online conference may simultaneously monitor information being discussed among several activity groups.

In a T.120 conference, nodes connect upward to a Multipoint Control Unit (MCU). The MCU model in T.120 provides a reliable approach that works in both public and private networks. Multiple MCUs may be easily chained together in a single domain. Figure 3 illustrates three potential topology structures. Each domain has a single *Top Provider* or MCU that houses the information base critical to the conference. If the Top Provider either fails or leaves a conference, the conference is terminated. If a lower level MCU (i.e., not the Top Provider) fails, only the nodes on the tree below that MCU are dropped from the conference. Because all nodes contain MCS, they are all potentially "MCUs."

One of the critical features of the T.120 approach is the ability to direct data. This feature allows applications to communicate efficiently. MCS applications direct data within a domain via the use of *channels*. An application can choose to use multiple channels simultaneously for whatever purposes it needs (for example, separating annotation and file transfer operations). Application instances choose to obtain information by *subscribing* to whichever channel(s) contains the desired data. These channel assignments can be dynamically changed during the life of the conference. Figure 4 presents an overview of multiple channels in use within a domain.

It is the application developer's responsibility to determine how to use channels

TABLE 1: CHANNEL SETUP EXAMPLE

Channel	Type	Priority	Routing
1	Error Control Channels	Top	Standard
2	Annotations	High	Uniform
3	Bitmap Images	Medium	Uniform
4	File Transfer	Low	Standard

within an application. For example, an application may send control information along a single channel and application data along a series of channels that may vary depending upon the type of data being sent. The application developer may also take advantage of the MCS concept of *private channels* to direct data to a discrete subset of a given conference.

Data may be sent with one of four *priority* levels. MCS applications may also specify that data is routed along the quickest path of delivery using the *standard* send command. If the application uses the *uniform* send command, it ensures that data from multiple senders will arrive at all destinations in the same order. Uniform data always travels all the way up the tree to the Top Provider. Table 1 provides an example of how a document conferencing application could set up its channels. Reliable or unreliable data delivery is determined by the application.

There are no constraints on the size of the data sent from an application to MCS. Segmentation of data is automatically performed on behalf of the application. However, after receiving the data it is the application's responsibility to reassemble the data by monitoring flags provided when the data is delivered.

Tokens are the last major facility provided by MCS. Services are provided to grab, pass, inhibit, release, and query tokens. Token resources may be used as either exclusive (i.e., locking) or non-exclusive entities.

Tokens can be used by an application in a number of ways. For example, an application may specify that only the holder of a specific token, such as the conductor, may send information in the conference.

Another popular use of tokens is to coordinate tasks within a domain. For example, suppose a teacher wants to be sure that every student in a distance learning session answered a particular question before displaying the answer. Each node in the underlying application *inhibits* a specific token after receiving the request to answer the question. The token is released by each node when an answer is provided. In the background, the teacher's application continuously polls the state of the token. When all nodes have released the token, the application presents the teacher with a visual cue that the class is ready for the answer.

Generic Conference Control (GCC) - T.124

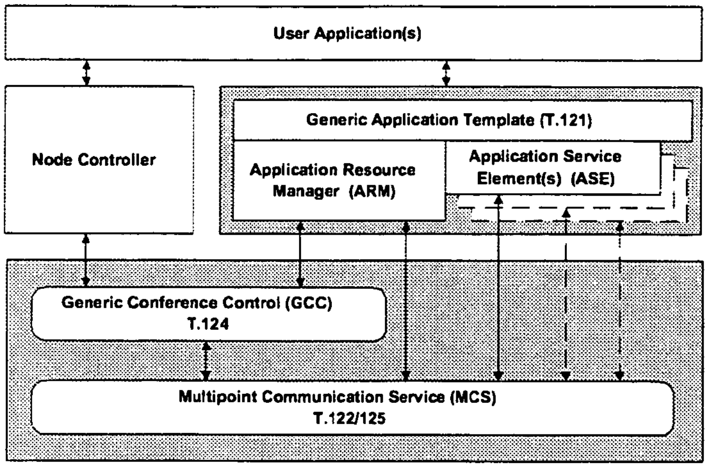
Generic Conference Control provides a comprehensive set of facilities for establishing and managing the multipoint conference. It is with GCC that we first see features that are specific to the electronic conference.

At the heart of GCC is an important information base about the state of the various conferences it may be servicing. One node, which may be the MCU itself, serves as the Top Provider for GCC information. Any actions or requests from lower GCC nodes ultimately filter up to this Top Provider.

One of GCC's most important roles is to maintain information about the nodes and applications that are in a conference.

Using mechanisms in GCC, applications create conferences, join conferences, and invite others to conferences. As endpoints join and leave conferences, the information base in GCC is updated and can be used to automatically notify all endpoints when these actions occur. GCC also knows who is the Top Provider for the conference. However, GCC does not contain detailed topology information about the means by which nodes from lower branches are connected to the conference.

FIGURE 5: T.121 GENERIC APPLICATION TEMPLATE


User Application(s)
Node Controller
Generic Application Template (T.121)
Application Resource Manager (ARM)
Application Service Element(s) (ASE)
Generic Conference Control (GCC) T.124
Multipoint Communication Service (MCS) T.122/125

Every application in a conference must register its unique *application key* with GCC. This enables any subsequent joining nodes to find compatible applications. Furthermore, GCC provides robust facilities for applications to exchange capabilities and arbitrate feature sets. In this way, applications from different vendors can readily establish whether or not they can interoperate and at what feature level. This arbitration facility is the mechanism used to ensure backward compatibility between different versions of the same application.

GCC also provides conference security. This allows applications to incorporate password protection or "lock" facilities to prevent uninvited users from joining a conference.

Another key function of GCC is its ability to dynamically track MCS resources. Since multiple applications can use MCS at the same time, applications rely on GCC to prevent conflicts for MCS resources, such as channels and tokens. This ensures that applications do not step on each other by attaching to the same channel or requesting a token already in use by another application.

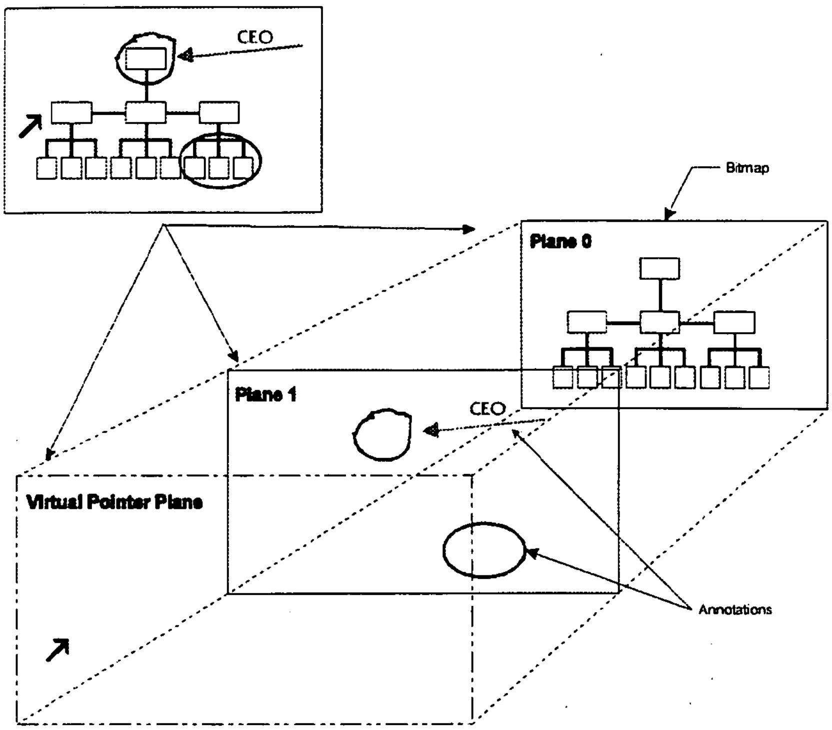
Finally, GCC provides capabilities for supporting the concept of *conductorship* in a conference. GCC allows the application to identify the conductor and a means in which to transfer the conductor's "baton." The developer is free to decide how to use these conductorship facilities within the application.

T.124 Revised

As part of the ongoing enhancement process for the T.120 standards, the ITU has completed a draft revision of T.124. The new version, called T.124 Revised, introduces a number of changes to improve scalability. The most significant changes address the need to distribute roster information to all nodes participating in a conference, as well as improvements in the efficiency of sending roster refresh information (from the Top Provider) any time a node joins or leaves a conference.

To improve the distribution of roster information, the concept of Node Categories was introduced. These categories provide a way for a T.124 node to join or leave a conference without affecting the roster information that was distributed throughout a conference. In addi-

FIGURE 6: T.126 WORKSPACE DIAGRAM



tion, the Full Roster Refresh, which was previously sent any time a new node joined a conference, was eliminated by sending out roster details from the Top Provder. These changes will not affect backward compatibility to earlier revisions of T.124. This revision will go to the ITU for Decision in March of 1998.

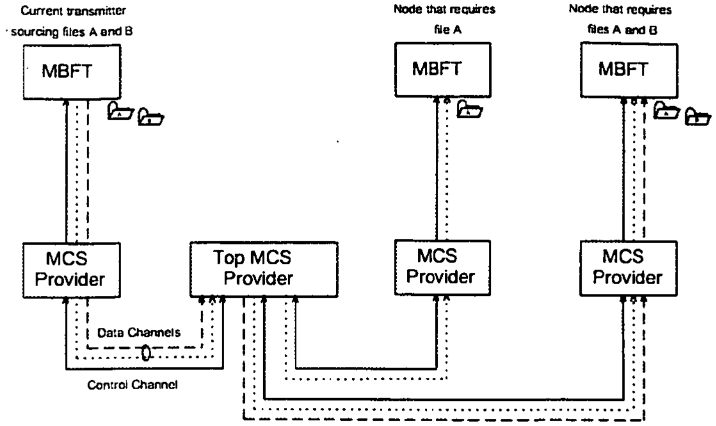
Generic Application Template (GAT) - T.121

T.121 provides a template for T.120 resource management that developers should use as a guide for building application protocols. T.121 is mandatory for standardized application protocols and is highly recommended for non-standard application protocols. The template ensures consistency and reduces the potential for unforeseen interaction between different protocol implementations.

Within the T.121 model, GAT defines a generic Application Resource Manager (ARM). This entity manages GCC and MCS resources on behalf of the application protocol-specific functionality defined as an Application Service Element (ASE). Figure 5 demonstrates the GAT model within the T.120 architecture. Simply put, GAT provides a consistent model for managing T.120 resources required by the application to which the developer adds application-specific functionality.

GAT's functionality is considered to be generic and common to all application protocols. GAT's services include enrolling the application in GCC and attaching to MCS domains. GAT also manages channels, tokens, and capabilities on behalf of the application. On a broader scale, GAT responds to GCC indica-

FIGURE 7: T.127 FILE TRANSFER MODEL


Current transmitter sourcing files A and B
Node that requires file A
Node that requires files A and B
MBFT
MBFT
MBFT
MCS Provider
Top MCS Provider
MCS Provider
MCS Provider
Data Channels
Control Channel

tions and can invoke peer applications on other nodes in the conference.

Still Image Exchange and Annotation (SI) - T.126

T.126 defines a protocol for viewing and annotating still images transmitted between two or more applications. This capability is often referred to as document conferencing or *shared whiteboarding.*

An important benefit of T.126 is that it readily shares visual information between applications that are running on dramatically different platforms. For example, a Windows-based desktop application could easily interoperate with a collaboration program running on a PowerMac. Similarly, a group-oriented conferencing system, without a PC-style interface, could share data with multiple users running common PC desktop software.

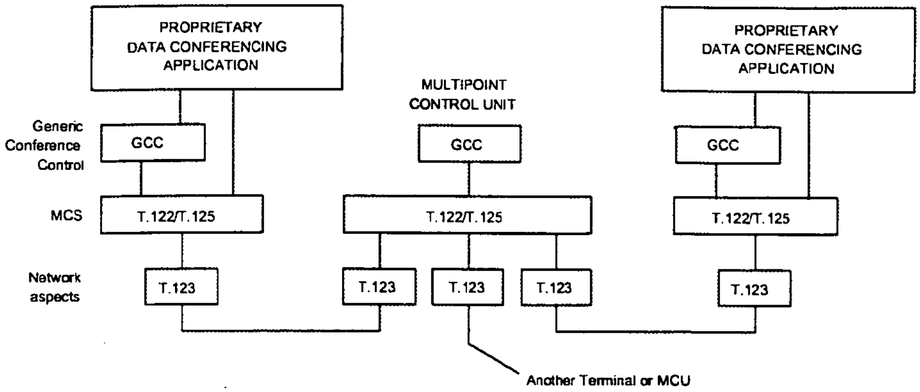
As Figure 6 illustrates, T.126 presents the concept of shared virtual *workspaces* that are manipulated by the endpoint applications. Each workspace may contain a collection of objects that include bitmap images and annotation primitives, such as rectangles and freehand lines. Bitmaps typically originate from application information, such as a word processing document or a presentation slide. Because of their size, bitmaps are often compressed to improve performance over lower-speed communication links.

T.126 is designed to provide a minimum set of capabilities required to share information between disparate applications. Because T.126 is simply a protocol, it does not provide any of the API-level structures that allow application developers to easily incorporate shared whiteboarding into an application. These types of facilities can only be found in toolkit-level implementations of the standard (such as DataBeam's Shared Whiteboard Application Toolkit, known as SWAT).

Multipoint Binary File Transfer - T.127

T.127 specifies a means for applications to transmit files between multiple endpoints in a conference. Files can be transferred to all participants in the conference or to a specified subset of the conference. Multiple file transfer operations may occur simultaneously in any given conference and developers can specify priority levels for the file delivery. Finally, T.127 provides options for compressing files before delivering the data. Figure 7 dis-

FIGURE 8: NETWORK-LEVEL INTEROPERABILITY DIAGRAM



plays a view of conference-wide and individual file transfers.

Node Controller

The Node Controller manages defined GCC Service Access Points (SAPs). This provides the node flexibility in responding to GCC events. Most of these GCC events relate to establishing conferences, adding or removing nodes from a conference, and breaking down and distributing information. The Node Controller's primary responsibility is to translate these events and respond appropriately.

Some GCC events can be handled automatically; for example, when a remote party joins a conference, each local Node Controller can post a simple message informing the local user that "Bill Smith has joined the conference." Other events may require user intervention; for example, when a remote party issues an invitation to join a conference, the local Node Controller posts a dialog box stating that "Mary Jones has invited you to the Design Review conference. <Accept> <Decline>."

Node controllers can be MCU-based, terminal-based, or dual-purpose. DataBeam's application, FarSite, for example, contains a dual-purpose Node Controller. The range of functionality found within a Node Controller can vary dramatically by implementation.

Only one Node Controller can exist on an active T.120 endpoint. Therefore, if multiple applications need to simultaneously use T.120 services, the Node Controller needs to be accessible to each application. The local interface to the Node Controller is application- and vendor-specific and is not detailed in the T.120 documentation.

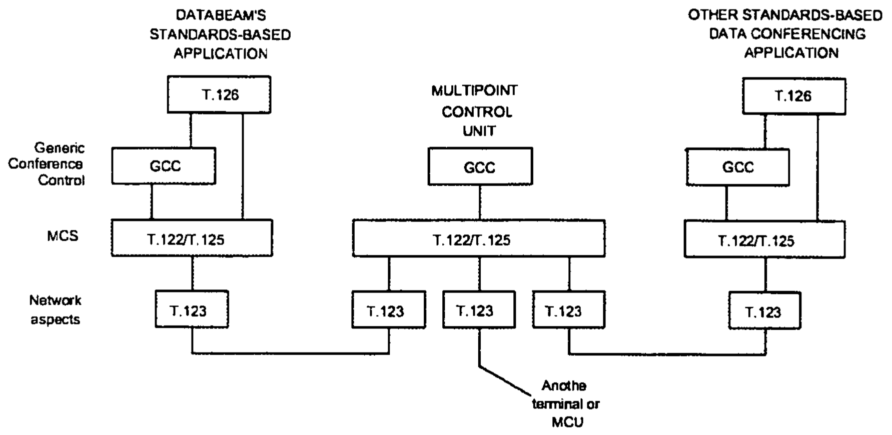
INTEROPERABILITY

Buyers overwhelmingly rate interoperability as the number one purchase criteria in their evaluation of teleconferencing products. For most end users, interoperability translates to "my application can talk to your application"—regardless of which vendor supplied the product or on what platform it runs. When examining the T.120 standard closely, buyers can see that it provides for two levels of interoperability: application-level interoperability and network-level interoperability.

Network-level Interoperability

Network-level interoperability means that a given product can interwork with like products through the infrastructure of

FIGURE 9: APPLICATION-LEVEL INTEROPERABILITY DIAGRAM


DATABEAM'S STANDARDS-BASED APPLICATION
OTHER STANDARDS-BASED DATA CONFERENCING APPLICATION
MULTIPOINT CONTROL UNIT
Generic Conference Control
MCS
Network aspects
T.126
T.126
GCC
GCC
GCC
T.122/T.125
T.122/T.125
T.122/T.125
T.123
T.123
T.123
T.123
T.123
Anothe terminal or MCU

network products and services that support T.120. For example, T.120-based conferencing bridges (MCUs) that can support hundreds of simultaneous users are now being developed. If an application supports only the lower layers of T.120, customers can use these MCUs to host a multipoint conference only if everyone in the conference is using the exact same product. Figure 8 displays network interoperability through a conference of *like products.*

Application-level Interoperability

The upper levels of T.120 specify protocols for common conferencing applications, such as shared whiteboarding and binary file transfer. Applications supporting these protocols can interoperate with any other application that provides similar support, regardless of the vendor or platform used. For example, through T.126, users of DataBeam's FarSite application will be able to share and mark up documents with users of group conferencing systems. This interoperability will exist in simple point-to-point conferences as well as large multipoint conferences using a conference bridge. Figure 9 represents application-level interoperability between two standards-based applications connected in a conference.

In the short-term, network-level interoperability will be the most common form of T.120 support found in conferencing applications. This is largely due to the fact that the lower-level T.120 layers were ratified by the ITU more than a year in advance of the application-level layers. However, end users will not be satisfied with network interoperability alone. For the market to grow, vendors will have to deliver the same application-level interoperability (or endpoint interoperability) that customers enjoy today with fax machines and telephones.

RATIFICATION OF THE T.120 AND FUTURE T.130 STANDARDS

The Recommendations for the core multipoint communications infrastructure components (T.122, T.123, T.124 and T.125) were ratified by the ITU between March of 1993 and March of 1995. The first of the application standards (T.126 and T.127) was approved in

March of 1995. An overview of the T.120 series was approved in February of 1996 as Recommendation T.120. T.121 (GAT) was also approved at that time. Stable drafts of these recommendations existed for some time prior to the ratification, thereby providing a means for DataBeam to actively develop products in parallel to the standardization effort.

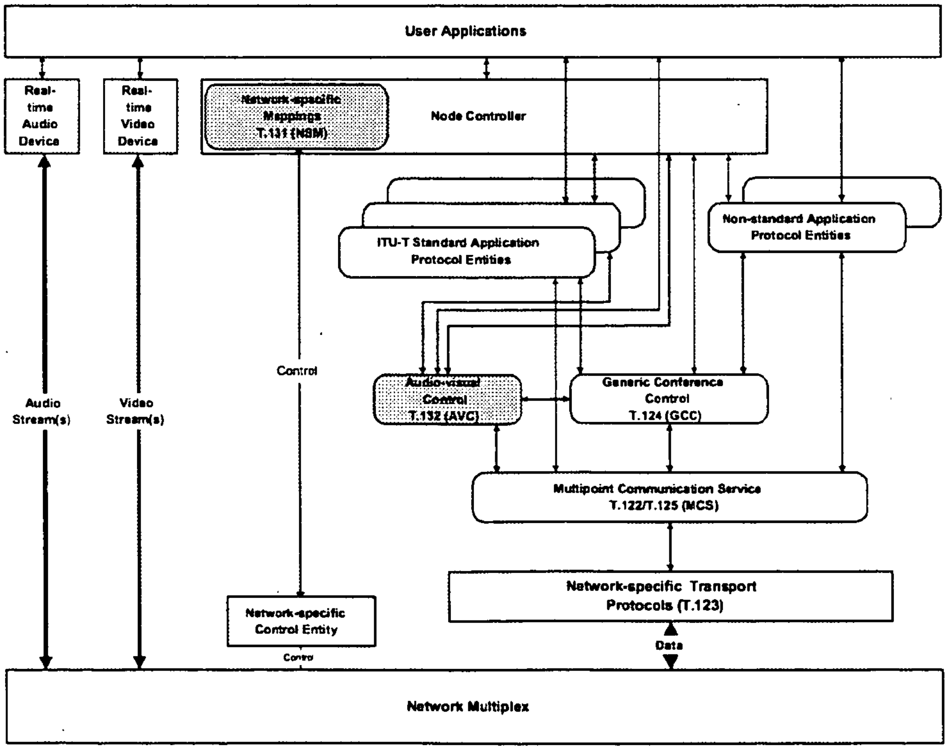
The existing ratified standards are being actively discussed for possible amendments and extensions. This commonly occurs when implementation and interoperability issues arise.

T.130 Audio-visual Control For Multimedia Conferencing

The T.130 series of recommendations define an architecture, a management and control protocol, and a set of services which together make up an Audio-Visual Control system (AVC). This system supports the use of real-time streams and services in a multimedia conferencing environment. The protocol and services section, outlined in T.132, consists of two parts: management and control. Together, they allow Network Elements, such as the traditional MCU, Gateway, or Conference Server, to provide T.132 audio and video services to their endpoints. Some of the services include Stream Identification, On-Air Identification, Video Switching, Audio Mixing, Remote Device Control, and Continuous Presence.

The T.130 series is built upon existing ITU-T conferencing recommendations such as the H.320 audio-visual conferencing series and the T.120 series for multipoint data conferencing. The T.130 series is compatible with systems, such as H.323, in which audio and video are

FIGURE 10: AUDIO-VISUAL CONTROL ARCHITECTURE


User Applications
Real-time Audio Device
Real-time Video Device
Network-specific Mappings T.131 (NSM)
Node Controller
ITU-T Standard Application Protocol Entities
Non-standard Application Protocol Entities
Control
Audio Stream(s)
Video Stream(s)
Audio-visual Control T.132 (AVC)
Generic Conference Control T.124 (GCC)
Multipoint Communication Service T.122/T.125 (MCS)
Network-specific Transport Protocols (T.123)
Network-specific Control Entity
Control
Data
Network Multiplex

transmitted independently of T.120, as well as systems which are capable of transmitting multiple media types within a common multiplex.

Unlike other standardized methods for managing real-time streams within a conference, T.130 provides some unique capabilities:

- Contains a network- and platform-independent control protocol for managing real-time streams
- Coordinates operations across network boundaries
- Processes and distributes media streams within a conference environment
- Delivers of Quality of Service (QoS) to multimediacommunications applications
- Provides distributed conference management
- Leverages the functionality of existing multimedia protocols

T.130 can be used in any conferencing scenario where there is a need for multipoint audio or video. T.130 relies upon the services of GCC and MCS to transmit control data, but the audio and video streams are transported in independent logical channels due to the transmission requirements of real-time data flows. (See Figure 10).

T.130 and T.132 were determined in March of 1997 and should be ratified in January of 1998. T.131, which defines network-specific mappings to allow AVC to communicate with the underlying Multimedia Control Protocol, such as H.245, should be determined in the Fall of 1997.

VENDOR COMMUNITY SUPPORT FOR T.120

More than 100 multinational companies have pledged their support for the T.120 standard and more are being added to this list every week. Public supporters of T.120 include international market leaders, such as Apple, AT&T, British Telecom, Cisco Systems, Deutsche Telecom, IBM, Intel, MCI, Microsoft, Motorola, PictureTel, and DataBeam.

Most supporters of T.120 are also members of the International Multimedia Teleconferencing Consortium (IMTC). The goals of the IMTC are to promote the awareness and adoption of ITU teleconferencing standards, including T.120 and H.32x. The IMTC provides a forum for interoperability testing and helps to define Application Programming Interfaces (APIs). DataBeam's co-founder and chief technical officer, C. J. "Neil" Starkey, serves as the president of the IMTC. Previously, Starkey served for six years as chairman of the ITU study group that defined T.120.

NEW MARKETS FOR T.120 DEPLOYMENT

The teleconferencing community is the first market segment to adopt the T.120 standard. Because the technology is broad in scope, it can be effectively used by a number of other application software vendors and equipment providers.

The computing paradigm is rapidly extending past today's personal productivity model. Over the next two years, we will witness the development of a new generation of application software that incorporates multi-party collaboration. Independent Software Vendors (ISVs) have begun to adopt T.120 as the means in which to incorporate real-time collaboration capabilities into common desktop applications, such as word processing and presentation graphics. Engineering products, such as Computer Aided Design (CAD) software, are also on the migration path to T.120 technology. Other ISVs with a strong interest in T.120 include developers of fax, remote control, document imaging, and "overtime" collaboration products, such as Lotus Notes.

With T.120 technology in the hands of operating system providers and horizontal application vendors, network equipment providers are beginning to take notice. For vendors of PBXs, network bridges, hubs, routers and switches, T.120 represents an important opportunity to provide value-added capabilities within their network products. In the short-term, these features will represent an opportunity for competitive advantage. However, within the next year, T.120 support will be a required feature.

Finally, we can envision a whole range of T.120 applications in the areas of interactive video, network gaming, and simulations. From Nintendo to DOOM to set-top boxes, the need for bidirectional multipoint data communications is acute. The ability to use a common set of APIs and protocols that are broadly supported from the desktop through the network will drive the adoption of T.120 into these important emerging markets.

IMTC, ITU, AND T.120

Standards have played an important part in the establishment and growth of several consumer and telecommunications markets. By creating a basic commonality, standards ensure compatibility among products from different manufacturers. This encourages companies to produce varying solutions and encourages end users to purchase the solutions without fear of obsolescence or incompatibility.

The work of both the IMTC and the ITU represents organized efforts to promote a basic connectivity protocol that will encourage the growth of the multimedia telecommunications market. The Standards First™ initiative, which is supported by many industry leaders, requires a minimum of H.320 and T.120 compliance, which is enough to establish this basic connectivityprotocol. Manufacturers are then able to build on the basic compliance by adding features to their products, creating Standards Plus equipment.

With Standards First, the IMTC has the end users' interests in mind. By ensuring interoperability among equipment from competing manufacturers, Standards First also ensures that a customer's initial investment is protected and future system upgrades are possible. The IMTC is helping to educate the industry and the public about the importance, function, and status of standards. In addition, the organization provides a coordination point for industry leaders to communicate their interests to the ITU-T. As the multipoint multimedia teleconferencing industry continues its rapid growth, the development and implementation of standards for interoperability, and the work of the IMTC, will be instrumental in securing the market's future.

IMPLEMENTING T.120

With the T.120 set of standards in place, third-party developers are faced with yet another challenge— implementation. DataBeam's Collaborative Computing Toolkit Series (CCTS™) has jump-started the conferencing industry by providing the first standards-based toolkits for developing multipoint, data-sharing applications. These toolkits encapsulate the complex system-wide, multipoint communications stacks that allow application developers to rapidly embed sophisticated real-time, data-sharing capabilities into new or existing products. Simply stated, CCTS provides a seamless solution for parties developing standards-based communication solutions.

As a result, DataBeam envisions an acceleration in the development of software applications and network infrastructure products such as, PBXs, bridges, routers, network switches, and LAN servers, that incorporate T.120. In addition, the industry will grow well beyond today's existing paradigms and the world will begin to see a whole range of new products and services that incorporate T.120. Users waiting for the standards dust to settle can now feel confident that with the support of vendors like Microsoft, DataBeam's T.120-based Collaborative Computing Toolkit Series is the best solution for industry-wide interoperability.

T.120 INFORMATION SOURCES

DataBeam Corporation
3191 Nicholasville Road
Lexington, Kentucky 40503
USA

Phone: (606) 245-3500
Fax: (606) 245-3528
E-Mail: info@databeam.com
Web Page: http://www.databeam.com

International Telecommunications Union
Sales Service
Place des Nations
CH-1211 Genéve 20
Switzerland

Phone: +41 22 730 6141
Fax: +41 22 730 5194
E-Mail: sales@itu.ch
Web Page: http://www.itu.ch

International Multimedia Teleconferencing Consortium, Inc.
111 Deerwood Road, Suite 372
San Ramon, California 94583
USA

Phone: (510) 743-4455
Fax: (510) 743-9011
E-Mail: dkamlani@imtc.fabrik.com
Web Page: http://www.imtc.org/imtc

DataBeam™

Copyright ©1995, 1996, 1997 DataBeam Corporation.
All Rights Reserved. Printed in the USA.

Updated May 14,1997.

This document may be reproduced, provided such reproduction is performed in its complete, unaltered form.

FarSite, CCTS, and DataBeam are registered trademarks of DataBeam Corporation. All other product and brand names are trademarks or registered trademarks of their respective holders.

Staff Publications

- Books
- Articles & White Papers
- Publication Archives
- Other Resources
  - Acronym List

New!
Online Classes
Click here to register

Hill Associates e-bulletin
A bimonthly newsletter
SUBSCRIBE

Staff Publications

An Overview of TCP/IP Protocols and the Internet

Gary C. Kessler
Hill Associates, Inc.
kumquat@hill.com

23 April 1999

This paper was originally submitted to the InterNIC, and posted on their Gopher site, on 5 August 1994. This document is an updated version of that paper.

Contents

1. Introduction
2. What are TCP/IP and the Internet?
    2.1. The Evolution of TCP/IP (and the Internet)
    2.2. Internet Growth
    2.3. Internet Administration
    2.4. Domain Names (and Politics)
3. The TCP/IP Protocol Architecture
    3.1. The Network Interface Layer
    3.2. The Internet Layer
        3.2.1. IP Addresses
        3.2.2. The Domain Name System
        3.2.3. ARP and Address Resolution
        3.2.4. IP Routing: OSPF, RIP, and BGP
        3.2.5. ICMP
        3.2.6. IP version 6
    3.3. The Transport Layer Protocols
        3.3.1. TCP

3.3.2. UDP
3.4. Applications
3.5. Summary
4. Other Information Sources
5. Acronyms and Abbreviations
6. Author's Address

---

1. Introduction

An increasing number of people are using the Internet and, many for the first time, are using the tools and utilities that at one time were only available on a limited number of computer systems (and only for really intense users!). One sign of this growth in use has been the significant number of TCP/IP and Internet books, articles, courses, and even TV shows that have become available in the last several years; there are so many such books that publishers are reluctant to authorize more because bookstores have reached their limit of shelf space! This memo provides a broad overview of the Internet and TCP/IP, with an emphasis on history, terms, and concepts. It is meant as a brief guide and starting point, referring to many other sources for more detailed information.

2. What are TCP/IP and the Internet?

While the TCP/IP protocols and the Internet *are* different, their histories are most definitely intertwingled! This section wil discuss some of the history. For additional information and insight, readers are urged to read two excellent histories of the Internet: *Casting The Net: From ARPANET to INTERNET and beyond...* by Peter Salus (Addison-Wesley, 1995) and *Where Wizards Stay Up Late: The Origins of the Internet* by Katie Hafner and Mark Lyon (Simon & Schuster, 1997).

2.1. The Evolution of TCP/IP (and the Internet)

Prior to the 1960s, what little computer communication existed comprised simple text and binary data, carried by the mos common telecommunications network technology of the day; namely, circuit switching, the technology of the telephone networks for nearly a hundred years. Because most data traffic is bursty in nature (i.e., most of the transmissions occur during a very short period of time), circuit switching results in highly inefficient use of network resources. In 1962, Paul Baran, of the Rand Corporation, described a robust, efficient, store-and-forward data network in a report for the U.S. Air Force; Donald Davies suggested a similar idea in independent work for the Postal Service in the U.K., and coined the term *packet* for the data units that would be carried. According to Baran and Davies, packet switching networks could be designed so that all components operated independently, eliminating single point-of-failure problems. In addition, networ communication resources appear to be dedicated to individual users but, in fact, statistical multiplexing and an upper lim on the size of a transmitted entity result in fast, economical data networks.

The modern Internet began as a U.S. Department of Defense (DoD) funded experiment to interconnect DoD-funded research sites in the U.S. In December 1968, the Advanced Research Projects Agency (ARPA) awarded a contract to design and deploy a packet switching network to Bolt Beranek and Newman (BBN). In September 1969, the first node of the ARPANET was installed at UCLA. With four nodes by the end of 1969, the ARPANET spanned the continental U.S. by 1971 and had connections to Europe by 1973.

The original ARPANET gave life to a number of protocols that were new to packet switching. One of the most lasting results of the ARPANET was the development of a user-network protocol that has become the standard interface between users and packet switched networks; namely, ITU-T (formerly CCITT) Recommendation X.25. This "standard" interface encouraged BBN to start Telenet, a commercial packet-switched data service, in 1974; after much renaming, Telenet is now a part of Sprint's X.25 service.

The initial host-to-host communications protocol introduced in the ARPANET was called the Network Control Protocol (NCP). Over time, however, NCP proved to be incapable of keeping up with the growing network traffic load. In 1974, a new, more robust suite of communications protocols was proposed and implemented throughout the ARPANET, based upon the Transmission Control Protocol (TCP) and Internet Protocol (IP). TCP and IP were originally envisioned functionally as a single protocol, thus the protocol suite, which actually refers to a large collection of protocols and applications, is usually referred to simply as *TCP/IP*. The original versions of both TCP and IP that are in common use today were written in September 1981, although both have had several modifications applied to them (in addition, the IP version 6, or IPv6, specification was released in December 1995). In 1983, the DoD mandated that all of their computer systems would use the TCP/IP protocol suite for long-haul communications, further enhancing the scope and importance of the ARPANET.

In 1983, the ARPANET was split into two components. One component, still called ARPANET, was used to interconnect research/development and academic sites; the other, called MILNET, was used to carry military traffic and became part of the Defense Data Network. That year also saw a huge boost in the popularity of TCP/IP with its inclusion in the communications kernel for the University of California s UNIX implementation, 4.2BSD (Berkeley Software Distribution) UNIX.

In 1986, the National Science Foundation (NSF) built a backbone network to interconnect four NSF-funded regional supercomputer centers and the National Center for Atmospheric Research (NCAR). This network, dubbed the NSFNET, was originally intended as a backbone for other networks, not as an interconnection mechanism for individual systems. Furthermore, the "Appropriate Use Policy" defined by the NSF limited traffic to non-commercial use. The NSFNET continued to grow and provide connectivity between both NSF-funded and non-NSF regional networks, eventually becoming the backbone that we know today as the Internet. Although early NSFNET applications were largely multiprotocol in nature, TCP/IP was employed for interconnectivity (with the ultimate goal of migration to Open Systems Interconnection).

The NSFNET originally comprised 56-kbps links and was completely upgraded to T1 (1.544 Mbps) links in 1989. Migration to a "professionally-managed" network was supervised by a consortium comprising Merit (a Michigan state regional network headquartered at the University of Michigan), IBM, and MCI. Advanced Network & Services, Inc. (ANS) a non-profit company formed by IBM and MCI, was responsible for managing the NSFNET and supervising the transition of the NSFNET backbone to T3 (44.736 Mbps) rates by the end of 1991. During this period of time, the NSF also funded a number of regional Internet service providers (ISPs) to provide local connection points for educational institutions and NSF-funded sites.

In 1993, the NSF decided that it did not want to be in the business of running and funding networks, but wanted instead to go back to the funding of research in the areas of supercomputing and high-speed communications. In addition, there was increased pressure to commercialize the Internet; in 1989, a trial gateway connected MCI, CompuServe, and Internet mail services, and commercial users were now finding out about all of the capabilities of the Internet that once belonged exclusively to academic and hard-core users! In 1991, the Commercial Internet Exchange (CIX) Association was formed by General Atomics, Performance Systems International (PSI), and UUNET Technologies to promote and provide a commercial Internet backbone service. Nevertheless, there remained intense pressure from non-NSF ISPs to open the network to all users.

In 1994, a plan was put in place to reduce the NSF's role in the public Internet. The new structure comprises three parts:

1. *Network Access Points (NAPs)*, where individual ISPs would interconnect. Although the NSF is only funding four such NAPs (Chicago, New York, San Francisco, and Washington, D.C.), several non-NSF NAPs are also in operation.
2. The *very High Speed Backbone Network Service*, a network interconnecting the NAPs and NSF-funded centers, operated by MCI. This network was installed in 1995 and operated at OC-3 (155.52 Mbps); it was completely upgraded to OC-12 (622.08 Mbps) in 1997.
3. The *Routing Arbiter*, to ensure adequate routing protocols for the Internet.

In addition, NSF-funded ISPs were given five years of reduced funding to become commercially self-sufficient. This funding ended by 1998.

In 1988, meanwhile, the DoD and most of the U.S. Government chose to adopt OSI protocols. TCP/IP was now viewed as an interim, proprietary solution since it ran only on limited hardware platforms and OSI products were only a couple of years away. The DoD mandated that all computer communications products would have to use OSI protocols by August 1990 and use of TCP/IP would be phased out. Subsequently, the U.S. Government OSI Profile (GOSIP) defined the set of protocols that would have to be supported by products sold to the federal government and TCP/IP was not included.

Despite this mandate, development of TCP/IP continued during the late 1980s as the Internet grew. TCP/IP developmen had always been carried out in an open environment (although the size of this open community was small due to the small number of ARPA/NSF sites), based upon the creed "We reject kings, presidents, and voting. We believe in rough consensus and running code" [*Dave Clark, M.I.T.*]. OSI products were still a couple of years away while TCP/IP became in the minds of many, the real open systems interconnection protocol suite.

It is not the purpose of this memo to take a position in the OSI vs. TCP/IP debate. Nevertheless, a number of observations are in order. First, the ISO Development Environment (ISODE) was developed in 1990 to provide an approach for OSI migration for the DoD. ISODE software allows OSI applications to operate over TCP/IP. During this same period, the Internet and OSI communities started to work together to bring about the best of both worlds as many TCP and IP features started to migrate into OSI protocols, particularly the OSI Transport Protocol class 4 (TP4) and the Connectionless Network Layer Protocol (CLNP), respectively. Finally, a report from the National Institute for Standards and Technology (NIST) in 1994 suggested that GOSIP should incorporate TCP/IP and drop the "OSI-only" requirement. [**NOTE:** Some industry observers have pointed out that OSI represents the ultimate example of a *sliding window*; OSI protocols have been "two years away" since about 1986.]

2.2. Internet Growth

The ARPANET started with four nodes in 1969 and grew to just under 600 nodes before it was split in 1983. The NSFNET also started with a modest number of sites in 1986. After that, the network has experienced literally exponentia growth. Internet growth between 1981 and 1991 is documented in "Internet Growth (1981-1991)" (RFC 1296).

Network Wizard's distributes a semi-annual Internet Domain Survey. According to them, the Internet had nearly 30 millio reachable hosts by January 1998. The Internet is growing at a rate of about a new network attachment every half-hour, interconnecting more than 200,000 networks. It is estimated that the Internet is doubling in size every ten to twelve months, and has been for the last several years.

And what of the original ARPANET? It grew smaller and smaller during the late 1980s as sites and traffic moved to the Internet, and was decommissioned in July 1990. Cerf & Kahn ("Selected ARPANET Maps," *Computer Communications Review*, October 1990) re-printed a number of network maps documenting the growth (and demise) of the ARPANET.

2.3. Internet Administration

The Internet has no single owner, yet everyone owns (a portion of) the Internet. The Internet has no central operator, yet everyone operates (a portion of) the Internet. The Internet has been compared to anarchy, but some claim that it is not nearly that well organized!

Some central authority is required for the Internet, however, to manage those things that can only be managed centrally, such as addressing, naming, protocol development, standardization, etc. Among the significant Internet authorities are:

- The Internet Society (ISOC), chartered in 1992, is a non-governmental international organization providing coordination for the Internet, and its internetworking technologies and applications. ISOC also provides oversight and communications for the Internet Activities Board.
- The Internet Activities Board (IAB) governs administrative and technical activities on the Internet.
- The Internet Engineering Task Force (IETF) is one of the two primary bodies of the IAB. The IETF's working groups have primary responsibility for the technical activities of the Internet, including writing specifications and protocols. The impact of these specifications is significant enough that ISO accredited the IETF as an internationa standards body at the end of 1994. RFCs 2028 and 2031 describe the organizations involved in the IETF standards process and the relationship between the IETF and ISOC, respectively, while RFC 2418 describes the IETF working group guidelines and procedures. The background and history of the IETF and the Internet standards process can be found in "IETF—History, Background, and Role in Today's Internet."
- The Internet Engineering Steering Group (IESG) is the other body of the IAB. The IESG provides direction to the IETF.
- The Internet Research Task Force (IRTF) comprises a number of long-term reassert groups, promoting research of importance to the evolution of the future Internet.
- The Internet Engineering Planning Group (IEPG) coordinates worldwide Internet operations. This group also assists Internet Service Providers (ISPs) to interoperate within the global Internet.
- The Forum of Incident Response and Security Teams is the coordinator of a number of Computer Emergency Response Teams (CERTs) representing many countries, governmental agencies, and ISPs throughout the world. Internet network security is greatly enhanced and facilitated by the FIRST member organizations.

2.4. Domain Names (and Politics)

Although not directly related to the administration of the Internet for operational purposes, the assignment of Internet domain names is the subject of some controversy and current activity. Internet hosts use a hierarchical naming structure comprising a top-level domain (TLD), domain and subdomain (optional), and host name. The IP address space (and all TCP/IP-related numbers) has historically been managed by the Internet Assigned Numbers Authority (IANA). Domain names are assigned by the TLD naming authority; until April 1998, the Internet Network Information Center (InterNIC) had overall authority of these names, with NICs around the world handling non-U.S. domains. The InterNIC was also responsible for the overall coordination and management of the Domain Name System (DNS), the distributed database that reconciles host names and IP addresses on the Internet.

The InterNIC is an interesting example of changes in the Internet. Starting in 1993, Network Solutions, Inc. (NSI) operated the InterNIC on behalf of the NSF and had exclusive registration authority for the *.com*, *.org*, *.net*, and *.edu* domains. NSI's contract ran out in April 1998 and was extended several times while everyone tried to determine who should pick up the registration for those domains. In October 1998, it was decided that NSI will remain the sole administrator for those domains but that users could register names in those domains with other firms. In addition, NSI's contract was extended to September 2000, although the registration business has to be opened to competition by June 1999.

Meanwhile, the newest body to handle gTLD registrations is the Internet Corporation for Assigned Names and Numbers (ICANN). Formed in October 1998, ICANN is the organization designated by the U.S. National Telecommunications and Information Administration (NTIA) to administer the DNS. Although still surrounded in some controversy (which is well beyond the scope of this paper!), ICANN has received wide industry support. ICANN will form several Support Organizations (SOs) to create policy for the administration of its areas of responsibility, including domain names (DNSO) IP addresses (ASO), and protocol parameter assignments (PSO).

On April 21, 1999, ICANN announced that five companies had been selected to be part of this new *competitive* Shared Registry System for the .com, .net, and .org domains:

- America Online, Inc. (U.S.)
- CORE (Internet Council of Registrars) (International)
- France Telecom/Oléane (France)
- Melbourne IT (Australia)
- register.com (U.S.)

Phase I of the competitive registrar testbed program will run until June, 1999. At that time, the Shared Registry System for the .com, .net, and .org domains will be opened to all ICANN-accredited registrars. ICANN also announced a list of 29 other applicant companies that had met its accreditation standards and will be able to enter the market as a registrar afte Phase I:

- 9 Net Avenue, Inc. (U.S.)
- A Technology Company (Canada)
- Active ISP (Norway)
- All West Communications (U.S.)
- Alldomains.com (U.S.)
- American Domain Name Registry (U.S.)
- AT&T (U.S.)
- Domain Direct (Canada)
- DomainRegistry.com (U.S.)
- eNom, Inc. (U.S.)
- Info Avenue Internet Services (U.S.)
- InfoNetworks (USA & United Kingdom)
- InfoRamp, Inc. (U.S.)
- Interactive Telecom Network, Inc. (U.S.)
- Interdomain, S.A. (Spain)
- Internet Domain Registrars (Canada)
- interQ Incorporated (Japan)
- MS Intergate, Inc. (U.S.)
- NameSecure.com (U.S.)
- Name.Space (U.S.)
- NetBenefit (United Kingdom)
- NetNames (United Kingdom)
- Nominalia (Catalonia)
- Port Information System (Sweden)
- RCN Corporation (U.S.)
- TelePartner AS (Denmark)
- Verio (U.S.)
- Virtual Internet (United Kingdom)
- WebTrends Corporation (U.S.)

The domain name structure is best understood if the name is read from right-to-left. Internet hosts names end with a top-level domain name. World-wide generic top-level domains include:

- *.com*: Commercial organizations (administered by the Shared Registry)
- *.edu*: Educational institutions, although today usually limited to 4-year colleges and universities (administered by the InterNIC)
- *.net*: Network providers (administered by the InterNIC and the Shared Registry)
- *.org*: Non-profit organizations (administered by the InterNIC and the Shared Registry)
- *.int*: Organizations established by international treaty
- *.gov*: U.S. Federal government agencies (delegated to the U.S. Federal Networking Council and administered by the InterNIC)

- *.mil*: U.S. military (managed by the U.S. Defense Data Network)

The host name *entc.tamu.edu*, for example, is assigned to a computer in the Engineering Technology and Industrial Distribution (ETID) Department at Texas A&M University (*tamu*), within the educational top-level domain (*edu*). The host name *golem.hill.com* refers to a host (*golem*) in the Hill Associates domain (*hill*) within the commercial top-level domain (*com*). Guidelines for selecting host names is the subject of RFC 1178.

Other top-level domain names use the two-letter country codes defined in ISO standard 3166; *munnari.oz.au*, for example, is the address of the Internet gateway to Australia and *myo.inst.keio.ac.jp* is a host at the Science and Technology Department of Keio University in Yokohama, Japan. Other ISO 3166-based domain country codes are *ca* (Canada), *de* (Germany), *es* (Spain), *fr* (France), *gb* (Great Britain) [**NOTE:** For some historical reasons, the TLD *.gb* is rarely used; the TLD *.uk* (United Kingdom) seems to be preferred although UK is not an official ISO 3166 country code.], *il* (Israel), *ie* (Ireland), *jp* (Japan), *mx* (Mexico), and *us* (United States). It is important to note that there is not necessarily any correlation between a country code and where a host is actually physically located.

The Western Hemisphere, European, and Asia-Pacific naming registries are managed by the American Registry for Internet Numbers (ARIN), RIPE, and Asia-Pacific NIC (APNIC), respectively. These authorities, in turn, delegate most of the country TLDs to national registries (such as RNP in Brazil and NIC-Mexico), which have ultimate authority to assign local domain names.

Different countries may organize the country-based subdomains in any way that they want. Many countries use a subdomain similar to the TLDs, so that *.com.mx* and *.edu.mx* are the suffixes for commercial and educational institutions in Mexico, and *.co.uk* and *.ac.uk* are the suffixes for commercial and educational institutions in the United Kingdom.

The *us* domain is largely organized on the basis of geography or function. Geographical names in the *us* name space us names of the form *entity-name.city-telegraph-code.state-postal-code*.us. The domain name *cnri.reston.va.us*, for example, refers to the Corporation for National Research Initiatives in Reston, Virginia. Functional branches are also reserved within the name space for schools (K12), community colleges (CC), technical schools (TEC), state government agencies (STATE), councils of governments (COG), libraries (LIB), museums (MUS), and several other generic types of entities. Domain names in the state government name space usually take the form *department*.state.*state-postal-code*.u (e.g., the domain name *dps.state.vt.us* points to the Vermont Department of Public Safety). The K12 name space can vary widely, usually using the form *school.school-district*.k12.*state-postal-code*.us (e.g., the domain *ccs.cssd.k12.vt.us* refers to the Charlotte Central School in the Chittenden South School District in Charlotte, Vermont.) More information about the *us* domain may be found in RFC 1480.

The scheme of TLD assignment and management has worked well for many years, but the pressures of increased commercial activity, network size, and international use have caused controversy about how names can be fairly assigned without violating trademarks and conflicting claims to names. In November 1996, an Internet International Ad Hoc Committee (IAHC) was formed to resolve some of these naming issues and to act as a focal point for the international debate over a proposal to establish additional global naming registries and global Top Level Domains (gTLDs). In February 1997, the IAHC proposed the creation of seven new gTLDs:

- *.firm* for businesses, or firms.
- *.store* for businesses offering goods to purchase.
- *.web* for entities emphasizing activities related to the WWW.
- *.arts* for entities emphasizing cultural and entertainment activities.
- *.rec* for entities emphasizing recreation/entertainment activities.
- *.info* for entities providing information services.

- *.nom* for those wishing individual or personal nomenclature.

The IAHC also proposed that up to 28 new registrars be established to grant second-level domain names under the new gTLDs, all of which will be shared among the new registrars. Furthermore, the three existing gTLDs .com, .net, and .org were also be shared upon conclusion of the NSF contract in the U.S. in 1998.

The IAHC was dissolved in May 1997 with the publication of the Generic Top Level Domain Memorandum of Understanding framework. The Council of Registrars (CORE) an operational body made up of all of the Registrars established under the gTLD-MoU framework.

3. The TCP/IP Protocol Architecture

TCP/IP is most commonly associated with the Unix operating system. While developed separately, they have been historically tied, as mentioned above, since 4.2BSD Unix started bundling TCP/IP protocols with the operating system. Nevertheless, TCP/IP protocols are available for all widely-used operating systems today and native TCP/IP support is provided in OS/2, OS/400, and Windows 95/98/NT, as well as most Unix variants.

Figure 1 shows the TCP/IP protocol architecture; this diagram is by no means exhaustive, but shows the major protocol and application components common to most commercial TCP/IP software packages and their relationship.

```
            ---------------------------------------------------------  ------
APPLICATION |Telnet|FTP|Gopher|SMTP|HTTP|BGP|Finger|POP|DNS|SNMP|RIP|  |Ping|
            |------+---+------+----+----+---+------+---+-+-+----+---|  |----+-----
  TRANSPORT |                     TCP                    |    UDP    |  |ICMP|OSPF|
            |--------------------------------------------+----------+--+----+----+----
   INTERNET |                              IP                                    |ARP|
            |----------+-------+----+------+-------+------+------+-----+------+--+---|
    NETWORK | Ethernet | Token |FDDI| X.25 | Frame | SMDS | ISDN | ATM | SLIP | PPP  |
  INTERFACE |          | Ring  |    |      | Relay |      |      |     |      |      |
            --------------------------------------------------------------------------
```

FIGURE 1. Simplified TCP/IP protocol stack.

The sections below will provide a brief overview of each of the layers in the TCP/IP suite and the protocols that compose those layers. A large number of books and papers have been written that describe all aspects of TCP/IP as a protocol suite, including detailed information about use and implementation of the protocols. Readers are referred to *Internetworking with TCP/IP, Vol. I: Principles, Protocols, and Architecture*, 2/e, by D. Comer (Prentice-Hall, 1991), *TCP/IP: Architecture, Protocols, and Implementation with IPv6 and IP Security*, 2nd. ed. by S. Feit (McGraw-Hill, 1997), "TCP/IP Tutorial" by T.J. Socolofsky and C.J. Kale (RFC 1180), and *TCP/IP Illustrated, Volume I: The Protocols* by W.R. Stevens (Addison-Wesley, 1994).

3.1. The Network Interface Layer

The TCP/IP protocols have been designed to operate over nearly any underlying local or wide area network technology. Although certain accommodations may need to be made, IP messages can be transported over all of the technologies shown in the figure, as well as numerous others.

Two of the underlying interface protocols are particularly relevant to TCP/IP. The Serial Line Internet Protocol (SLIP, RFC 1055) and Point-to-Point Protocol (PPP, RFC 1661), respectively, may be used to provide data link layer protocol services where no other underlying data link protocol may be in use, such as in leased line or dial-up environments. Mos commercial TCP/IP software packages for PC-class systems include these two protocols. With SLIP or PPP, a remote computer can attach directly to a host server and, therefore, connect to the Internet using IP rather than being limited to an asynchronous connection. PPP, in addition, provides support for simultaneous multiple protocols over a single connection (see the IANA list of PPP protocols), security mechanisms, and dynamic bandwidth allocation (e.g., when running over ISDN).

3.2. The Internet Layer

The Internet Protocol (RFC 791), provides services that are roughly equivalent to the OSI Network Layer. IP provides a datagram (connectionless) transport service across the network. This service is sometimes referred to as *unreliable* because the network does not guarantee delivery nor notify the end host system about packets lost due to errors or network congestion. IP datagrams contain a message, or one fragment of a message, that may be up to 65,535 bytes (octets) in length. IP does not provide a mechanism for flow control.

```
                     1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |      TOS      |          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|     Fragment Offset     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      TTL      |   Protocol    |        Header Checksum        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Source Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Destination Address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Options....                               (Padding)   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Data...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

FIGURE 2. IP packet (datagram) header format.

The basic IP packet header format is shown in Figure 2. The format of the diagram is consistent with the RFC; bits are numbered from left-to-right, starting at 0. Each row represents a single *32-bit word*; note that an IP header will be at leas 5 words (20 bytes) in length. The fields contained in the header, and their functions, are:

- *Version:* Specifies the IP version of the packet. The current version of IP is version 4, so this field will contain the binary value `0100`. [NOTE: Actually, many IP version numbers have been assigned besides 4 and 6; see the IANA's list of IP Version Numbers.]
- *Internet Header Length (IHL):* Indicates the length of the datagram header in 32 bit (4 octet) words. A minimum-length header is 20 octets, so this field always has a value of at least 5 (`0101`).
- *Type of Service (TOS):* Allows an originating host to request different classes of service for packets it transmits. Although not generally supported today in IPv4, the TOS field can be set by the originating host in response to service requests across the Transport Layer/Internet Layer service interface, and can specify a service priority (0-7) or can request that the route be optimized for either cost, delay, throughput, or reliability.
- *Total Length:* Indicates the length (in bytes, or octets) of the entire packet, including both header and data. Given the size of this field, the maximum size of an IP packet is 64 KB, or 65,535 bytes. In practice, packet sizes are limited to the maximum transmission unit (MTU).
- *Identification:* Used when a packet is fragmented into smaller pieces while traversing the Internet, this identifier is assigned by the transmitting host so that different fragments arriving at the destination can be associated with each other for reassembly.
- *Flags:* Also used for fragmentation and reassembly. The first bit is called the More Fragments (MF) bit, and is use to indicate the last fragment of a packet so that the receiver knows that the packet can be reassembled. The second bit is the Don't Fragment (DF) bit, which suppresses fragmentation. The third bit is unused (and always se to 0).
- *Fragment Offset:* Indicates the position of this fragment in the original packet. In the first packet of a fragment stream, the offset will be 0; in subsequent fragments, this field will indicates the offset in increments of 8 bytes.
- *Time-to-Live (TTL):* A value from 0 to 255, indicating the number of hops that this packet is allowed to take before discarded within the network. Every router that sees this packet will decrement the TTL value by one; if it gets to 0 the packet will be discarded.
- *Protocol:* Indicates the higher layer protocol contents of the data carried in the packet; options include ICMP (1), TCP (6), UDP (17), or OSPF (89). A complete list of IP protocol numbers can be found at the IANA's list of Protocol Numbers.

- *Header Checksum:* Carries information to ensure that the received IP header is error-free. Remember that IP provides an *unreliable* service and, therefore, this field only checks the header rather than the entire packet.
- *Source Address:* IP address of the host sending the packet.
- *Destination Address:* IP address of the host intended to receive the packet.
- *Options:* A set of options which may be applied to any given packet, such as sender-specified source routing or security indication. The option list may use up to 40 bytes (10 words), and will be padded to a word boundary; IP options are taken from the IANA's list of IP Option Numbers.

3.2.1. IP Addresses

IP addresses are 32 bits in length (Figure 3). They are typically written as a sequence of four numbers, representing the decimal value of each of the address bytes. Since the values are separated by periods, the notation is referred to as *dotted decimal*. A sample IP address is 208.162.106.17.

```
                              1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
          0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
         --+-------------+------------------------------------------------
Class A  |0|   NET_ID    |                    HOST_ID                    |
         |-+-+-----------+---------------+-------------------------------|
Class B  |1|0|  NET_ID                   |            HOST_ID            |
         |-+-+-+-------------------------+---------------+---------------|
Class C  |1|1|0|                 NET_ID                  |    HOST_ID    |
         |-+-+-+-+---------------------------------------+---------------|
Class D  |1|1|1|0|                     MULTICAST_ID                      |
         |-+-+-+-+-------------------------------------------------------|
Class E  |1|1|1|1|               EXPERIMENTAL_ID                         |
         --+-+-+-+--------------------------------------------------------
```

FIGURE 3. IP Address Format.

IP addresses are hierarchical for routing purposes and are subdivided into two subfields. The Network Identifier (NET_ID subfield identifies the TCP/IP subnetwork connected to the Internet. The NET_ID is used for high-level routing between networks, much the same way as the country code, city code, or area code is used in the telephone network. The Host Identifier (HOST_ID) subfield indicates the specific host within a subnetwork.

To accommodate different size networks, IP defines several *address classes*. Classes A, B, and C are used for host addressing and the only difference between the classes is the length of the NET_ID subfield:

- A Class A address has a 7-bit NET_ID and 24-bit HOST_ID. Class A addresses are intended for very large networks and can address up to 16,777,216 (2^{24}) hosts per network. The first digit of a Class A addresses will be number between 1 and 126. Relatively few Class A addresses have been assigned; examples include 4.0.0.0 (BBN Planet) and 9.0.0.0 (IBM).
- A Class B address has a 14-bit NET_ID and 16-bit HOST_ID. Class B addresses are intended for moderate sized networks and can address up to 65,536 (2^{16}) hosts per network. The first digit of a Class B address will be a number between 128 and 191. The Class B address space has long been threatened with being used up and it is has been very difficult to get a new Class B address for some time. Class B address assignment examples includ 128.138.0.0 (WestNet) and 152.163.0.0 (America Online).
- A Class C address has a 21-bit NET_ID and 8-bit HOST_ID. These addresses are intended for small networks and can address only up to 254 (2^8-2) hosts per network. The first digit of a Class C address will be a number between 192 and 223. Most addresses assigned to networks today are Class C (or sub-Class C!); examples include 208.162.102.0 (Hill Associates) and 209.198.87.0 (SoverNet, Bellows Falls, VT).

The remaining two address classes are used for special functions only and are not commonly assigned to individual hosts. Class D addresses may begin with a value between 224 and 239, and are used for IP multicasting (i.e., sending a single datagram to multiple hosts); the IANA maintains a list of Internet Multicast Addrsses. Class E addresses begin wit a value between 240 and 255, and are reserved for experimental use.

Several address values are reserved and/or have special meaning. A HOST_ID of 0 (as used above) is a dummy value reserved as a place holder when referring to an entire subnetwork; the address 208.162.106.0, then, refers to the Class C address with a NET_ID of 208.162.106. A HOST_ID of all ones (usually written "255" when referring to an all-ones byte, but also denoted as "-1") is a broadcast address and refers to all hosts on a network. A NET_ID value of 127 is used for loopback testing and the specific host address 127.0.0.1 refers to the *localhost*.

Several NET_IDs have been reserved in RFC 1918 for private network addresses and packets will not be routed over th Internet to these networks. Reserved NET_IDs are the Class A address 10.0.0.0 (formerly assigned to ARPANET), the sixteen Class B addresses 172.16.0.0-172.31.0.0, and the 256 Class C addresses 192.168.0.0-192.168.255.0.

An additional addressing tool is the *subnet mask*. Subnet masks are used to indicate the portion of the address that identifies the network (and/or subnetwork) for routing purposes. The subnet mask is written in dotted decimal and the number of 1s indicates the significant NET_ID bits. For "classful" IP addresses, the subnet mask and number of significant address bits for the NET_ID are:

Class	**Subnet Mask**	**Number of Bits**
A	255.0.0.0	8
B	255.255.0.0	16
C	255.255.255.0	24

Depending upon the context and literature, subnet masks may be written in dotted decimal form or just as a number representing the number of significant address bits for the NET_ID. Thus, `208.162.106.17 255.255.255.0` and `208.162.106.17/24` both refer to a Class C NET_ID of 208.162.106.

Subnet masks can also be used to subdivide a large address space or to combine multiple small address spaces. For example, a network may subdivide their address space to define multiple logical networks by segmenting the HOST_ID subfield into a Subnetwork Identifier (SUBNET_ID) and (smaller) HOST_ID. For example, a user might be assigned the Class B address space 172.16.0.0 which might be segmented into a 16-bit NET_ID, 4-bit SUBNET_ID, and 12-bit HOST_ID. In this case, the subnet mask for routing to the NET_ID on the Internet would be 255.255.0.0 (or "/16"), while the mask for routing to individual subnets within the larger Class B address space would be 255.255.240.0 (or "/20").

Alternatively, a single user might be assigned the four Class C addresses 192.168.128.0, 192.168.129.0, 192.168.130.0 and 192.168.131.0, and use the subnet mask 255.255.252.0 (or "/22") for routing to this domain. This use of subnet masks in routing tables to consolidate addresses uses a process called *Classless Interdomain Routing (CIDR)*, describe in RFCs 1518 and 1519. It should be obvious from this example that CIDR address consolidation results in smaller route tables; in the example here, routing information for four Class C addresses can be specified in a single router table entry

As of January 1996, there were 95 Class A addresses, 5892 Class B addresses, and 128,378 Class C addresses assigned; this number is undoubtedly larger today, particularly in the Class C space. Because CIDR is becoming so widely used, however, these numbers are not a true reflection of the number of networks attached to the public Internet because multiple addresses may be assigned to a single organizational entity.

3.2.2. The Domain Name System

While IP addresses are 32 bits in length, most users do not memorize the numeric addresses of the hosts to which they attach; instead, people are more comfortable with host names. Most IP hosts, then, have both a numeric IP address and a name. While this is convenient for people, however, the name must be translated back to a numeric address for routing purposes.

Earlier discussion in this paper described the domain naming structure of the Internet. In the early ARPANET, every host maintained a file called `HOSTS.TXT` that contained a list of all hosts, which included the IP address, host name, and alias (es). This was an adequate measure while the ARPANET was small and had a slow rate of growth, but was not a scalable solution as the network grew.

[NOTE: `HOSTS.TXT` files are still found on Unix systems although usually used to reconcile names of hosts on the local network to cut down on local DNS traffic. On Microsoft Windows systems, the file is called `HOSTS` and can typically be found in the `c:\windows` folder.]

To handle the fast rate of new names on the network, the Domain Name System (DNS) was created. The DNS is a distributed database containing host name and IP address information for all domains on the Internet. There is a single *authoritative name server* for every domain that contains all DNS-related information about the domain; each domain als has at least one secondary name server that also contains a copy of this information. Thirteen *root servers* around the globe (most in the U.S., actually, with the remainder in Asia and Europe) maintain a list of all of these authoritative name servers.

When a host on the Internet needs to obtain a host's IP address based upon the host's name, a DNS request is made by the initial host to the to a local name server. The local name server may be able to respond to the request with information that is either configured or cached at the name server; if necessary information is not available, the local name server forwards the request to one of the root servers. The root server, then, will determine an appropriate name server for the target host and the DNS request will be forwarded to the domain's name server.

Name servers contain the following types of information:

- *A-record:* An address record maps a hostname to an IP address.
- *PTR-record:* A pointer record maps an IP address to a hostname.
- *NS-record:* A name server record lists the authoritative name server(s) for a given domain.
- *MX-record:* A mail exchange record lists the mail servers for a given domain. As an example, consider the author' e-mail address, *kumquat@hill.com*. Note that the "hill.com" portion of the address is a domain name, not a host name, and mail has to be sent to a specific host. The MX-records in the *hill.com* name database specifies the hos *mail.hill.com* is the mail server for this domain.

More information about the DNS can be found from the World Internetworking Alliance (WIA) Web site. Additional DNS references include *DNS and BIND* by P. Albitz and C. Liu (O'Reilly & Associates) and "Setting up Your own DNS" by G. Kessler. The concepts, structure, and delegation of the DNS are described in RFCs 1034 and 1591. In addition, the IANA maintains a list of DNS parameters.

3.2.3. ARP and Address Resolution

Early IP implementations ran on hosts commonly interconnected by Ethernet local area networks (LAN). Every transmission on the LAN contains the local network, or medium access control (MAC), address of the source and destination nodes. MAC addresses are 48-bits in length and are non-hierarchical, so routing cannot be performed using the MAC address. MAC addresses are never the same as IP addresses.

When a host needs to send a datagram to another host on the same network, the sending application must know both the IP and MAC addresses of the intended receiver; this is because the destination IP address is placed in the IP packet and the destination MAC address is placed in the LAN MAC protocol frame. (If the destination host is on another network the sender will look instead for the MAC address of the default gateway, or router.)

Unfortunately, the sender's IP process may not know the MAC address of the intended receiver on the same network. The Address Resolution Protocol (ARP), described in RFC 826, provides a mechanism so that a host can learn a receiver's MAC address when knowing only the IP address. The process is actually relatively simple: the host sends an ARP Request packet in a frame containing the MAC broadcast address; the ARP request advertises the destination IP address and asks for the associated MAC address. The station on the LAN that recognizes its own IP address will send an ARP Response with its own MAC address. As Figure 1 shows, ARP message are carried directly in the LAN frame and ARP is an independent protocol from IP. The IANA maintains a list of all ARP parameters.

Other address resolution procedures have also been defined, including:

- Reverse ARP (RARP), which allows a disk-less processor to determine its IP address based on knowing its own MAC address
- Inverse ARP (InARP), which provides a mapping between an IP address and a frame relay virtual circuit identifier
- ATMARP and ATMInARP provide a mapping between an IP address and ATM virtual path/channel identifiers.
- LAN Emulation ARP (LEARP), which maps a recipient's ATM address to its LAN Emulation (LE) address (which takes the form of an IEEE 802 MAC address).

[NOTE: IP hosts maintain a cache storing recent ARP information. The ARP cache can be viewed from a Unix or DOS (i Windows 95/98/NT) command line using the `arp -a` command.]

3.2.4. IP Routing: OSPF, RIP, and BGP

As an OSI Network Layer protocol, IP has the responsibility to route packets. It performs this function by looking up a packet's destination IP NET_ID in a routing table and forwarding based on the information in the table. But it is *routing protocols*, and *not* IP, that populate the routing tables with routing information. There are three routing protocols commonly associated with IP and the Internet, namely, RIP, OSPF, and BGP.

OSPF and RIP are primarily used to provide routing within a particular domain, such as within a corporate network or within an ISP's network. Since the routing is *inside* of the domain, these protocols are generically referred to as *interior gateways protocols*.

The Routing Information Protocol version 2 (RIP-2), described in RFC 2453, describes how routers will exchange routing table information using a distance-vector algorithm. With RIP, neighboring routers periodically exchange their entire routing tables. RIP uses hop count as the metric of a path's cost, and a path is limited to 16 hops. Unfortunately, RIP has become increasingly inefficient on the Internet as the network continues its fast rate of growth. Current routing protocols for many of today's LANs are based upon RIP, including those associated with NetWare, AppleTalk, VINES, and DECne The IANA maintains a list of RIP message types.

The Open Shortest Path First (OSPF) protocol is a link state routing algorithm that is more robust than RIP, converges faster, requires less network bandwidth, and is better able to scale to larger networks. With OSPF, a router broadcasts only changes in its links' status rather than entire routing tables. OSPF Version 2, described in RFC 1583, is rapidly replacing RIP in the Internet.

The Border Gateway Protocol version 4 (BGP-4) is an *exterior gateway protocol* because it is used to provide routing information between Internet routing domains. BGP is a distance vector protocol, like RIP, but unlike almost all other distance vector protocols, BGP tables store the actual route to the destination network. BGP-4 also supports policy-base routing, which allows a network's administrator to create routing policies based on political, security, legal, or economic issues rather than technical ones. BGP-4 also supports CIDR. BGP-4 is described in RFC 1771, while RFC 1268 describes use of BGP in the Internet. In addition, the IANA maintains a list of BGP parameters.

Figure 1 shows the protocol relationship of RIP, OSPF, and BGP to IP. A RIP message is carried in a UDP datagram which, in turn, is carried in an IP packet. An OSPF message, on the other hand, is carried directly in an IP datagram. BGP messages, in a total departure, are carried in TCP segments over IP. Although all of the TCP/IP books mentioned above discuss IP routing to some level of detail, *Routing in the Internet* by Christian Huitema is one of the best available references on this specific subject.

3.2.5. ICMP

The Internet Control Message Protocol, described in RFC 792, is an adjunct to IP that notifies the sender of IP datagram about abnormal events. This collateral protocol is particularly important in the connectionless environment of IP.

The commonly employed ICMP message types include:

- *Destination Unreachable:* Indicates that a packet cannot be delivered because the destination host cannot be reached. The reason for the non-delivery may be that the host or network is unreachable or unknown, the protoco or port is unknown or unusable, fragmentation is required but not allowed (DF-flag is set), or the network or host is unreachable for this type of service.
- *Echo* and *Echo Reply:* These two messages are used to check whether hosts are reachable on the network. One host sends an Echo message to the other, optionally containing some data, and the receiving host responds with an Echo Reply containing the same data. These messages are the basis for the Ping command.
- *Parameter Problem:* Indicates that a router or host encountered a problem with some aspect of the packet's

Header.

- *Redirect:* Used by a host or router to let the sending host know that packets should be forwarded to another address. *For security reasons, Redirect messages should usually be blocked at the firewall.*
- *Source Quench:* Sent by a router to indicate that it is experiencing congestion (usually due to limited buffer space and is discarding datagrams.
- *TTL Exceeded:* Indicates that a datagram has been discarded because the TTL field reached 0 or because the entire packet was not received before the fragmentation timer expired.
- *Timestamp* and *Timestamp Reply:* These messages are similar to the Echo messages, but place a timestamp (with millisecond granularity) in the message, yielding a measure of how long remote systems spend buffering and processing datagrams, and providing a mechanism so that hosts can synchronize their clocks.

ICMP messages are carried in IP packets. The IANA maintains a complete list of ICMP parameters.

3.2.6. IP version 6

The official version of IP that has been in use since the early 1980s is *version 4*. Due to the tremendous growth of the Internet and new emerging applications, it was recognized that a new version of IP was becoming necessary. In late 1995, IP version 6 (IPv6) was entered into the Internet Standards Track. The primary description of IPv6 is contained in RFC 1883 and a number of related specifications, including ICMPv6.

IPv6 is designed as an evolution from IPv4, rather than a radical change. Primary areas of change relate to:

- Increasing the IP address size to 128 bits
- Better support for traffic types with different quality-of-service objectives
- Extensions to support authentication, data integrity, and data confidentiality

For more information about IPv6, check out:

- *IPng: Internet Protocol Next Generation* by Scott Bradner and Allison Mankin (Addison-Wesley, 1996)
- *IPv6: The New Internet Protocol* by Christian Huitema (Prentice-Hall, 1996).
- "IPv6: The Next Generation Internet Protocol" by Gary Kessler.
- *IPng and the TCP/IP Protocols* by Stephen Thomas (John Wiley & Sons, 1996)
- IPng Working Group page (IETF)
- IP Next Generation Web Page (Sun)
- 6bone Web Page (LBL)

3.3. The Transport Layer Protocols

The TCP/IP protocol suite comprises two protocols that correspond roughly to the OSI Transport and Session Layers; these protocols are called the Transmission Control Protocol and the User Datagram Protocol (UDP). One can argue tha it is a misnomer to refer to "TCP/IP applications," as most such applications actually run over TCP or UDP, as shown in Figure 1.

Higher-layer applications are referred to by a port identifier in TCP/UDP messages. The port identifier and IP address together form a *socket*, and the end-to-end communication between two hosts is uniquely identified on the Internet by th four-tuple (source port, source address, destination port, destination address). *Well-known port numbers* denote the server side of a connection and include:

Port #	Protocol	Application
20	TCP	FTP data transfer
21	TCP	FTP control
23	TCP	Telnet
25	TCP	SMTP
43	TCP	whois
53	TCP/UDP	DNS
70	TCP	Gopher
79	TCP	finger
80	TCP	HTTP
110	TCP	POPv3
161	UDP	SNMP
162	UDP	SNMP-trap
520	UDP	RIP

A complete list of port numbers that have been assigned can be found in the IANA's list of Port Numbers.

3.3.1. TCP

TCP, described in RFC 793, provides a virtual circuit (connection-oriented) communication service across the network. TCP includes rules for formatting messages, establishing and terminating virtual circuits, sequencing, flow control, and error correction. Most of the applications in the TCP/IP suite operate over the *reliable* transport service provided by TCP.

```
                     1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgement Number                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Offset |(reserved) |   Flags   |          Window               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Checksum               |      Urgent Pointer           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Options....                               (Padding)   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Data...
+-+-+-+-+-+-+-+-+-+-+-+-+-
```

FIGURE 4. TCP segment format.

The TCP data unit is called a *segment*; the name is due to the fact that TCP does not recognize messages, per se, but merely sends a block of bytes from the byte stream between sender and receiver. The fields of the segment (Figure 4) are:

- *Source Port* and *Destination Port:* Identify the source and destination ports to identify the end-to-end connection and higher-layer application.
- *Sequence Number:* Contains the sequence number of this segment's first data byte in the overall connection byte stream; since the sequence number refers to a byte count rather than a segment count, sequence numbers in contiguous TCP segments are not numbered sequentially.
- *Acknowledgment Number:* Used by the sender to acknowledge receipt of data; this field indicates the sequence number of the next byte expected from the receiver.
- *Data Offset:* Points to the first data byte in this segment; this field, then, indicates the segment header length.
- *Control Flags:* A set of flags that control certain aspects of the TCP virtual connection. The flags include:
    - *Urgent Pointer Field Significant (URG):* When set, indicates that the current segment contains urgent (or high-priority) data and that the Urgent Pointer field value is valid.
    - *Acknowledgment Field Significant (ACK):* When set, indicates that the value contained in the Acknowledgment Number field is valid. This bit is usually set, except during the first message during connection establishment.
    - *Push Function (PSH):* Used when the transmitting application wants to force TCP to immediately transmit the data that is currently buffered without waiting for the buffer to fill; useful for transmitting small units of data.
    - *Reset Connection (RST):* When set, immediately terminates the end-to-end TCP connection.
    - *Synchronize Sequence Numbers (SYN):* Set in the initial segments used to establish a connection, indicating that the segments carry the initial sequence number.
    - *Finish (FIN):* Set to request normal termination of the TCP connection in the direction this segment is traveling; completely closing the connection requires one FIN segment in each direction.
- *Window:* Used for flow control, contains the value of the *receive window size* which is the number of transmitted bytes that the sender of this segment is willing to accept from the receiver.
- *Checksum:* Provides rudimentary bit error detection for the segment (including the header and data).
- *Urgent Pointer:* Urgent data is information that has been marked as high-priority by a higher layer application; this data, in turn, usually bypasses normal TCP buffering and is placed in a segment between the header and "norma data. The Urgent Pointer, valid when the URG flag is set, indicates the position of the first octet of nonexpedited data in the segment.
- *Options:* Used at connection establishment to negotiate a variety of options; maximum segment size (MSS) is the most commonly used option and, if absent, defaults to an MSS of 536. The IANA maintains a list of all TCP Option Numbers.

3.3.2. UDP

UDP, described in RFC 768, provides an end-to-end datagram (connectionless) service. Some applications, such as those that involve a simple query and response, are better suited to the datagram service of UDP because there is no time lost to virtual circuit establishment and termination. UDP's primary function is to add a port number to the IP address to provide a socket for the application.

```
                     1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length             |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Data...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

FIGURE 5. UDP datagram format.

The fields of a UDP datagram (Figure 5) are:

- *Source Port:* Identifies the UDP port at the source side of the connection; use of this field is optional in UDP and may be set to 0.
- *Destination Port:* Identifies the destination port of the end-to-end connection.
- *Length:* Indicates the total length of the UDP datagram.
- *Checksum:* Provides rudimentary bit error detection for the datagram (including the header and data).

3.4. Applications

The TCP/IP Application Layer protocols support the applications and utilities that are the Internet. Commonly used protocols include:

- *Telnet:* Short for *Telecommunication Network*, a virtual terminal protocol allowing a user logged on to one TCP/IP host to access other hosts on the network (RFC 854).

- *FTP:* The File Transfer Protocol allows a user to transfer files between local and remote host computers (RFC 959).
- *Archie:* A utility that allows a user to search all registered anonymous FTP sites for files on a specified topic.
- *Gopher:* A tool that allows users to search through data repositories using a menu-driven, hierarchical interface, with links to other sites (RFC 1436).
- *SMTP:* The Simple Mail Transfer Protocol is the standard protocol for the exchange of electronic mail over the Internet (RFC 821). SMTP is used between e-mail servers on the Internet or to allow an e-mail client to send mail to a server. RFC 822 specifically describes the mail message body format, and RFCs 1521 and 1522 describe MIME (Multipurpose Internet Mail Extensions). Reference books on electronic mail systems include *!%@:: Addressing and Networks* by D. Frey and R. Adams (O'Reilly & Associates, 1993) and *THE INTERNET MESSAGE: Closing the Book With Electronic Mail* by M. Rose (PTR Prentice Hall, 1993).
- *HTTP:* The Hypertext Transfer Protocol is the basis for exchange of information over the World Wide Web (WWW). Various versions of HTTP are in use over the Internet, with HTTP version 1.0 (RFC 1945) being the mos current. WWW pages are written in the Hypertext Markup Language (HTML), an ASCII-based, platform-independent formatting language (RFC 1866).
- *Finger:* Used to determine the status of other hosts and/or users (RFC 1288).
- *POP:* The Post Office Protocol defines a simple interface between a user's mail client software and an e-mail server; POP is used to download mail from the server to the client and allows the user to manage their mailboxes The current version is POP3 (RFC 1460).
- *DNS:* The Domain Name System (described in slightly more detail in Section 3.2.2 above) defines the structure of Internet names and their association with IP addresses, as well as the association of mail and name servers with domains.
- *SNMP:* The Simple Network Management Protocol defines procedures and management information databases for managing TCP/IP-based network devices. SNMP (RFC 1157) is widely deployed in local and wide area network. SNMP Version 2 (SNMPv2, RFC 1441) adds security mechanisms that are missing in SNMP, but is also very complex; widespread use of SNMPv2 has yet to be seen. Additional information on SNMP and TCP/IP-based network management can be found in *SNMP* by S. Feit (McGraw-Hill, 1994) and *THE SIMPLE BOOK: An Introduction to Internet Management*, 2/e, by M. Rose (PTR Prentice Hall, 1994).
- *Ping:* The Packet Internet Groper, a utility that allows a user at one system to determine the status of other hosts and the latency in getting a message to that host. Uses ICMP Echo messages.
- *Whois/NICNAME:* Utilities that search databases for information about Internet domains and domain contact information (RFC 954).
- *Traceroute:* A tool that displays the route that packets will take when traveling to a remote host.

A guide to using most of these applications can be found in "A Primer on Internet and TCP/IP Tools and Utilities" (FYI 30/RFC 2151) by Gary Kessler & Steve Shepard (also available in HTML, Postscript, and Word).

3.5. Summary

As this discussion has shown, *TCP/IP* is not merely a pair of communication protocols but is a suite of protocols, applications, and utilities. Increasingly, these protocols are referred to as the *Internet Protocol Suite*, but the older name will not disappear anytime soon.

```
 ---------------                                      ---------------
| Application   |<------ end-to-end connection ------>| Application   |
|---------------|                                     |---------------|
|     TCP       |<--------- virtual circuit --------->|     TCP       |
|---------------|         -----------------           |---------------|
|     IP        |<-- DG -->|      IP       |<-- DG -->|      IP       |
|---------------|          |-------+-------|          |---------------|
| Subnetwork 1  |<-------->|Subnet1|Subnet2|<-------->| Subnetwork 2  |
 ---------------           --------+--------           ---------------
      HOST                      GATEWAY                      HOST
```

FIGURE 6. TCP/IP protocol suite architecture.

Figure 6 shows the relationship between the various protocol layers of TCP/IP. Applications and utilities reside in host, o end-communicating, systems. TCP provides a reliable, virtual circuit connection between the two hosts. (UDP, not shown, provides an end-to-end datagram connection at this layer.) IP provides a datagram (DG) transport service over any intervening subnetworks, including local and wide area networks. The underlying subnetwork may employ nearly any common local or wide area network technology.

Note that the term *gateway* is used for the device interconnecting the two subnets, a device usually called a *router* in LAN environments or *intermediate system* in OSI environments. In OSI terminology, a *gateway* is used to provide protocol conversion between two networks and/or applications.

4. Other Information Sources

This memo has only provided background information about the TCP/IP protocols and the Internet. There is a wide rang of additional information that the reader can access to further use and understand the tools and scope of the Internet. The real fun begins now!

Internet specifications, standards, reports, humor, and tutorials are distributed as Request for Comments (RFC) documents. RFCs are all freely available on-line, and most are available in ASCII text format.

Internet standards are documented in a subset of the RFCs, identified with an "STD" designation. RFC 2026 describes the Internet standards process and STD 1 always contains the official list of Internet standards.

For Your Information (FYI) documents are another RFC subset, specifically providing background information for the Internet community. The FYI notes are described in RFC 1150.

Frequently Asked Question (FAQ) lists may be found for a number of topics, ranging from ISDN and cryptography to the Internet and Gopher. Two such FAQs are of particular interest to Internet users: "FYI on Questions and Answers - Answers to Commonly asked 'New Internet User' Questions" (RFC 1594) and "FYI on Questions and Answers: Answers to Commonly Asked 'Experienced Internet User' Questions" (RFC 1207). All three of these documents point to even mor information sources.

5. Acronyms and Abbreviations

ARP	Address Resolution Protocol
ARPANET	Advanced Research Projects Agency Network
ASCII	American Standard Code for Information Interchange
ATM	Asynchronous Transfer Mode
BGP	Border Gateway Protocol
BSD	Berkeley Software Development
CCITT	International Telegraph and Telephone Consultative Committee
CIX	Commercial Internet Exchange
DARPA	Defense Advanced Research Projects Agency
DNS	Domain Name System
DoD	U.S. Department of Defense
FAQ	Frequently Asked Questions lists
FDDI	Fiber Distributed Data Interface
FTP	File Transfer Protocol
FYI	For Your Information series of RFCs
GOSIP	U.S. Government Open Systems Interconnection Profile
HTML	Hypertext Markup Language
HTTP	Hypetext Transfer Protocol
IAB	Internet Activities Board
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IESG	Internet Engineering Steering Group
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISO	International Organization for Standardization
ISOC	Internet Society
ITU-T	International Telecommunication Union Telecommunication Standardization Sector
MAC	Medium (or media) access control
Mbps	Megabits (millions of bits) per second
NICNAME	Network Information Center name service
NSF	National Science Foundation
NSFNET	National Science Foundation Network

OSI	Open Systems Interconnection
OSPF	Open Shortest Path First
PPP	Point-to-Point Protocol
RARP	Reverse Address Resolution Protocol
RIP	Routing Information Protocol
RFC	Request For Comments
SLIP	Serial Line IP
SMDS	Switched Multimegabit Data Service
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
STD	Internet Standards series of RFCs
TCP	Transmission Control Protocol
TLD	Top-level domain
UDP	User Datagram Protocol

6. Author's Address

Gary C. Kessler
Hill Associates
106 Highpoint Center
Colchester, VT 05446

+1 802-879-3375 (home office)
+1 630-604-5529 (fax)

E-mail: kumquat@hill.com or kumquat@sover.net
http://www.hill.com or http://www.sover.net/~kessfam

Copyright © 1995, 1996, 1997, 1998, 1999, 2000, 2001 Hill Associates, Inc. All Rights Reserved.

The Common Object Request Broker: Architecture and Specification

Revision 2.6
December 2001

Copyright 1998, 1999, Alcatel
Copyright 1997, 1998, 1999 BEA Systems, Inc.
Copyright 1995, 1996 BNR Europe Ltd.
Copyright 1998, Borland International
Copyright 1998, Cooperative Research Centre for Distributed Systems Technology (DSTC Pty Ltd)
Copyright 2001, Concept Five Technologies
Copyright 1991, 1992, 1995, 1996, Digital Equipment Corporation
Copyright 2001, Eternal Systems, Inc.
Copyright 1995, 1996, 1998, Expersoft Corporation
Copyright 1996, 1997 FUJITSU LIMITED
Copyright 1996, Genesis Development Corporation
Copyright 1989- 2001, Hewlett-Packard Company
Copyright 2001, HighComm
Copyright 1998, 1999, Highlander Communications, L.C.
Copyright 1991, 1992, 1995, 1996 HyperDesk Corporation
Copyright 1998, 1999, Inprise Corporation
Copyright 1996 - 2001, International Business Machines Corporation
Copyright 1995, 1996 ICL, plc
Copyright 1998 - 2001, Inprise Corporation
Copyright 1998, International Computers, Ltd.
Copyright 1995 - 2001, IONA Technologies, Ltd.
Copyright 1998 - 2001, Lockheed Martin Federal Systems, Inc.
Copyright 1998, 1999, 2001, Lucent Technologies, Inc.
Copyright 1996, 1997 Micro Focus Limited
Copyright 1991, 1992, 1995, 1996 NCR Corporation
Copyright 1998, NEC Corporation
Copyright 1998, Netscape Communications Corporation
Copyright 1998, 1999, Nortel Networks
Copyright 1998, 1999, Northern Telecom Corporation
Copyright 1995, 1996, 1998, Novell USG
Copyright 1991, 1992, 1995, 1996 by Object Design, Inc.
Copyright 1991- 2001 Object Management Group, Inc.
Copyright 1998, 1999, 2001, Objective Interface Systems, Inc.
Copyright 1998, 1999, Object-Oriented Concepts, Inc.
Copyright 1998, 2001, Oracle Corporation
Copyright 1998, PeerLogic, Inc.
Copyright 1996, Siemens Nixdorf Informationssysteme AG
Copyright 1991 - 2001, Sun Microsystems, Inc.
Copyright 1995, 1996, SunSoft, Inc.
Copyright 1996, Sybase, Inc.
Copyright 1998, Telefónica Investigación y Desarrollo S.A. Unipersonal
Copyright 1998, TIBCO, Inc.
Copyright 1998, 1999, Tri-Pacific Software, Inc.
Copyright 1996, Visual Edge Software, Ltd.

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013 OMG® and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBAfacilities, CORBAservices, COSS, and IIOP are trademarks of the Object Management Group, Inc. X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents & Specifications, Report a Bug/Issue.

Contents

Preface .... **xxxvii**

1. **The Object Model** **1-1**
    - 1.1 Overview 1-1
    - 1.2 Object Semantics 1-2
        - **1.2.1 Objects** **1-2**
        - **1.2.2 Requests** **1-3**
        - **1.2.3 Object Creation and Destruction** **1-4**
        - **1.2.4 Types** **1-4**
            - 1.2.4.1 Basic types 1-4
            - 1.2.4.2 Constructed types 1-5
        - **1.2.5 Interfaces** **1-6**
        - **1.2.6 Value Types** **1-6**
        - **1.2.7 Abstract Interfaces** **1-7**
        - **1.2.8 Operations** **1-7**
            - 1.2.8.1 Parameters 1-8
            - 1.2.8.2 Return Result 1-8
            - 1.2.8.3 Exceptions 1-8
            - 1.2.8.4 Contexts 1-8
            - 1.2.8.5 Execution Semantics 1-8
        - **1.2.9 Attributes** **1-9**
    - 1.3 Object Implementation 1-9
        - **1.3.1 The Execution Model: Performing Services** **1-9**
        - **1.3.2 The Construction Model** **1-10**
2. **CORBA Overview** **2-1**
    - 2.1 Structure of an Object Request Broker 2-1
        - **2.1.1 Object Request Broker** **2-6**
        - **2.1.2 Clients** **2-7**
        - **2.1.3 Object Implementations** **2-7**
        - **2.1.4 Object References** **2-8**

2.1.5 **OMG Interface Definition Language** **2-8**
2.1.6 **Mapping of OMG IDL to Programming Languages** **2-8**
2.1.7 **Client Stubs** **2-9**
2.1.8 **Dynamic Invocation Interface** **2-9**
2.1.9 **Implementation Skeleton** **2-9**
2.1.10 **Dynamic Skeleton Interface** **2-10**
2.1.11 **Object Adapters** **2-10**
2.1.12 **ORB Interface** **2-10**
2.1.13 **Interface Repository** **2-11**
2.1.14 **Implementation Repository** **2-11**

2.2 Example ORBs 2-11
2.2.1 **Client- and Implementation-resident ORB** **2-11**
2.2.2 **Server-based ORB** **2-12**
2.2.3 **System-based ORB** **2-12**
2.2.4 **Library-based ORB** **2-12**

2.3 Structure of a Client 2-12
2.4 Structure of an Object Implementation 2-13
2.5 Structure of an Object Adapter 2-15
2.6 CORBA Required Object Adapter 2-17
2.6.1 **Portable Object Adapter** **2-17**

2.7 The Integration of Foreign Object Systems 2-17

3. OMG IDL Syntax and Semantics **3-1**

3.1 Overview 3-2
3.2 Lexical Conventions 3-3
3.2.1 **Tokens** **3-5**
3.2.2 **Comments** **3-6**
3.2.3 **Identifiers** **3-6**
3.2.3.1 Escaped Identifiers 3-6
3.2.4 **Keywords** **3-7**
3.2.5 **Literals** **3-8**
3.2.5.1 Integer Literals 3-8
3.2.5.2 Character Literals 3-9
3.2.5.3 Floating-point Literals 3-10
3.2.5.4 String Literals 3-10
3.2.5.5 Fixed-Point Literals 3-11

3.3 Preprocessing 3-11
3.4 OMG IDL Grammar 3-12
3.5 OMG IDL Specification 3-16
3.6 Module Declaration 3-17
3.7 Interface Declaration 3-17
3.7.1 **Interface Header** **3-17**
3.7.2 **Interface Inheritance Specification** **3-18**
3.7.3 **Interface Body** **3-18**

3.7.4 Forward Declaration **3-19**
3.7.5 Interface Inheritance **3-19**

3.8 Value Declaration 3-24
3.8.1 Regular Value Type **3-24**
3.8.1.1 Value Header 3-24
3.8.1.2 Value Element 3-25
3.8.1.3 Value Inheritance Specification 3-25
3.8.1.4 State Members 3-25
3.8.1.5 Initializers 3-26
3.8.1.6 Value Type Example 3-26
3.8.2 Boxed Value Type **3-26**
3.8.3 Abstract Value Type **3-27**
3.8.4 Value Forward Declaration **3-28**
3.8.5 Valuetype Inheritance **3-28**

3.9 Constant Declaration 3-29
3.9.1 Syntax **3-29**
3.9.2 Semantics **3-30**

3.10 Type Declaration 3-33
3.10.1 Basic Types **3-34**
3.10.1.1 Integer Types 3-35
3.10.1.2 Floating-Point Types 3-36
3.10.1.3 Char Type 3-36
3.10.1.4 Wide Char Type 3-36
3.10.1.5 Boolean Type 3-36
3.10.1.6 Octet Type 3-36
3.10.1.7 Any Type 3-37
3.10.2 Constructed Types **3-37**
3.10.2.1 Structures 3-37
3.10.2.2 Discriminated Unions 3-37
3.10.2.3 Constructed Recursive Types and lForward Declarations 3-39
3.10.2.4 Enumerations 3-41
3.10.3 Template Types **3-41**
3.10.3.1 Sequences 3-41
3.10.3.2 Strings 3-42
3.10.3.3 Wstrings 3-42
3.10.3.4 Fixed Type 3-43
3.10.4 Complex Declarator **3-43**
3.10.4.1 Arrays 3-43
3.10.5 Native Types **3-43**

3.11 Exception Declaration 3-47

3.12 Operation Declaration 3-47
3.12.1 Operation Attribute **3-48**
3.12.2 Parameter Declarations **3-48**
3.12.3 Raises Expressions **3-49**
3.12.4 Context Expressions **3-49**

3.13 Attribute Declaration 3-50

3.14 CORBA Module 3-51

3.15 Names and Scoping 3-52
  3.15.1 Qualified Names 3-52
  3.15.2 Scoping Rules and Name Resolution 3-54
  3.15.3 Special Scoping Rules for Type Names 3-57

4. ORB Interface 4-1

4.1 Overview 4-1
4.2 The ORB Operations 4-2
  4.2.1 ORB Identity 4-7
    4.2.1.1 id 4-7
  4.2.2 Converting Object References to Strings 4-8
    4.2.2.1 object_to_string 4-8
    4.2.2.2 string_to_object 4-8
  4.2.3 Getting Service Information 4-8
    4.2.3.1 get_service_information 4-8
  4.2.4 Thread-Related Operations 4-9
    4.2.4.1 work_pending 4-9
    4.2.4.2 perform_work 4-9
    4.2.4.3 run 4-10
    4.2.4.4 shutdown 4-10
    4.2.4.5 destroy 4-11
4.3 Object Reference Operations 4-12
  4.3.1 Determining the Object Interface 4-13
    4.3.1.1 get_interface 4-13
  4.3.2 Duplicating and Releasing Copies of Object References 4-14
    4.3.2.1 duplicate 4-14
    4.3.2.2 release 4-14
  4.3.3 Nil Object References 4-14
    4.3.3.1 is_nil 4-14
  4.3.4 Equivalence Checking Operation 4-15
    4.3.4.1 is_a 4-15
  4.3.5 Probing for Object Non-Existence 4-15
    4.3.5.1 non_existent 4-15
  4.3.6 Object Reference Identity 4-16
    4.3.6.1 Hashing Object Identifiers 4-16
    4.3.6.2 Equivalence Testing 4-16
  4.3.7 Type Coercion Considerations 4-17
  4.3.8 Getting Policy Associated with the Object 4-17
    4.3.8.1 get_policy 4-17
    4.3.8.2 get_client_policy 4-18
    4.3.8.3 get_policy_overrides 4-19
  4.3.9 Overriding Associated Policies on an Object Reference 4-19
    4.3.9.1 set_policy_overrides 4-19
  4.3.10 Validating Connection 4-20
    4.3.10.1 validate_connection 4-20
  4.3.11 Getting the Domain Managers Associated with the Object 4-20
    4.3.11.1 get_domain_managers 4-20
4.4 ValueBase Operations 4-21

4.5 ORB and OA Initialization and Initial References 4-21

4.5.1 ORB Initialization 4-22

4.5.2 Obtaining Initial Object References. 4-23

4.5.3 Configuring Initial Service References. 4-26

4.5.3.1 ORB-specific Configuration 4-26
4.5.3.2 ORBInitRef 4-26
4.5.3.3 ORBDefaultInitRef 4-27
4.5.3.4 Configuration Effect on resolve_initial_references 4-27
4.5.3.5 Configuration Effect on list_initial_services 4-28

4.6 Context Object 4-28

4.6.1 Introduction 4-28

4.6.2 Context Object Operations 4-29

4.6.2.1 get_default_context 4-30
4.6.2.2 set_one_value 4-30
4.6.2.3 set_values 4-30
4.6.2.4 get_values 4-31
4.6.2.5 delete_values 4-31
4.6.2.6 create_child 4-32
4.6.2.7 delete 4-32

4.7 Current Object 4-32

4.8 Policy Object 4-33

4.8.1 Definition of Policy Object 4-33

4.8.1.1 Copy 4-34
4.8.1.2 Destroy 4-34
4.8.1.3 Policy_type 4-34

4.8.2 Creation of Policy Objects. 4-34

4.8.2.1 PolicyErrorCode 4-35
4.8.2.2 PolicyError 4-35
4.8.2.3 Create_policy 4-35

4.8.3 Usages of Policy Objects 4-36

4.8.4 Policy Associated with the Execution Environment 4-37

4.8.5 Specification of New Policy Objects 4-37

4.8.6 Standard Policies 4-39

4.9 Management of Policies. 4-43

4.9.1 Client Side Policy Management 4-43

4.9.2 Server Side Policy Management. 4-43

4.9.3 Policy Management Interfaces 4-44

4.9.3.1 interface PolicyManager 4-44
4.9.3.2 interface PolicyCurrent 4-46

4.10 Management of Policy Domains 4-46

4.10.1 Basic Concepts 4-46

4.10.1.1 Policy Domain 4-46
4.10.1.2 Policy Domain Manager 4-47
4.10.1.3 Policy Objects 4-47
4.10.1.4 Object Membership of Policy Domains 4-47
4.10.1.5 Domains Association at Object Reference Creation 4-48
4.10.1.6 Implementor's View of Object Creation 4-48

4.10.2 Domain Management Operations. 4-49

4.10.2.7 Domain Manager 4-50
4.10.2.8 Construction Policy 4-51

4.11 TypeCodes 4-51

4.11.1 The TypeCode Interface 4-52
4.11.2 TypeCode Constants 4-56
4.11.3 Creating TypeCodes 4-57

4.12 Exceptions 4-61

4.12.1 Definition of Terms 4-61
4.12.2 System Exceptions 4-62
4.12.3 Standard System Exception Definitions 4-63

4.12.3.1 UNKNOWN 4-65
4.12.3.2 BAD_PARAM 4-65
4.12.3.3 NO_MEMORY 4-65
4.12.3.4 IMP_LIMIT 4-66
4.12.3.5 COMM_FAILURE 4-66
4.12.3.6 INV_OBJREF 4-66
4.12.3.7 NO_PERMISSION 4-66
4.12.3.8 INTERNAL 4-66
4.12.3.9 MARSHAL 4-66
4.12.3.10 INITIALIZE 4-67
4.12.3.11 NO_IMPLEMENT 4-67
4.12.3.12 BAD_TYPECODE 4-67
4.12.3.13 BAD_OPERATION 4-67
4.12.3.14 NO_RESOURCES 4-67
4.12.3.15 NO_RESPONSE 4-67
4.12.3.16 PERSIST_STORE 4-67
4.12.3.17 BAD_INV_ORDER............... 4-67
4.12.3.18 TRANSIENT 4-68
4.12.3.19 FREE_MEM 4-68
4.12.3.20 INV_IDENT 4-68
4.12.3.21 INV_FLAG 4-68
4.12.3.22 INTF_REPOS 4-68
4.12.3.23 BAD_CONTEXT 4-68
4.12.3.24 OBJ_ADAPTER 4-68
4.12.3.25 DATA_CONVERSION 4-68
4.12.3.26 OBJECT_NOT_EXIST 4-69
4.12.3.27 TRANSACTION_REQUIRED 4-69
4.12.3.28 TRANSACTION_ROLLEDBACK .. 4-69
4.12.3.29 INVALID_TRANSACTION 4-69
4.12.3.30 INV_POLICY 4-69
4.12.3.31 CODESET_INCOMPATIBLE 4-69
4.12.3.32 REBIND 4-69
4.12.3.33 TIMEOUT 4-70
4.12.3.34 TRANSACTION_UNAVAILABLE . 4-70
4.12.3.35 TRANSACTION_MODE 4-70
4.12.3.36 BAD_QOS 4-70

4.12.4 Standard Minor Exception Codes 4-70

5. Value Type Semantics.................................. 5-1

5.1 Overview 5-1
5.2 Architecture 5-2

5.2.1 Abstract Values 5-3

- **5.2.2 Operations 5-3**
- **5.2.3 Value Type vs. Interfaces.................... 5-4**
- **5.2.4 Parameter Passing......................... 5-4**
  - 5.2.4.1 Value vs. Reference Semantics 5-4
  - 5.2.4.2 Sharing Semantics 5-4
  - 5.2.4.3 Identity Semantics 5-4
  - 5.2.4.4 Any parameter type 5-5
- **5.2.5 Substitutability Issues 5-5**
  - 5.2.5.1 Value instance -> Interface type 5-5
  - 5.2.5.2 Value Instance -> Abstract interface type 5-5
  - 5.2.5.3 Value instance -> Value type 5-5
- **5.2.6 Widening/Narrowing....................... 5-6**
- **5.2.7 Value Base Type.......................... 5-6**
- **5.2.8 Life Cycle issues.......................... 5-7**
  - 5.2.8.1 Creation and Factories 5-7
- **5.2.9 Security Considerations 5-7**
  - 5.2.9.1 Value as Value 5-8
  - 5.2.9.2 Value as Object Reference 5-8

5.3 Standard Value Box Definitions 5-9

5.4 Language Mappings 5-9

- **5.4.1 General Requirements...................... 5-9**
- **5.4.2 Language Specific Marshaling 5-9**
- **5.4.3 Language Specific Value Factory Requirements. 5-9**
- **5.4.4 Value Method Implementation............... 5-10**

5.5 Custom Marshaling 5-10

- **5.5.1 Implementation of Custom Marshaling........ 5-11**
- **5.5.2 Marshaling Streams....................... 5-11**

5.6 Access to the Sending Context Run Time 5-18

6. Abstract Interface Semantics............................ 6-1

6.1 Overview 6-1

6.2 Semantics of Abstract Interfaces 6-1

6.3 Usage Guidelines 6-3

6.4 Example 6-3

6.5 Security Considerations 6-4

- **6.5.1 Passing Values to Trusted Domains 6-4**

7. Dynamic Invocation Interface 7-1

7.1 Overview 7-1

- **7.1.1 Common Data Structures 7-2**
- **7.1.2 Memory Usage 7-4**
- **7.1.3 Return Status and Exceptions................ 7-4**

7.2 Request Operations 7-4

- **7.2.1 create_request 7-5**
- **7.2.2 add_arg................................ 7-7**
- **7.2.3 invoke 7-8**

7.2.4 **delete** 7-8
7.2.5 **send** 7-8
7.2.6 **poll_response** 7-9
7.2.7 **get_response** 7-9
7.2.8 **sendp** 7-10
7.2.9 **prepare** 7-10
7.2.10 **sendc** 7-10
7.3 ORB Operations 7-11
7.3.1 **send_multiple_requests** 7-11
7.3.2 **get_next_response and poll_next_response** 7-11
7.4 Polling 7-12
7.4.1 Abstract Valuetype **Pollable** 7-14
7.4.1.1 is_ready 7-14
7.4.1.2 create_pollable_set 7-14
7.4.2 Abstract Valuetype DIIPollable 7-14
7.4.3 interface **PollableSet** 7-14
7.4.3.1 create_dii_pollable 7-15
7.4.3.2 add_pollable 7-15
7.4.3.3 get_ready_pollable 7-15
7.4.3.4 remove 7-16
7.4.3.5 number_left 7-16
7.5 List Operations 7-16
7.5.1 **create_list** 7-17
7.5.2 **add_item** 7-17
7.5.3 **free** 7-17
7.5.4 **free_memory** 7-18
7.5.5 **get_count** 7-18
7.5.6 **create_operation_list** 7-18
8. Dynamic Skeleton Interface **8-1**
8.1 Introduction 8-1
8.2 Overview 8-2
8.3 ServerRequestPseudo-Object 8-3
8.3.1 **ExplicitRequest State: ServerRequestPseudo-Object** 8-3
8.4 DSI: Language Mapping 8-4
8.4.1 **ServerRequest's Handling of Operation Parameters** 8-4
8.4.2 **Registering Dynamic Implementation Routines** . 8-5
9. Dynamic Management of Any Values **9-1**
9.1 Overview 9-1
9.2 DynAny API 9-3
9.2.1 **Locality and Usage Constraints** 9-9
9.2.2 **Creating a DynAny Object** 9-9
9.2.3 **The DynAny Interface** 9-11
9.2.3.1 Obtaining the TypeCode associated

with a DynAny object 9-11
9.2.3.2 Initializing a DynAny object from another DynAny object 9-12
9.2.3.3 Initializing a DynAny object from an any value 9-12
9.2.3.4 Generating an any value from a DynAny object 9-12
9.2.3.5 Comparing DynAny values 9-12
9.2.3.6 Destroying a DynAny object 9-13
9.2.3.7 Creating a copy of a DynAny object 9-13
9.2.3.8 Accessing a value of some basic type in a DynAny object 9-13
9.2.3.9 Iterating through components of a DynAny 9-15
9.2.4 The DynFixed Interface 9-16
9.2.5 The DynEnum Interface 9-16
9.2.6 The DynStruct Interface 9-17
9.2.7 The DynUnion interface 9-19
9.2.8 The DynSequence Interface 9-21
9.2.9 The DynArray Interface 9-22
9.2.10 The DynValueCommon Interface............. 9-23
9.2.11 The DynValue Interface 9-24
9.2.12 The DynValueBox Interface 9-24
9.3 Usage in C++ Language.......................... 9-25
9.3.1 Dynamic creation of CORBA::Any values...... 9-25
9.3.1.1 Creating an any that contains a struct ... 9-25
9.3.2 Dynamic interpretation of CORBA::Any values. 9-26
9.3.2.1 Filtering of events 9-26
10. The Interface Repository.................................. 10-1
10.1 Overview 10-1
10.2 Scope of an Interface Repository 10-2
10.3 Implementation Dependencies..................... 10-4
10.3.1 Managing Interface Repositories 10-4
10.4 Basics.. 10-5
10.4.1 Names and Identifiers 10-6
10.4.2 Types and TypeCodes 10-6
10.4.3 Interface Repository Objects 10-6
10.4.4 Structure and Navigation of the Interface Repository 10-7
10.5 Interface Repository Interfaces..................... 10-9
10.5.1 Supporting Type Definitions................. 10-10
10.5.2 IRObject.................................. 10-11
10.5.2.1 Read Interface 10-11
10.5.2.2 Write Interface 10-11
10.5.3 Contained................................. 10-11
10.5.3.1 Read Interface 10-12
10.5.3.2 Write Interface 10-13
10.5.4 Container 10-14
10.5.4.1 Read Interface 10-17

10.5.4.2 Write Interface 10-18

10.5.5 IDLType 10-19

10.5.6 Repository 10-20

10.5.6.1 Read Interface 10-21
10.5.6.2 Write Interface 10-21

10.5.7 ModuleDef 10-22

10.5.8 ConstantDef 10-22

10.5.8.1 Read Interface 10-22
10.5.8.2 Write Interface 10-23

10.5.9 TypedefDef 10-23

10.5.10 StructDef 10-23

10.5.10.1 Read Interface 10-24
10.5.10.2 Write Interface 10-24

10.5.11 UnionDef 10-24

10.5.11.1 Read Interface 10-24
10.5.11.2 Write Interface 10-25

10.5.12 EnumDef 10-25

10.5.12.1 Read Interface 10-25
10.5.12.2 Write Interface 10-25

10.5.13 AliasDef 10-25

10.5.13.1 Read Interface 10-26
10.5.13.2 Write Interface 10-26

10.5.14 PrimitiveDef 10-26

10.5.15 StringDef 10-26

10.5.16 WstringDef 10-27

10.5.17 FixedDef 10-27

10.5.18 SequenceDef 10-27

10.5.18.1 Read Interface 10-28
10.5.18.2 Write Interface 10-28

10.5.19 ArrayDef 10-28

10.5.19.1 Read Interface 10-28
10.5.19.2 Write Interface 10-28

10.5.20 ExceptionDef 10-29

10.5.20.1 Read Interface 10-29
10.5.20.2 Write Interface 10-29

10.5.21 AttributeDef 10-29

10.5.21.1 Read Interface 10-30
10.5.21.2 Write Interface 10-30

10.5.22 OperationDef 10-30

10.5.22.1 Read Interface 10-31
10.5.22.2 Write Interface 10-32

10.5.23 InterfaceDef 10-32

10.5.23.1 Read Interface 10-33
10.5.23.2 Write Interface 10-34

10.5.24 AbstractInterfaceDef 10-34

10.5.24.1 Read Interface 10-34
10.5.24.2 Write Interface 10-35

10.5.25 LocalInterfaceDef 10-35

10.5.25.1 Read Interface 10-36
10.5.25.2 Write Interface 10-36

10.5.26 ValueMemberDef 10-37

10.5.26.1 Read Interface 10-37
10.5.26.2 Write Interface 10-38

10.5.27 ValueDef **10-38**
    10.5.27.1 Read Interface 10-40
    10.5.27.2 Write Interface 10-40
10.5.28 ValueBoxDef **10-41**
    10.5.28.1 Read Interface 10-41
    10.5.28.2 Write Interface 10-41
10.5.29 NativeDef **10-41**

10.6 RepositoryIds 10-42
10.6.1 OMG IDL Format **10-42**
10.6.2 RMI Hashed Format **10-43**
10.6.3 DCE UUID Format **10-44**
10.6.4 LOCAL Format **10-45**
10.6.5 Pragma Directives for RepositoryId **10-45**
    10.6.5.1 The ID Pragma 10-45
    10.6.5.2 The Prefix Pragma 10-45
    10.6.5.3 The Version Pragma 10-48
    10.6.5.4 Generation of OMG IDL - Format IDs . 10-49
10.6.6 For More Information **10-50**
10.6.7 RepositoryIDs for OMG-Specified Types **10-50**

10.7 OMG IDL for Interface Repository 10-51

11. The Portable Object Adapter **11-1**

11.1 Overview 11-1

11.2 Abstract Model Description 11-2
11.2.1 Model Components **11-2**
11.2.2 Model Architecture **11-4**
11.2.3 POA Creation **11-6**
11.2.4 Reference Creation **11-7**
11.2.5 Object Activation States **11-8**
11.2.6 Request Processing **11-9**
11.2.7 Implicit Activation **11-10**
11.2.8 Multi-threading **11-11**
    11.2.8.1 POA Threading Models 11-11
    11.2.8.2 Using the Single Thread Model 11-11
    11.2.8.3 Using the ORB Controlled Model 11-12
    11.2.8.4 Using the Main Thread Model 11-12
    11.2.8.5 Limitations When Using Multiple Threads 11-12
11.2.9 Dynamic Skeleton Interface **11-12**
11.2.10 Location Transparency **11-14**

11.3 Interfaces 11-14
11.3.1 The Servant IDL Type **11-15**
11.3.2 POAManager Interface **11-15**
    11.3.2.1 Processing States 11-16
    11.3.2.2 activate 11-18
    11.3.2.3 hold_requests 11-18
    11.3.2.4 discard_requests 11-19
    11.3.2.5 deactivate 11-19
    11.3.2.6 get_state 11-20

11.3.3 AdapterActivator Interface 11-20
11.3.3.1 unknown_adapter 11-20
11.3.4 ServantManager Interface 11-22
11.3.4.1 Common Information for Servant Manager Types 11-22
11.3.5 ServantActivator Interface 11-23
11.3.5.1 incarnate 11-23
11.3.5.2 etherealize 11-24
11.3.6 ServantLocator Interface 11-25
11.3.6.1 preinvoke 11-26
11.3.6.2 postinvoke 11-27
11.3.6.3 ServantLocator and Location Determination 11-27
11.3.7 POA Policy Objects 11-28
11.3.7.1 Thread Policy 11-28
11.3.7.2 Lifespan Policy 11-29
11.3.7.3 Object Id Uniqueness Policy 11-29
11.3.7.4 Id Assignment Policy 11-30
11.3.7.5 Servant Retention Policy 11-30
11.3.7.6 Request Processing Policy 11-31
11.3.7.7 Implicit Activation Policy 11-32
11.3.8 POA Interface 11-33
11.3.8.1 create_POA 11-33
11.3.8.2 find_POA 11-34
11.3.8.3 destroy 11-34
11.3.8.4 Policy Creation Operations 11-35
11.3.8.5 the_name 11-36
11.3.8.6 the_parent 11-36
11.3.8.7 the_children 11-36
11.3.8.8 the_POAManager 11-36
11.3.8.9 the_activator 11-36
11.3.8.10 get_servant_manager 11-37
11.3.8.11 set_servant_manager 11-37
11.3.8.12 get_servant 11-37
11.3.8.13 set_servant 11-37
11.3.8.14 activate_object 11-38
11.3.8.15 activate_object_with_id 11-38
11.3.8.16 deactivate_object 11-38
11.3.8.17 create_reference 11-39
11.3.8.18 create_reference_with_id 11-39
11.3.8.19 servant_to_id 11-40
11.3.8.20 servant_to_reference 11-41
11.3.8.21 reference_to_servant 11-41
11.3.8.22 reference_to_id 11-42
11.3.8.23 id_to_servant 11-42
11.3.8.24 id_to_reference 11-42
11.3.8.25 id 11-42
11.3.9 Current Operations 11-43
11.3.9.1 get_POA 11-43
11.3.9.2 get_object_id 11-43
11.3.9.3 get_reference 11-43
11.3.9.4 get_servant 11-44

11.4 IDL for PortableServer Module 11-44

11.5 UML Description of PortableServer. 11-50

11.6 Usage Scenarios 11-52
    11.6.1 Getting the Root POA 11-52
    11.6.2 Creating a POA 11-53
    11.6.3 Explicit Activation with POA-assigned Object Ids 11-53
    11.6.4 Explicit Activation with User-assigned Object Ids 11-54
    11.6.5 Creating References before Activation. 11-55
    11.6.6 Servant Manager Definition and Creation. 11-55
    11.6.7 Object Activation on Demand. 11-57
    11.6.8 Persistent Objects with POA-assigned Ids. 11-59
    11.6.9 Multiple Object Ids Mapping to a Single Servant 11-59
    11.6.10 One Servant for All Objects 11-59
    11.6.11 Single Servant, Many Objects and Types, Using DSI 11-62

12. Interoperability Overview 12-1

12.1 Elements of Interoperability. 12-1
    12.1.1 ORB Interoperability Architecture 12-2
    12.1.2 Inter-ORB Bridge Support 12-2
    12.1.3 General Inter-ORB Protocol (GIOP). 12-3
    12.1.4 Internet Inter-ORB Protocol (IIOP). 12-3
    12.1.5 Environment-Specific Inter-ORB Protocols (ESIOPs). 12-4

12.2 Relationship to Previous Versions of CORBA 12-4

12.3 Examples of Interoperability Solutions 12-5
    12.3.1 Example 1. 12-5
    12.3.2 Example 2. 12-5
    12.3.3 Example 3. 12-5
    12.3.4 Interoperability Compliance. 12-5

12.4 Motivating Factors 12-8
    12.4.1 ORB Implementation Diversity 12-8
    12.4.2 ORB Boundaries 12-8
    12.4.3 ORBs Vary in Scope, Distance, and Lifetime. . . . 12-9

12.5 Interoperability Design Goals. 12-9
    12.5.1 Non-Goals. 12-10

13. ORB Interoperability Architecture 13-1

13.1 Overview 13-1
    13.1.1 Domains 13-2
    13.1.2 Bridging Domains 13-2

13.2 ORBs and ORB Services 13-3
    13.2.1 The Nature of ORB Services. 13-3
    13.2.2 ORB Services and Object Requests 13-3
    13.2.3 Selection of ORB Services. 13-4

13.3 Domains. 13-5
    13.3.1 Definition of a Domain. 13-5

13.3.2 Mapping Between Domains: Bridging 13-6

13.4 Interoperability Between ORBs 13-7

13.4.1 ORB Services and Domains 13-7

13.4.2 ORBs and Domains 13-7

13.4.3 Interoperability Approaches................. 13-8

13.4.3.1 Mediated Bridging 13-8
13.4.3.2 Immediate Bridging 13-9
13.4.3.3 Location of Inter-Domain Functionality 13-9
13.4.3.4 Bridging Level 13-10

13.4.4 Policy-Mediated Bridging 13-10

13.4.5 Configurations of Bridges in Networks 13-11

13.5 Object Addressing 13-11

13.5.1 Domain-relative Object Referencing 13-12

13.5.2 Handling of Referencing Between Domains 13-12

13.6 An Information Model for Object References 13-14

13.6.1 What Information Do Bridges Need?.......... 13-14

13.6.2 Interoperable Object References: IORs........ 13-14

13.6.3 IOR Profiles 13-15

13.6.4 Standard IOR Profiles...................... 13-17

13.6.4.1 The TAG_INTERNET_IOP Profile ... 13-17
13.6.4.2 The TAG_MULTIPLE_COMPONENTS Profile 13-18
13.6.4.3 The TAG_SCCP_IOP Profile 13-18

13.6.5 IOR Components........................... 13-18

13.6.6 Standard IOR Components 13-19

13.6.6.1 TAG_ORB_TYPE Component 13-20
13.6.6.2 TAG_ALTERNATE_IIOP_ADDRESS Component 13-20
13.6.6.3 Other Components 13-20

13.6.7 Profile and Component Composition in IORs... 13-21

13.6.8 IOR Creation and Scope 13-22

13.6.9 Stringified Object References................ 13-22

13.6.10 Object URLs............................. 13-23

13.6.10.1 corbaloc URL 13-24
13.6.10.2 corbaloc:rir URL 13-25
13.6.10.3 corbaloc:iiop URL 13-26
13.6.10.4 corbaloc Server Implementation 13-27
13.6.10.5 corbaname URL 13-27
13.6.10.6 Future corbaloc URL Protocols 13-27
13.6.10.7 Future URL Schemes 13-27

13.7 Service Context 13-28

13.7.1 Standard Service Contexts 13-29

13.7.2 Service Context Processing Rules............. 13-31

13.8 Coder/Decoder Interfaces.......................... 13-31

13.8.1 Codec Interface 13-31

13.8.1.1 Exceptions 13-32
13.8.1.2 Operations 13-32

13.8.2 Codec Factory 13-33

13.8.2.1 Encoding Structure 13-34

13.8.2.2 CodecFactory Interface 13-34

13.9 Feature Support and GIOP Versions. 13-35

13.10 Code Set Conversion 13-36

13.10.1 Character Processing Terminology 13-36

13.10.1.1 Character Set 13-36
13.10.1.2 Coded Character Set, or Code Set 13-36
13.10.1.3 Code Set Classifications 13-37
13.10.1.4 Narrow and Wide Characters 13-37
13.10.1.5 Char Data and Wchar Data 13-38
13.10.1.6 Byte-Oriented Code Set 13-38
13.10.1.7 Multi-Byte Character Strings 13-38
13.10.1.8 Non-Byte-Oriented Code Set 13-38
13.10.1.9 Char and Wchar Transmission Code Set (TCS-C and TCS-W) 13-38
13.10.1.10 Process Code Set and File Code Set . 13-38
13.10.1.11 Native Code Set 13-39
13.10.1.12 Transmission Code Set 13-39
13.10.1.13 Conversion Code Set (CCS) 13-39

13.10.2 Code Set Conversion Framework. 13-39

13.10.2.1 Requirements 13-39
13.10.2.2 Overview of the Conversion Framework 13-40
13.10.2.3 ORB Databases and Code Set Converters 13-41
13.10.2.4 CodeSet Component of IOR Multi-Component Profile 13-42
13.10.2.5 GIOP Code Set Service Context 13-43
13.10.2.6 Code Set Negotiation 13-44

13.10.3 Mapping to Generic Character Environments . . 13-47

13.10.3.1 Describing Generic Interfaces 13-48
13.10.3.2 Interoperation 13-48

13.10.4 Example of Generic Environment Mapping 13-48

13.10.4.1 Generic Mappings 13-49
13.10.4.2 Interoperation and Generic Mappings . 13-49

13.10.5 Relevant OSFM Registry Interfaces. 13-49

13.10.5.1 Character and Code Set Registry 13-49
13.10.5.2 Access Routines 13-50

14. Building Inter-ORB Bridges 14-1

14.1 Introduction 14-1

14.2 In-Line and Request-Level Bridging 14-2

14.2.1 In-line Bridging 14-3

14.2.2 Request-level Bridging 14-3

14.2.3 Collocated ORBs 14-4

14.3 Proxy Creation and Management 14-5

14.4 Interface-specific Bridges and Generic Bridges 14-6

14.5 Building Generic Request-Level Bridges. 14-6

14.6 Bridging Non-Referencing Domains 14-7

14.7 Bootstrapping Bridges 14-7

Contents

15. General Inter-ORB Protocol 15-1
15.1 Goals of the General Inter-ORB Protocol. 15-2
15.2 GIOP Overview 15-2
15.2.1 Common Data Representation (CDR) 15-3
15.2.2 GIOP Message Overview 15-3
15.2.3 GIOP Message Transfer 15-4
15.3 CDR Transfer Syntax 15-4
15.3.1 Primitive Types 15-5
15.3.1.1 Alignment 15-5
15.3.1.2 Integer Data Types 15-6
15.3.1.3 Floating Point Data Types 15-7
15.3.1.4 Octet 15-10
15.3.1.5 Boolean 15-10
15.3.1.6 Character Types 15-10
15.3.2 OMG IDL Constructed Types. 15-11
15.3.2.1 Alignment 15-11
15.3.2.2 Struct 15-12
15.3.2.3 Union 15-12
15.3.2.4 Array 15-12
15.3.2.5 Sequence 15-12
15.3.2.6 Enum 15-12
15.3.2.7 Strings and Wide Strings 15-12
15.3.2.8 Fixed-Point Decimal Type 15-13
15.3.3 Encapsulation. 15-14
15.3.4 Value Types 15-15
15.3.4.1 Partial Type Information and Versioning 15-16
15.3.4.2 Example 15-17
15.3.4.3 Scope of the Indirections 15-19
15.3.4.4 Null Values 15-19
15.3.4.5 Other Encoding Information 15-19
15.3.4.6 Fragmentation 15-19
15.3.4.7 Notation 15-22
15.3.4.8 The Format 15-22
15.3.5 Pseudo-Object Types 15-23
15.3.5.1 TypeCode 15-23
15.3.5.2 Any 15-29
15.3.5.3 Principal 15-29
15.3.5.4 Context 15-29
15.3.5.5 Exception 15-29
15.3.6 Object References 15-30
15.3.7 Abstract Interfaces 15-30
15.4 GIOP Message Formats 15-30
15.4.1 GIOP Message Header 15-31
15.4.2 Request Message 15-33
15.4.2.1 Request Header 15-33
15.4.2.2 Request Body 15-36
15.4.3 Reply Message 15-37
15.4.3.1 Reply Header 15-37
15.4.3.2 Reply Body 15-38
15.4.4 CancelRequest Message 15-40
15.4.4.1 Cancel Request Header 15-40

15.4.5 LocateRequest Message 15-41
    15.4.5.1 LocateRequest Header 15-41
15.4.6 LocateReply Message 15-42
    15.4.6.1 Locate Reply Header 15-42
    15.4.6.2 LocateReply Body 15-43
    15.4.6.3 Handling ForwardRequest Exception from ServantLocator 15-44
15.4.7 CloseConnection Message................... 15-44
15.4.8 MessageError Message 15-44
15.4.9 Fragment Message......................... 15-44

15.5 GIOP Message Transport.......................... 15-46
15.5.1 Connection Management.................... 15-46
    15.5.1.1 Connection Closure 15-47
    15.5.1.2 Multiplexing Connections 15-48
15.5.2 Message Ordering 15-48

15.6 Object Location 15-48

15.7 Internet Inter-ORB Protocol (IIOP) 15-50
15.7.1 TCP/IP Connection Usage................... 15-51
15.7.2 IIOP IOR Profiles 15-51
15.7.3 IIOP IOR Profile Components 15-54

15.8 Bi-Directional GIOP 15-55
15.8.1 Bi-Directional IIOP........................ 15-57
    15.8.1.1 IIOP/SSL considerations 15-58

15.9 Bi-directional GIOP policy.......................... 15-58

15.10 OMG IDL...................................... 15-59
15.10.1 GIOP Module............................ 15-59
15.10.2 IIOP Module 15-63
15.10.3 BiDirPolicy Module....................... 15-64

16. The DCE ESIOP 16-1

16.1 Goals of the DCE Common Inter-ORB Protocol 16-1

16.2 DCE Common Inter-ORB Protocol Overview 16-2
16.2.1 DCE-CIOP RPC 16-2
16.2.2 DCE-CIOP Data Representation 16-3
16.2.3 DCE-CIOP Messages....................... 16-4
16.2.4 Interoperable Object Reference (IOR) 16-5

16.3 DCE-CIOP Message Transport 16-5
16.3.1 Pipe-based Interface 16-6
    16.3.1.1 Invoke 16-8
    16.3.1.2 Locate 16-8
16.3.2 Array-based Interface..................... 16-8
    16.3.2.1 Invoke 16-10
    16.3.2.2 Locate 16-11

16.4 DCE-CIOP Message Formats....................... 16-11
16.4.1 DCE_CIOP Invoke Request Message.......... 16-11
    16.4.1.1 Invoke request header 16-11
    16.4.1.2 Invoke request body 16-12

16.4.2 DCE-CIOP Invoke Response Message 16-12
16.4.2.1 Invoke response header 16-13
16.4.2.2 Invoke Response Body 16-13
16.4.3 DCE-CIOP Locate Request Message 16-14
16.4.3.1 Locate Request Header 16-14
16.4.4 DCE-CIOP Locate Response Message 16-15
16.4.4.1 Locate Response Header 16-15
16.4.4.2 Locate Response Body 16-16
16.5 DCE-CIOP Object References 16-16
16.5.1 DCE-CIOP String Binding Component. 16-17
16.5.2 DCE-CIOP Binding Name Component 16-18
16.5.2.1 BindingNameComponent 16-18
16.5.3 DCE-CIOP No Pipes Component. 16-19
16.5.4 Complete Object Key Component 16-19
16.5.5 Endpoint ID Position Component. 16-20
16.5.6 Location Policy Component 16-20
16.6 DCE-CIOP Object Location. 16-21
16.6.1 Location Mechanism Overview 16-22
16.6.2 Activation. 16-23
16.6.3 Basic Location Algorithm 16-23
16.6.4 Use of the Location Policy and the Endpoint ID . 16-24
16.6.4.1 Current location policy 16-24
16.6.4.2 Original location policy 16-24
16.6.4.3 Original Endpoint ID 16-24
16.7 OMG IDL for the DCE CIOP Module 16-25
16.8 References for this Chapter 16-26
17. Interworking Architecture 17-1
17.1 Purpose of the Interworking Architecture 17-2
17.1.1 Comparing COM Objects to CORBA Objects . . 17-2
17.2 Interworking Object Model 17-3
17.2.1 Relationship to CORBA Object Model 17-3
17.2.2 Relationship to the OLE/COM Model 17-4
17.2.3 Basic Description of the Interworking Model . . . 17-4
17.3 Interworking Mapping Issues. 17-8
17.4 Interface Mapping 17-8
17.4.1 CORBA/COM 17-9
17.4.2 CORBA/Automation 17-9
17.4.3 COM/CORBA 17-10
17.4.4 Automation/CORBA 17-10
17.5 Interface Composition Mappings 17-11
17.5.1 CORBA/COM 17-11
17.5.1.1 COM/CORBA 17-12
17.5.1.2 CORBA/Automation 17-12
17.5.1.3 Automation/CORBA 17-13
17.5.2 Detailed Mapping Rules 17-13
17.5.2.1 Ordering Rules for the CORBA->MIDL

Transformation 17-13
17.5.2.2 Ordering Rules for the CORBA->Automation Transformation . 17-13

17.5.3 Example of Applying Ordering Rules 17-14

17.5.4 Mapping Interface Identity.................. 17-16

17.5.4.1 Mapping Interface Repository IDs to COM IIDs 17-17
17.5.4.2 Mapping COM IIDs to CORBA Interface IDs 17-18

17.6 Object Identity, Binding, and Life Cycle 17-18

17.6.1 Object Identity Issues 17-19

17.6.1.1 CORBA Object Identity and Reference Properties 17-19
17.6.1.2 COM Object Identity and Reference Properties 17-19

17.6.2 Binding and Life Cycle 17-20

17.6.2.1 Lifetime Comparison 17-20
17.6.2.2 Binding Existing CORBA Objects to COM Views 17-21
17.6.2.3 Binding COM Objects to CORBA Views 17-22
17.6.2.4 COM View of CORBA Life Cycle 17-22
17.6.2.5 CORBA View of COM/Automation Life Cycle 17-23

17.7 Interworking Interfaces 17-23

17.7.1 SimpleFactory Interface 17-23

17.7.2 IMonikerProvider Interface and Moniker Use .. 17-23

17.7.3 ICORBAFactory Interface 17-24

17.7.4 IForeignObject Interface.................... 17-26

17.7.5 ICORBAObject Interface 17-27

17.7.6 ICORBAObject2 17-28

17.7.7 IORBObject Interface...................... 17-28

17.7.8 Naming Conventions for View Components 17-30

17.7.8.1 Naming the COM View Interface 17-30
17.7.8.2 Tag for the Automation Interface Id ... 17-30
17.7.8.3 Naming the Automation View Dispatch Interface 17-30
17.7.8.4 Naming the Automation View Dual Interface 17-31
17.7.8.5 Naming the Program Id for the COM Class 17-31
17.7.8.6 Naming the Class Id for the COM Class 17-32

17.8 Distribution 17-32

17.8.1 Bridge Locality........................... 17-32

17.8.2 Distribution Architecture 17-33

17.9 Interworking Targets 17-34

17.10 Compliance to COM/CORBA Interworking........... 17-34

17.10.1 Products Subject to Compliance............. 17-34

17.10.1.1 Interworking solutions 17-34
17.10.1.2 Mapping solutions 17-35

17.10.1.3 Mapped components 17-35
17.10.2 Compliance Points 17-36

18. Mapping: COM and CORBA 18-1

18.1 Data Type Mapping 18-1

18.2 CORBA to COM Data Type Mapping 18-2
18.2.1 Mapping for Basic Data Types 18-2
18.2.2 Mapping for Constants 18-2
18.2.3 Mapping for Enumerators 18-3
18.2.4 Mapping for String Types 18-4
18.2.4.1 Mapping for Unbounded String Types . 18-4
18.2.4.2 Mapping for Bounded String Types ... 18-5
18.2.5 Mapping for Struct Types 18-5
18.2.6 Mapping for Union Types 18-6
18.2.7 Mapping for Sequence Types 18-8
18.2.7.1 Mapping for Unbounded Sequence Types 18-8
18.2.7.2 Mapping for Bounded Sequence Types 18-8
18.2.8 Mapping for Array Types 18-9
18.2.9 Mapping for the any Type................... 18-9
18.2.10 Interface Mapping......................... 18-11
18.2.10.1 Mapping for interface identifiers 18-11
18.2.10.2 Mapping for exception types 18-11
18.2.10.3 Mapping for Nested Types 18-21
18.2.10.4 Mapping for Operations 18-22
18.2.10.5 Mapping for Oneway Operations 18-24
18.2.10.6 Mapping for Attributes 18-24
18.2.10.7 Indirection Levels for Operation Parameters 18-26
18.2.11 Inheritance Mapping...................... 18-26
18.2.12 Mapping for Pseudo-Objects 18-29
18.2.12.1 Mapping for TypeCode pseudo-object 18-29
18.2.12.2 Mapping for context pseudo-object ... 18-31
18.2.12.3 Mapping for principal pseudo-object . 18-32
18.2.13 Interface Repository Mapping 18-32

18.3 COM to CORBA Data Type Mapping 18-33
18.3.1 Mapping for Basic Data Types 18-33
18.3.2 Mapping for Constants 18-34
18.3.3 Mapping for Enumerators 18-34
18.3.4 Mapping for String Types 18-35
18.3.4.1 Mapping for unbounded string types ... 18-35
18.3.4.2 Mapping for bounded string types 18-36
18.3.4.3 Mapping for Unicode Unbounded String Types 18-36
18.3.4.4 Mapping for unicode bound string types 18-37
18.3.5 Mapping for Structure Types 18-37
18.3.6 Mapping for Union Types 18-38
18.3.6.1 Mapping for Encapsulated Unions 18-38
18.3.6.2 Mapping for nonencapsulated unions .. 18-39
18.3.7 Mapping for Array Types 18-40
18.3.7.1 Mapping for nonfixed arrays 18-40

18.3.7.2 Mapping for SAFEARRAY 18-40
18.3.8 Mapping for VARIANT..................... 18-41
18.3.9 Mapping for Pointers....................... 18-43
18.3.10 Interface Mapping......................... 18-44
18.3.10.1 Mapping for Interface Identifiers 18-44
18.3.10.2 Mapping for COM Errors 18-44
18.3.10.3 Mapping of Nested Data Types 18-47
18.3.10.4 Mapping of Names 18-47
18.3.10.5 Mapping for Operations 18-47
18.3.10.6 Mapping for Properties 18-48
18.3.11 Mapping for Read-Only Attributes 18-49
18.3.12 Mapping for Read-Write Attributes 18-49
18.3.12.1 Inheritance Mapping 18-50
18.3.12.2 Type Library Mapping 18-52

19. Mapping: Automation and CORBA 19-1

19.1 Mapping CORBA Objects to Automation 19-2
19.1.1 Architectural Overview..................... 19-2
19.1.2 Main Features of the Mapping............... 19-3

19.2 Mapping for Interfaces........................... 19-3
19.2.1 Mapping for Attributes and Operations 19-4
19.2.2 Mapping for OMG IDL Single Inheritance..... 19-5
19.2.3 Mapping of OMG IDL Multiple Inheritance.... 19-6

19.3 Mapping for Basic Data Types..................... 19-9
19.3.1 Basic Automation Types 19-9
19.3.2 Special Cases of Basic Data Type Mapping..... 19-10
19.3.2.1 Converting Automation long to CORBA unsigned long 19-10
19.3.2.2 Demoting CORBA unsigned long to Automation long 19-11
19.3.2.3 Demoting Automation long to CORBA unsigned short 19-11
19.3.2.4 Converting Automation boolean to CORBA boolean and CORBA boolean to Automation boolean 19-11
19.3.3 Mapping for Strings 19-11

19.4 IDL to ODL Mapping.............................19-12
19.4.1 A Complete IDL to ODL Mapping for the Basic Data Types 19-12

19.5 Mapping for Object References 19-15
19.5.1 Type Mapping 19-15
19.5.2 Object Reference Parameters and IForeignObject........................... 19-16

19.6 Mapping for Enumerated Types....................19-17

19.7 Mapping for Arrays and Sequences 19-18

19.8 Mapping for CORBA Complex Types 19-19
19.8.1 Mapping for Structure Types 19-20
19.8.2 Mapping for Union Types 19-21

19.8.3 Mapping for TypeCodes 19-22
19.8.4 Mapping for anys.......................... 19-24
19.8.5 Mapping for Typedefs...................... 19-25
19.8.6 Mapping for Constants..................... 19-25
19.8.7 Getting Initial CORBA Object References 19-26
19.8.8 Creating Initial in Parameters for Complex Types 19-27
  19.8.8.1 ITypeFactory Interface 19-29
  19.8.8.2 DIObjectInfo Interface 19-29
19.8.9 Mapping CORBA Exceptions to Automation Exceptions 19-30
  19.8.9.1 Overview of Automation Exception Handling 19-30
  19.8.9.2 CORBA Exceptions 19-30
  19.8.9.3 CORBA User Exceptions 19-31
  19.8.9.4 Operations that Raise User Exceptions . 19-32
  19.8.9.5 CORBA System Exceptions 19-33
  19.8.9.6 Operations that raise system exceptions 19-34
19.8.10 Conventions for Naming Components of the Automation View 19-36
19.8.11 Naming Conventions for Pseudo-Structs, Pseudo-Unions, and Pseudo-Exceptions 19-36
19.8.12 Automation View Interface as a Dispatch Interface (Nondual) 19-36
19.8.13 Aggregation of Automation Views 19-38
19.8.14 DII and DSI 19-38

19.9 Mapping Automation Objects as CORBA Objects....... 19-38
19.9.1 Architectural Overview.................... 19-38
19.9.2 Main Features of the Mapping 19-39
19.9.3 Getting Initial Object References............. 19-40
19.9.4 Mapping for Interfaces 19-40
19.9.5 Mapping for Inheritance.................... 19-40
19.9.6 Mapping for ODL Properties and Methods..... 19-41
19.9.7 Mapping for Automation Basic Data Types..... 19-42
  19.9.7.1 Basic automation types 19-42
19.9.8 Conversion Errors 19-43
19.9.9 Special Cases of Data Type Conversion 19-43
  19.9.9.1 Translating COM::Currency to Automation CURRENCY 19-43
  19.9.9.2 Translating CORBA double to Automation DATE 19-43
  19.9.9.3 Translating CORBA boolean to Automation boolean and Automation boolean to CORBA boolean 19-43
19.9.10 A Complete OMG IDL to ODL Mapping for the Basic Data Types 19-44
19.9.11 Mapping for Object References 19-46
19.9.12 Mapping for Enumerated Types.............. 19-47
19.9.13 Mapping for SafeArrays 19-48
  19.9.13.1 Multidimensional SafeArrays 19-48
19.9.14 Mapping for Typedefs...................... 19-48

19.9.15 Mapping for VARIANTs 19-48
19.9.16 Mapping Automation Exceptions to CORBA ... 19-49
19.10 Older Automation Controllers 19-49
19.10.1 Mapping for OMG IDL Arrays and Sequences to Collections 19-49
19.11 Example Mappings.................................. 19-51
19.11.1 Mapping the OMG Naming Service to Automation.............................. 19-51
19.11.2 Mapping a COM Service to OMG IDL 19-51
19.11.3 Mapping an OMG Object Service to Automation 19-55
20. Interoperability with non-CORBA Systems 20-1
20.1 Introduction 20-1
20.1.1 COM/CORBA Part A 20-2
20.2 Conformance Issues 20-2
20.2.1 Performance Issues 20-3
20.2.2 Scalability Issues 20-3
20.2.3 CORBA Clients for DCOM Servers........... 20-3
20.3 Locality of the Bridge 20-4
20.4 Extent Definition 20-5
20.4.1 Marshaling Constraints..................... 20-6
20.4.2 Marshaling Key 20-6
20.4.3 Extent Format 20-7
20.4.3.1 DVO_EXTENT 20-8
20.4.3.2 DVO_IFACE 20-8
20.4.3.3 DVO_IMPLDATA 20-8
20.4.3.4 DVO_BLOB 20-8
20.5 Request/Reply Extent Semantics 20-8
20.6 Consistency 20-9
20.6.1 IValueObject 20-10
20.6.2 ISynchronize and DISynchronize............. 20-11
20.6.2.1 Mode Property 20-11
20.6.2.2 SyncNow Method 20-11
20.6.2.3 ReCopy Method 20-11
20.7 DCOM Value Objects.................................20-11
20.7.1 Passing Automation Compound Types as DCOM Value Objects............................ 20-11
20.7.2 Passing CORBA-Defined Pseudo-Objects as DCOM Value Objects 20-12
20.7.3 IForeignObject........................... 20-12
20.7.4 DIForeignComplexType 20-12
20.7.5 DIForeignException........................ 20-12
20.7.6 DISystemException 20-12
20.7.7 DICORBAUserException 20-13
20.7.8 DICORBAStruct 20-13
20.7.9 DICORBAUnion 20-13

20.7.10 **DICORBATypeCode and ICORBATypeCode** ... 20-13
20.7.11 **DICORBAAny** ... 20-14
20.7.12 **ICORBAAny** ... 20-15
20.7.13 **User Exceptions In COM** ... 20-15
20.8 Chain Avoidance ... 20-16
20.8.1 **CORBA Chain Avoidance** ... 20-16
20.8.2 **COM Chain Avoidance** ... 20-17
20.9 Chain Bypass ... 20-19
20.9.1 **CORBA Chain Bypass** ... 20-19
20.9.2 **COM Chain Bypass** ... 20-20
20.10 Thread Identification ... 20-21
21. Portable Interceptors ... **21-1**
21.1 Introduction ... 21-1
21.1.1 **Object Creation** ... 21-2
21.1.2 **Client Sends Request** ... 21-3
21.1.3 **Server Receives Request** ... 21-4
21.1.4 **Server Sends Reply** ... 21-4
21.1.5 **Client Receives Reply** ... 21-5
21.2 Interceptor Interface ... 21-5
21.3 Request Interceptors ... 21-6
21.3.1 **Design Principles** ... 21-6
21.3.2 **General Flow Rules** ... 21-7
21.3.3 **The Flow Stack Visual Model** ... 21-8
21.3.4 **The Request Interceptor Points** ... 21-8
21.3.5 **Client-Side Interceptor** ... 21-9
21.3.6 **Client-Side Interception Points** ... 21-9
21.3.6.1 send_request ... 21-9
21.3.6.2 send_poll ... 21-9
21.3.6.3 receive_reply ... 21-10
21.3.6.4 receive_exception ... 21-10
21.3.6.5 receive_other ... 21-11
21.3.7 **Client-Side Interception Point Flow** ... 21-11
21.3.7.1 Client-side Flow Rules ... 21-11
21.3.7.2 Additional Client-side Details ... 21-12
21.3.7.3 Client-side Flow Examples ... 21-12
21.3.8 **Server-Side Interceptor** ... 21-14
21.3.9 **Server-Side Interception Points** ... 21-14
21.3.9.1 receive_request_service_contexts ... 21-14
21.3.9.2 receive_request ... 21-15
21.3.9.3 send_reply ... 21-15
21.3.9.4 send_exception ... 21-16
21.3.9.5 send_other ... 21-16
21.3.10 **Server-Side Interception Point Flow** ... 21-17
21.3.10.1 Server-side Flow Rules ... 21-17
21.3.10.2 Additional Server-side Details ... 21-17
21.3.10.3 Server-side Flow Examples ... 21-18
21.3.11 **Request Information** ... 21-20

21.3.12 RequestInfo Interface 21-21
21.3.12.1 request_id 21-21
21.3.12.2 operation 21-21
21.3.12.3 arguments 21-21
21.3.12.4 exceptions 21-22
21.3.12.5 contexts 21-22
21.3.12.6 operation_context 21-22
21.3.12.7 result 21-22
21.3.12.8 response_expected 21-23
21.3.12.9 sync_scope 21-23
21.3.12.10 reply_status 21-23
21.3.12.11 forward_reference 21-24
21.3.12.12 get_slot 21-24
21.3.12.13 get_request_service_context 21-25
21.3.12.14 get_reply_service_context 21-25
21.3.13 ClientRequestInfo Interface 21-25
21.3.13.1 target 1-27
21.3.13.2 effective_target 21-27
21.3.13.3 effective_profile 21-27
21.3.13.4 received_exception 21-27
21.3.13.5 received_exception_id 21-27
21.3.13.6 get_effective_component 21-27
21.3.13.7 get_effective_components 21-28
21.3.13.8 get_request_policy 21-28
21.3.13.9 add_request_service_context 21-28
21.3.14 ServerRequestInfo Interface.................. 21-29
21.3.14.1 sending_exception 21-30
21.3.14.2 object_id 21-30
21.3.14.3 adapter_id 21-31
21.3.14.4 target_most_derived_interface 21-31
21.3.14.5 get_server_policy 21-31
21.3.14.6 set_slot 21-31
21.3.14.7 target_is_a 21-31
21.3.14.8 add_reply_service_context 21-32
21.3.15 ForwardRequest Exception.................. 21-32

21.4 Portable Interceptor Current........................21-33
21.4.1 Overview................................ 21-33
21.4.2 Obtaining the Portable Interceptor Current.... 21-33
21.4.3 Portable Interceptor Current Interface........ 21-33
21.4.3.1 get_slot........................ 21-34
21.4.3.2 set_slot 21-34
21.4.4 Use of Portable Interceptor Current 21-34
21.4.4.1 Client-side use of PICurrent 21-34
21.4.4.2 Example of PICurrent to Handle Client-side Recursion 21-35
21.4.4.3 Server-side use of PICurrent 21-36
21.4.4.4 Request Scope vs Thread Scope 21-37
21.4.4.5 Flow of PICurrent between Scopes 21-37
21.4.4.6 Notes on PICurrent and Scopes 21-39

21.5 IOR Interceptor 21-39
21.5.1 Overview................................ 21-39
21.5.2 IORInterceptor Interface 21-39
21.5.2.1 establish_components 21-40

21.5.3 IORInfo Interface **21-40**
21.5.3.1 get_effective_policy 21-40
21.5.3.2 add_ior_component 21-41
21.5.3.3 add_ior_component_to_profile 21-41
21.6 PolicyFactory 21-42
21.6.1 PolicyFactory Interface **21-42**
21.6.1.1 create_policy 21-42
21.7 Registering Interceptors 21-42
21.7.1 ORBInitializer Interface **21-43**
21.7.1.1 pre_init 21-43
21.7.1.2 post_init 21-43
21.7.2 ORBInitInfo Interface **21-43**
21.7.2.1 DuplicateName Exception 21-44
21.7.2.2 InvalidName Exception 21-44
21.7.2.3 arguments 21-45
21.7.2.4 orb_id 21-45
21.7.2.5 codec_factory 21-45
21.7.2.6 register_initial_reference 21-45
21.7.2.7 resolve_initial_references 21-45
21.7.2.8 add_client_request_interceptor 21-45
21.7.2.9 add_server_request_interceptor 21-46
21.7.2.10 add_ior_interceptor 21-46
21.7.2.11 allocate_slot_id 21-46
21.7.2.12 register_policy_factory 21-46
21.7.3 register_orb_initializer Operation **21-47**
21.7.3.1 Mappings of register_orb_initializer . . . 21-47
21.7.4 Notes about Registering Interceptors **21-49**
21.8 Dynamic Initial References 21-49
21.8.1 register_initial_reference **21-49**
21.9 Module Dynamic 21-50
21.9.1 NVList PIDL Represented by ParameterList IDL **21-50**
21.9.2 ContextList PIDL Represented by ContextList IDL **21-50**
21.9.3 ExceptionList PIDL Represented by ExceptionList IDL **21-51**
21.9.4 Context PIDL Represented by RequestContext IDL **21-51**
21.10 Portable Interceptor IDL 21-51
22. CORBA Messaging **22-1**
22.1 Section I - Introduction 22-2
22.2 Messaging Quality of Service 22-2
22.2.1 Rebind Support **22-5**
22.2.1.1 typedef short RebindMode 22-5
22.2.1.2 interface RebindPolicy 22-5
22.2.2 Synchronization Scope **22-6**
22.2.2.1 typedef short SyncScope 22-6
22.2.2.2 interface SyncScopePolicy 22-7

21.5.3 IORInfo Interface 21-40
21.5.3.1 get_effective_policy 21-40
21.5.3.2 add_ior_component 21-41
21.5.3.3 add_ior_component_to_profile 21-41
21.6 PolicyFactory 21-42
21.6.1 PolicyFactory Interface..................... 21-42
21.6.1.1 create_policy 21-42
21.7 Registering Interceptors 21-42
21.7.1 ORBInitializer Interface.................... 21-43
21.7.1.1 pre_init 21-43
21.7.1.2 post_init 21-43
21.7.2 ORBInitInfo Interface...................... 21-43
21.7.2.1 DuplicateName Exception 21-44
21.7.2.2 InvalidName Exception 21-44
21.7.2.3 arguments 21-45
21.7.2.4 orb_id 21-45
21.7.2.5 codec_factory 21-45
21.7.2.6 register_initial_reference 21-45
21.7.2.7 resolve_initial_references 21-45
21.7.2.8 add_client_request_interceptor 21-45
21.7.2.9 add_server_request_interceptor 21-46
21.7.2.10 add_ior_interceptor 21-46
21.7.2.11 allocate_slot_id 21-46
21.7.2.12 register_policy_factory 21-46
21.7.3 register_orb_initializer Operation 21-47
21.7.3.1 Mappings of register_orb_initializer ... 21-47
21.7.4 Notes about Registering Interceptors.......... 21-49
21.8 Dynamic Initial References 21-49
21.8.1 register_initial_reference 21-49
21.9 Module Dynamic 21-50
21.9.1 NVList PIDL Represented by ParameterList IDL 21-50
21.9.2 ContextList PIDL Represented by ContextList IDL.......................... 21-50
21.9.3 ExceptionList PIDL Represented by ExceptionList IDL 21-51
21.9.4 Context PIDL Represented by RequestContext IDL 21-51
21.10 Portable Interceptor IDL 21-51
22. CORBA Messaging 22-1
22.1 Section I - Introduction 22-2
22.2 Messaging Quality of Service 22-2
22.2.1 Rebind Support 22-5
22.2.1.1 typedef short RebindMode 22-5
22.2.1.2 interface RebindPolicy 22-5
22.2.2 Synchronization Scope 22-6
22.2.2.1 typedef short SyncScope 22-6
22.2.2.2 interface SyncScopePolicy 22-7

22.2.3 Request and Reply Priority 22-7
- 22.2.3.1 struct PriorityRange 22-7
- 22.2.3.2 interface RequestPriorityPolicy 22-7
- 22.2.3.3 interface ReplyPriorityPolicy 22-8

22.2.4 Request and Reply Timeout 22-8
- 22.2.4.1 interface RequestStartTimePolicy 22-8
- 22.2.4.2 interface RequestEndTimePolicy 22-9
- 22.2.4.3 interface ReplyStartTimePolicy 22-9
- 22.2.4.4 interface ReplyEndTimePolicy 22-9
- 22.2.4.5 interface RelativeRequestTimeoutPolicy 22-9
- 22.2.4.6 interface RelativeRoundtripTimeout Policy 22-10

22.2.5 Routing 22-10
- 22.2.5.1 typedef short RoutingType 22-10
- 22.2.5.2 struct RoutingTypeRange 22-10
- 22.2.5.3 interface RoutingPolicy 22-11
- 22.2.5.4 interface MaxHopsPolicy 22-11

22.2.6 Queue Ordering 22-11
- 22.2.6.1 typedef short Ordering 22-11
- 22.2.6.2 interface QueueOrderPolicy 22-12

22.3 Propagation of Messaging QoS 22-12

22.3.1 Structures 22-12

22.3.2 Messaging QoS Profile Component 22-13

22.3.3 Messaging QoS Service Context 22-13

22.4 Section II - Introduction 22-13

22.5 Running Example 22-15

22.6 Async Operation Mapping 22-16

22.6.1 Callback Model Signatures (sendc) 22-16
- 22.6.1.1 Implied-IDL for Operations 22-16
- 22.6.1.2 Implied-IDL for Attributes 22-17
- 22.6.1.3 Example 22-17

22.6.2 Polling Model Signatures (sendp) 22-18
- 22.6.2.1 Implied-IDL for Operations 22-18
- 22.6.2.2 Implied-IDL for Attributes 22-19
- 22.6.2.3 Example 22-19

22.7 Exception Delivery in the Callback Model 22-20

22.7.1 Generic ExceptionHolder Value 22-20

22.7.2 Type-Specific ExceptionHolder Mapping 22-21

22.7.3 Example 22-21

22.8 Type-Specific ReplyHandler Mapping 22-22

22.8.1 ReplyHandler Operations for NO_EXCEPTION Replies 22-23

22.8.2 ReplyHandler Operations for Exceptional Replies 22-24

22.8.3 Example 22-24

22.9 Generic Poller Value 22-25

22.9.1 operation_target 22-26

22.9.2 operation_name 22-26

22.9.3 associated_handler 22-26

22.9.4 is_from_poller 22-26

22.10 Type-Specific Poller Mapping 22-26

22.10.1 Basic Type-Specific Poller 22-27

22.10.1.1 Poller operations for Interface operations 22-27

22.10.1.2 Poller operations for Interface attributes 22-28

22.10.2 Persistent Type-Specific Poller 22-29

22.10.3 Example 22-29

22.11 Example Programmer Usage 22-30

22.11.1 Example Programmer Usage (Examples Mapped to C++).......................... 22-30

22.11.2 Client-Side C++ Example for the Asynchronous Method Signatures........................ 22-31

22.11.3 Client-Side C++ Example of the Callback Model 22-32

22.11.3.1 C++ Example of Generated ExceptionHolder 22-32

22.11.3.2 C++ Example of Generated ReplyHandler 22-32

22.11.3.3 C++ Example of User-Implemented ReplyHandler 22-34

22.11.3.4 C++ Example of Callback Client Program 22-38

22.11.4 Client-Side C++ Example of the Polling Model.. 22-39

22.11.4.1 C++ Example of Generated Poller ... 22-39

22.11.4.2 C++ Example of Polling Client Program 22-40

22.11.4.3 C++ Example of Using PollableSet in a Client Program 22-42

22.11.5 Server Side 22-44

22.12 Section III - Introduction 22-45

22.13 Routing Object References......................... 22-46

22.14 Message Routing 22-47

22.14.1 Structures............................... 22-49

22.14.1.1 MessageBody 22-49

22.14.1.2 RequestMessage 22-49

22.14.1.3 ReplyDestination 22-50

22.14.1.4 RequestInfo 22-50

22.14.2 Interfaces 22-51

22.14.2.1 ReplyHandler................... 22-51

22.14.2.2 Router 22-51

22.14.2.3 send_request 22-51

22.14.2.4 send_multiple_requests 22-51

22.14.2.5 UntypedReplyHandler 22-51

22.14.2.6 reply 22-51

22.14.2.7 PersistentRequest................. 22-52

22.14.2.8 readonly attribute reply_available 22-52

22.14.2.9 get_reply 22-52

22.14.2.10 attribute associated_handler 22-52

22.14.2.11 PersistentRequestRouter 22-53

22.14.2.12 create_persistent_request 22-53

22.14.3 Routing Protocol 22-53
  22.14.3.1 Invoking Client 22-54
  22.14.3.2 Initial Request Router 22-55
  22.14.3.3 Request Routing Algorithm 22-55
  22.14.3.4 Intermediate Request Router 22-56
  22.14.3.5 Target Router 22-56
  22.14.3.6 Replying to a Type-specific ReplyHandler 22-58
  22.14.3.7 Replying to an UntypedReplyHandler 22-58
  22.14.3.8 Handling of Service Contexts 22-58
  22.14.3.9 Handling LOCATION_FORWARD Replies 22-59
  22.14.3.10 Routing of Replies 22-59
  22.14.3.11 UntypedReplyHandler 22-59

22.15 Router Administration 22-60

22.15.1 Constants 22-63
  22.15.1.1 typedef short RegistrationState 22-63
22.15.2 Exceptions 22-64
  22.15.2.1 exception InvalidState 22-64
22.15.3 Valuetypes 22-64
  22.15.3.1 RetryPolicy 22-64
  22.15.3.2 ImmediateSuspend 22-64
  22.15.3.3 UnlimitedPing 22-64
  22.15.3.4 LimitedPing 22-64
  22.15.3.5 DecayPolicy 22-65
  22.15.3.6 ResumePolicy 22-65
22.15.4 Interfaces 22-65
  22.15.4.1 RouterAdmin 22-65
  22.15.4.2 register_destination 22-65
  22.15.4.3 suspend_destination 22-65
  22.15.4.4 resume_destination 22-65
  22.15.4.5 unregister_destination 22-66

23. Minimum CORBA 23-1

23.1 Introduction 23-2

23.2 IDL 23-2

23.3 CORBA Omitted Features 23-2

23.4 ORB Interface Omissions 23-3

23.4.1 ORB 23-3
23.4.2 Object 23-4
23.4.3 ConstructionPolicy 23-4

23.5 Dynamic Invocation Interface 23-5

23.6 Dynamic Skeleton Interface 23-5

23.7 Dynamic Any 23-5

23.8 Interface Repository 23-5

23.8.1 TypeCode 23-5

23.9 Portable Object Adapter 23-6

23.9.1 Interfaces 23-6
  23.9.1.1 POA 23-6

23.9.1.2 Current 23-6
23.9.1.3 Policy interfaces 23-7
23.9.1.4 POAManager 23-7
23.9.1.5 AdapterActivator 23-7
23.9.1.6 ServantManagers 23-7
23.9.2 Policies 23-7
23.9.2.1 ThreadPolicy 23-7
23.9.2.2 LifespanPolicy 23-8
23.9.2.3 ObjectIdUniquenessPolicy 23-8
23.9.2.4 IdAssignmentPolicy 23-8
23.9.2.5 ServantRetentionPolicy 23-8
23.9.2.6 RequestProcessingPolicy 23-8
23.9.2.7 ImplicitActivationPolicy 23-9

23.10 Interoperability................................. 23-9
23.10.1 DCE Interoperability....................... 23-9

23.11 COM/CORBA Interworking....................... 23-10

23.12 Interceptors....................................23-10

23.13 Language Mappings.............................23-10
23.13.1 C++ Mapping Specific Issues 23-10
23.13.2 Java Mapping Specific Issues 23-10

23.14 minimumCORBA OMG IDL.......................23-11
23.14.1 ORB Interface 23-11
23.14.2 Dynamic Invocation Interface................ 23-14
23.14.3 Dynamic Skeleton Interface 23-14
23.14.4 Dynamic Management of Any Values 23-14
23.14.5 Interface Repository 23-14
23.14.6 Portable Object Adapter.................... 23-22
23.14.7 Interceptors 23-29

24. Real-Time CORBA 24-1

24.1 Goals of the Specification............................ 24-2

24.2 Extending CORBA................................ 24-3

24.3 Approach to Real-Time CORBA 24-3
24.3.1 The Nature of Real-Time.................... 24-3
24.3.2 Meeting Real-Time Requirements 24-4
24.3.3 activities 24-4
24.3.4 End-to-End Predictability................... 24-5
24.3.5 Management of Resources 24-6

24.4 Compatibility 24-6
24.4.1 Interoperability 24-6
24.4.2 Portability................................ 24-7
24.4.3 CORBA - Real-Time CORBA Interworking.... 24-7

24.5 Real-Time CORBA Architectural Overview 24-7
24.5.1 Real-Time CORBA Modules................. 24-8
24.5.2 Real-Time ORB 24-8
24.5.3 Thread Scheduling........................ 24-9

24.5.4 Real-Time CORBA Priority 24-9
24.5.5 Native Priority and PriorityMappings......... 24-9
24.5.6 Real-Time CORBA Current................. 24-9
24.5.7 Priority Models 24-10
24.5.8 Real-Time CORBA Mutexes and Priority Inheritance .24-10
24.5.9 Threadpools 24-10
24.5.10 Priority Banded Connections 24-11
24.5.11 Non-Multiplexed Connections 24-11
24.5.12 Invocation Timeouts 24-11
24.5.13 Client and Server Protocol Configuration...... 24-11
24.5.14 Real-Time CORBA Configuration 24-11
24.5.15 Scheduling Service......................... 24-12

24.6 Real-Time ORB 24-12
24.6.1 Real-Time ORB Initialization................ 24-13
24.6.2 Real-Time CORBA System Exceptions 24-13

24.7 Real-Time POA 24-14

24.8 Native Thread Priorities 24-15

24.9 CORBA Priority................................. 24-16

24.10 CORBA Priority Mappings 24-16
24.10.1 C Language binding for PriorityMapping...... 24-17
24.10.2 C++ Language binding for PriorityMapping ... 24-17
24.10.3 Ada Language binding for PriorityMapping.... 24-18
24.10.4 Java Language binding for PriorityMapping ... 24-18
24.10.5 Semantics 24-18

24.11 Real-Time Current 24-19

24.12 Real-Time CORBA Priority Models................. 24-20
24.12.1 PriorityModelPolicy 24-20
24.12.2 Scope of PriorityModelPolicy................ 24-21
24.12.3 Client Propagated Priority Model 24-22
24.12.4 Server Declared Priority Model.............. 24-23
24.12.5 Setting Server Priority on a per-Object Reference Basis 24-23

24.13 Priority Transforms 24-25
24.13.1 C Language Binding for PriorityTransform 24-26
24.13.2 C++ Language Binding for PriorityTransform.. 24-26
24.13.3 Ada Language binding for PriorityTransform .. 24-27
24.13.4 Java Language binding for PriorityTransform.. 24-27
24.13.5 Semantics 24-27

24.14 Mutex Interface 24-28

24.15 Threadpools 24-29
24.15.1 Creation of Threadpool without Lanes 24-31
24.15.2 Creation of Threadpool with Lanes 24-32
24.15.3 Request Buffering 24-32

24.15.4 **Scope of ThreadpoolPolicy** 24-33

24.16 Implicit and Explicit Binding 24-33

24.17 Priority Banded Connections 24-34

24.17.1 **Scope of PriorityBandedConnectionPolicy** 24-35

24.17.2 **Binding of Priority Banded Connection** 24-36

24.18 PrivateConnectionPolicy 24-37

24.19 Invocation Timeout 24-38

24.20 Protocol Configuration 24-38

24.20.1 **ServerProtocolPolicy** 24-39

24.20.2 **Scope of ServerProtocolPolicy** 24-41

24.20.3 **ClientProtocolPolicy** 24-41

24.20.4 **Scope of ClientProtocolPolicy** 24-42

24.20.5 **Protocol Configuration Semantics** 24-42

24.21 Consolidated IDL 24-43

24.22 Introduction 24-48

24.23 IDL 24-49

24.24 Semantics 24-50

24.25 Example 24-51

24.25.1 **Server C++ Example Code** 24-51

24.25.2 **Client C++ Example Code** 24-52

24.25.3 **Explanation of Example** 24-53

25. Fault Tolerant CORBA **25-1**

25.1 Fault Tolerant CORBA 25-1

25.1.1 **Fault Tolerance for Diverse Applications** 25-1

25.1.2 **Objectives** 25-2

25.1.3 **Basic Concepts** 25-3

25.1.3.1 Replication and Object Groups 25-3

25.1.3.2 Fault Tolerance Domains 25-3

25.1.3.3 Fault Tolerance Properties 25-3

25.1.3.4 Strong Replica Consistency 25-4

25.1.4 **Architectural Overview** 25-4

25.1.4.1 Fault Tolerance Property Management . 25-6

25.1.4.2 Replication Management 25-6

25.1.4.3 Fault Detection and Notification 25-7

25.1.4.4 Logging and Recovery 25-7

25.1.5 **Requirements** 25-8

25.1.6 **Limitations** 25-11

25.2 Basic Fault Tolerance Mechanisms 25-12

25.2.1 **Overview** 25-12

25.2.2 **Interoperable Object Group References** 25-13

25.2.2.1 TAG_FT_GROUP Component 25-14

25.2.2.2 TAG_FT_PRIMARY Component 25-16

25.2.3 **Interoperable Object Group Reference Operations** 25-16

25.2.3.1 get_interface 25-17
25.2.3.2 is_a 25-17
25.2.3.3 is_nil 25-17
25.2.3.4 non_existent 25-17
25.2.3.5 is_equivalent 25-17
25.2.3.6 hash 25-18
25.2.3.7 create_request 25-18
25.2.3.8 get_policy 25-18
25.2.3.9 get_domain_managers 25-18
25.2.3.10 set_policy_overrides 25-18

25.2.4 Modes of Profile Addressing 25-18

25.2.4.1 Profiles That Address Object Group Members 25-18
25.2.4.2 Profiles That Address Gateways 25-19
25.2.4.3 Choice of Profile Addressing Mode ... 25-19

25.2.5 Accessing Server Object Groups 25-19

25.2.5.1 Access via IIOP Directly to the Primary Member 25-20
25.2.5.2 Access via IIOP and a Gateway 25-20
25.2.5.3 Access via a Multicast Group Communication Protocol 25-20

25.2.6 Extensions to CORBA Failover Semantics 25-21

25.2.7 Most Recent Object Group Reference 25-22

25.2.7.1 FT_GROUP_VERSION Service Context 25-22

25.2.8 Transparent Reinvocation 25-23

25.2.8.1 FT_REQUEST Service Context 25-24
25.2.8.2 Request Duration Policy 25-26
25.2.8.3 Fault Handling for GIOP Messages ... 25-26

25.2.9 Transport Heartbeats 25-27

25.2.9.1 TAG_FT_HEARTBEAT_ENABLED Component 25-28
25.2.9.2 Heartbeat Policy 25-28
25.2.9.3 Heartbeat Enabled Policy 25-30

25.3 Replication Management 25-31

25.3.1 Overview 25-31

25.3.2 Fault Tolerance Properties 25-32

25.3.2.1 ReplicationStyle 25-32
25.3.2.2 MembershipStyle 25-33
25.3.2.3 ConsistencyStyle 25-34
25.3.2.4 FaultMonitoringStyle 25-35
25.3.2.5 FaultMonitoringGranularity 25-35
25.3.2.6 Factories 25-36
25.3.2.7 InitialNumberReplicas 25-36
25.3.2.8 MinimumNumberReplicas 25-36

25.3.3 FaultMonitoringIntervalAndTimeout 25-37

25.3.4 CheckpointInterval 25-37

25.3.5 Common Types 25-38

25.3.5.1 Identifiers 25-40
25.3.5.2 Exceptions 25-42

25.3.6 Replication Manager 25-44

25.3.6.1 Operations 25-44

25.3.7 PropertyManager 25-45

25.3.7.1 Operations 25-46

25.3.7.2 get_properties 25-49

25.3.8 ObjectGroupManager 25-49

25.3.8.1 Operations 25-50

25.3.9 GenericFactory 25-56

25.3.9.1 Identifiers 25-59

25.3.9.2 Operations 25-59

25.3.10 Obtaining the Reference for the Replication Manager 25-61

25.3.11 Use Cases 25-61

25.3.11.1 Infrastructure-Controlled Membership Style 25-61

25.3.11.2 Application-Controlled Membership Style 25-63

25.3.11.3 Unreplicated Object Creation and Deletion 25-65

25.4 Fault Management 25-66

25.4.1 Overview 25-66

25.4.2 Architecture 25-67

25.4.2.1 Fault Detection 25-68

25.4.2.2 Fault Notification 25-68

25.4.2.3 Fault Analysis 25-68

25.4.2.4 Scalability 25-68

25.4.2.5 Deployment of Fault Detectors 25-69

25.4.3 Connecting Fault Detectors to Applications 25-70

25.4.4 Pull-Based Monitoring 25-71

25.4.4.1 PULL Fault Monitoring Style 25-71

25.4.4.2 PullMonitorable Interface 25-71

25.4.5 Fault Event Types 25-72

25.4.5.1 ObjectCrashFault 25-72

25.4.6 Fault Notifier 25-73

25.4.6.1 Identifiers 25-75

25.4.6.2 Operations 25-75

25.4.6.3 Filtering 25-77

25.4.6.4 Mapping of the Fault Notifier to the CosNotification Service 25-78

25.4.7 Use Cases 25-79

25.4.7.1 The Fault Detector as a Fault Notification Supplier 25-79

25.4.7.2 The Replication Manager as a Fault Notification Consumer 25-80

25.5 Logging & Recovery Management.................. 25-81

25.5.1 Overview 25-81

25.5.2 Logging Mechanism 25-81

25.5.3 Recovery Mechanism 25-82

25.5.4 Checkpointable and Updateable Interfaces 25-84

25.5.4.1 Identifiers 25-85

25.5.4.2 Exceptions 25-85

25.5.4.3 Operations 25-86

25.5.4.4 set_update 25-87

25.5.5 Use Case 25-87

25.5.5.1 Infrastructure-Controlled Consistency Style 25-87

26. Secure Interoperability 26-1

26.1 Overview 26-2

26.1.1 Assumptions 26-3

26.2 Protocol Message Definitions 26-4

26.2.1 The Security Attribute Service Context Element 26-4

26.2.2 SAS context_data Message Body Types 26-5

26.2.2.1 EstablishContext Message Format 26-5
26.2.2.2 ContextError Message Format 26-7
26.2.2.3 CompleteEstablishContext Message Format 26-7
26.2.2.4 MessageInContext Message Format ... 26-9

26.2.3 Authorization Token Format 26-10

26.2.3.1 Extensions of the IETF AC Profile for CSIv2 26-11

26.2.4 Client Authentication Token Format 26-11

26.2.4.1 Username Password GSS Mechanism (GSSUP) 26-12

26.2.5 Identity Token Format 26-14

26.2.6 Principal Names and Distinguished Names 26-15

26.3 Security Attribute Service Protocol 26-16

26.3.1 Compound Mechanisms 26-16

26.3.1.1 Context Validation 26-17
26.3.1.2 Legend for Request Principal Interpretations 26-18
26.3.1.3 Anonymous Identity Assertion 26-19
26.3.1.4 Presumed Trust 26-19
26.3.1.5 Failed Trust Evaluations 26-19
26.3.1.6 Request Principal Interpretations 26-20

26.3.2 Session Semantics 26-21

26.3.2.1 Negotiation of Statefulness 26-21
26.3.2.2 Stateful/Reusable Contexts 26-22

26.3.3 TSS State Machine 26-23

26.3.3.1 TSS State Machine Actions 26-25

26.3.4 CSS State Machine 26-27

26.3.4.1 CSS State Machine Actions 26-30

26.3.5 ContextError Values and Exceptions 26-30

26.4 Transport Security Mechanisms 26-31

26.4.1 Transport Layer Interoperability 26-31

26.4.2 Transport Mechanism Configuration 26-31

26.4.2.1 Recommended SSL/TLS Ciphersuites . 26-31

26.5 Interoperable Object References 26-32

26.5.1 Target Security Configuration 26-32

26.5.1.1 AssociationOptions Type 26-33
26.5.1.2 Transport Address 26-35
26.5.1.3 TAG_TLS_SEC_TRANS 26-35
26.5.1.4 TAG_SECIOP_SEC_TRANS 26-37
26.5.1.5 TAG_CSI_SEC_MECH_LIST 26-38
26.5.1.6 TAG_NULL_TAG 26-43

26.5.2 Client-side Mechanism Selection 26-43

26.5.3 Client-Side Requirements and Location Binding 26-44

26.5.3.1 Comments on Establishing Trust in Client 26-45

26.6 Conformance Levels 26-45

26.6.1 Conformance Level 0 26-45

26.6.1.1 Transport-Layer Requirements 26-45

26.6.1.2 Service Context Protocol Requirements 26-46

26.6.1.3 Interoperable Object References (IORs) 26-47

26.6.2 Conformance Level 1 26-47

26.6.2.1 Authorization Tokens 26-47

26.6.3 Conformance Level 2 26-47

26.6.3.1 Authorization-Token-Based Delegation 26-47

26.6.4 Stateful Conformance 26-48

26.7 Sample Message Flows and Scenarios 26-48

26.7.1 Confidentiality, Trust in Server, and Trust in Client Established in the Connection 26-49

26.7.1.1 Sample IOR Configuration 26-50

26.7.2 Confidentiality and Trust in Server Established in the Connection - Stateless Trust in Client Established in Service Context 26-51

26.7.2.1 Sample IOR Configuration 26-52

26.7.3 Confidentiality, Trust in Server, and Trust in Client Established in the Connection - Stateless Trust Association Established in Service Context 26-53

26.7.3.1 Sample IOR Configuration 26-54

26.7.3.2 Validating the Trusted Server 26-54

26.7.3.3 Presuming the Security of the Connection 26-55

26.7.4 Confidentiality, Trust in Server, and Trust in Client Established in the Connection - Stateless Forward Trust Association Established in Service Context 26-56

26.7.4.1 Sample IOR Configuration 26-57

26.8 References for this Chapter 26-57

26.9 IDL 26-58

26.9.1 Module IOP 26-58

26.9.1.1 New Types Defined for CSIv2 26-58

26.9.2 Module GSSUP - Username/Password GSSAPI Token Formats 26-58

26.9.3 Module CSI - Common Secure Interoperability . 26-59

26.9.4 Module CSIIOP - CSIv2 IOR Component Tag Definitions 26-63

Appendix A - OMG IDL Tags A-1

Glossary 1

Index 1

Preface

About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd., this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd. ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems. X/Open's strategy for achieving its mission is to combine existing and emerging standards into a comprehensive, integrated systems environment called the Common Applications Environment (CAE).

The components of the CAE are defined in X/Open CAE specifications. These contain, among other things, an evolving portfolio of practical application programming interfaces (APIs), which significantly enhance portability of application programs at the source code level. The APIs also enhance the interoperability of applications by providing definitions of, and references to, protocols and protocol profiles.

The X/Open specifications are also supported by an extensive set of conformance tests and by the X/Open trademark (XPG brand), which is licensed by X/Open and is carried only on products that comply with the CAE specifications.

Intended Audience

The architecture and specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for the Object Request Broker (ORB). The benefit of compliance is, in general, to be able to produce interoperable applications that are based on distributed, interoperating objects. As defined by the Object Management Group (OMG) in the *Object Management Architecture Guide,* the ORB provides the mechanisms by which objects transparently make requests and receive responses. Hence, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

Context of CORBA

The key to understanding the structure of the CORBA architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in this manual.
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in *CORBAservices: Common Object Services Specification*.

- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility. Information about Common Facilities will be contained in *CORBAfacilities: Common Facilities Architecture*.
- **Application Objects,** which are products of a single vendor on in-house development group that controls their interfaces. Application Objects correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model.

The Object Request Broker, then, is the core of the Reference Model. It is like a telephone exchange, providing the basic mechanism for making and receiving calls. Combined with the Object Services, it ensures meaningful communication between CORBA-compliant applications.

Associated Documents

The CORBA documentation set includes the following books:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBAservices: Common Object Services Specification* contains specifications for the Object Services.
- *CORBAfacilities: Common Facilities Architecture* contains the architecture for Common Facilities.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote.

To obtain books in the documentation set, or other OMG publications, refer to the enclosed subscription card or contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren't required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding specified in the *C++ Language Mapping Specification.*

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to "Compliance to COM/CORBA Interworking" on page 17-34.

As described in the *OMA Guide*, the OMG's Core Object Model consists of a core and components. Likewise, the body of *CORBA* specifications is divided into core and component-like specifications. The structure of this manual reflects that division.

The *CORBA* core specifications are categorized as follows:

CORBA Core, as specified in Chapters 1-11

CORBA Interoperability, as specified in Chapters 12-16

CORBA Interworking, as specified in Chapters 17-21

CORBA Quality of Service, as specified in Chapters 22-26

Note – The CORBA Language Mappings have been separated from the CORBA Core and each language mapping is its own separate book. Refer to CORBA Language Mappings at the OMG Formal Document web area for this information.

Structure of This Manual

This manual is divided into the categories of Core, Interoperability, and Interworking. These divisions reflect the compliance points of CORBA. In addition to this preface, CORBA: Common Object Request Broker Architecture and Specification contains the following chapters:

Core

Chapter 1 - The Object Model describes the computation model that underlies the CORBA architecture.

Chapter 2 - CORBA Overview contains the overall structure of the ORB architecture and includes information about CORBA interfaces and implementations.

Chapter 3 - OMG IDL Syntax and Semantics details the OMG interface definition language (OMG IDL), which is the language used to describe the interfaces that client objects call and object implementations provide.

Chapter 4 - ORB Interface defines the interface to the ORB functions that do not depend on object adapters: these operations are the same for all ORBs and object implementations.

Chapter 5 - Value Type Semantics describes the semantics of passing an object by value, which is similar to that of standard programming languages.

Chapter 6 - Abstract Interface Semantics explains an IDL abstract interface, which provides the capability to defer the determination of whether an object is passed by reference or by value until runtime.

Chapter 7 - The Dynamic Invocation Interface details the DII, the client's side of the interface that allows dynamic creation and invocation of request to objects.

Chapter 8 -- The Dynamic Skeleton Interface describes the DSI, the server's-side interface that can deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. DSI is the server's analogue of the client's Dynamic Invocation Interface (DII).

Chapter 9 - Dynamic Management of Any Values details the interface for the Dynamic Any type. This interface allows statically-typed programming languages such as C and Java to create or receive values of type Any without compile-time knowledge that the typer contained in the Any.

Chapter 10 - Interface Repository explains the component of the ORB that manages and provides access to a collection of object definitions.

Chapter 11 - Portable Object Adapter defines a group of IDL interfaces than an implementation uses to access ORB functions.

Interoperability

Chapter 12 - Interoperability Overview describes the interoperability architecture and introduces the subjects pertaining to interoperability: inter-ORB bridges; general and Internet inter-ORB protocols (GIOP and IIOP); and environment-specific, inter-ORB protocols (ESIOPs).

Chapter 13 - ORB Interoperability Architecture introduces the framework of ORB interoperability, including information about domains; approaches to inter-ORB bridges; what it means to be compliant with ORB interoperability; and ORB Services and Requests.

Chapter 14 - Building Inter-ORB Bridges explains how to build bridges for an implementation of interoperating ORBs.

Chapter 15 - General Inter-ORB Protocol describes the general inter-ORB protocol (GIOP) and includes information about the GIOP's goals, syntax, format, transport, and object location. This chapter also includes information about the Internet inter-ORB protocol (IIOP).

Chapter 16 - DCE ESIOP - Environment-Specific Inter-ORB Protocol (ESIOP) details a protocol for the OSF DCE environment. The protocol is called the DCE Environment Inter-ORB Protocol (DCE ESIOP).

Interworking

Chapter 17 - Interworking Architecture describes the architecture for communication between two object management systems: Microsoft's COM (including OLE) and the OMG's CORBA.

Chapter 18 - Mapping: COM and CORBA explains the data type and interface mapping between COM and CORBA. The mappings are described in the context of both Win16 and Win32 COM.

Chapter 19 - Mapping: OLE Automation and CORBA details the two-way mapping between OLE Automation (in ODL) and CORBA (in OMG IDL).

Note: Chapter 19 also includes an appendix describing solutions that vendors might implement to support existing and older OLE Automation controllers and an appendix that provides an example of how the Naming Service could be mapped to an OLE Automation interface according to the Interworking specification.

Chapter 20 - Interoperability with non-CORBA Systems describes the effective access to CORBA servers through DCOM and the reverse.

Chapter 21 - Portable Interceptors defines ORB operations that allow services such as security to be inserted in the invocation path.

Quality of Service (QoS)

Chapter 22 - CORBA Messaging includes three general topics: Quality of Service, Asynchronous Method Invocations (to include Time-Independent or "Persistent" Requests), and the specification of interoperable Routing interfaces to support the transport of requests asynchronously from the handling of their replies.

Chapter 23 - Minimum CORBA describes minimumCORBA, a subset of CORBA designed for systems with limited resources.

Chapter 24 - Real-Time CORBA defines an optional set of extensions to CORBA tailored to equip ORBs to be used as a component of a Real-Time system.

Chapter 25 - Fault Tolerant CORBA describes Fault Tolerant systems, basic fault tolerance mechanisms, replication management, and logging and recovery management.

Chapter 26 - Common Secure Interoperability defines the CORBA Security Attribute Service (SAS) protocol and its use within the CSIv2 architecture to address the requirements of CORBA security for interoperable authentication, delegation, and privileges.

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgements

The following companies submitted and/or supported parts of the specifications that were approved by the Object Management Group to become *CORBA:*

- Adiron, LLC
- Alcatel
- BEA Systems, Inc.
- BNR Europe Ltd.
- Borland International, Inc.
- Compaq Computer Corporation
- Concept Five Technologies
- Cooperative Research Centre for Distributed Systems Technology (DSTC)
- Defense Information Systems Agency
- Digital Equipment Corporation
- Ericsson
- Eternal Systems, Inc.
- Expersoft Corporation
- France Telecom
- FUJITSU LIMITED
- Genesis Development Corporation
- Gensym Corporation
- Hewlett-Packard Company
- HighComm
- Highlander Communications, L.C.
- Humboldt-University
- HyperDesk Corporation
- ICL, Plc.
- Inprise Corporation
- International Business Machines Corporation
- International Computers, Inc.

- IONA Technologies, Plc.
- Lockheed Martin Federal Systems, Inc.
- Lucent Technologies, Inc.
- Micro Focus Limited
- MITRE Corporation
- Motorola, Inc.
- NCR Corporation
- NEC Corporation
- Netscape Communications Corporation
- Nortel Networks
- Northern Telecom Corporation
- Novell, Inc.
- Object Design, Inc.
- Objective Interface Systems, Inc.
- Object-Oriented Concepts, Inc.
- OC Systems, Inc.
- Open Group - Open Software Foundation
- Oracle Corporation
- PeerLogic, Inc.
- Persistence Software, Inc.
- Promia, Inc.
- Siemens Nixdorf Informationssysteme AG
- SPAWAR Systems Center
- Sun Microsystems, Inc.
- SunSoft, Inc.
- Sybase, Inc.
- Telefónica Investigación y Desarrollo S.A. Unipersonal
- TIBCO, Inc.
- Tivoli Systems, Inc.
- Tri-Pacific Software, Inc.
- University of California, Santa Barbara
- University of Rhode Island
- Visual Edge Software, Ltd.
- Washington University

In addition to the preceding contributors, the OMG would like to acknowledge Mark Linton at Silicon Graphics and Doug Lea at the State University of New York at Oswego for their work on the C++ mapping.

References

IDL Type Extensions RFP, March 1995. OMG TC Document 95-1-35.

The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998.

CORBAservices: Common Object Services Specification, Revised Edition, OMG TC Document 95-3-31.

COBOL Language Mapping RFP, December 1995. OMG TC document 95-12-10.

COBOL 85 ANSI X3.23-1985 / ISO 1989-1985.

IEEE Standard for Binary Floating-Point Arithmetic, ANIS/IEEE Std 754-1985.

XDR: External Data Representation Standard, RFC1832, R. Srinivasan, Sun Microsystems, August 1995.

OSF Character and Code Set Registry, OSF DCE SIG RFC 40.1 (Public Version), S. (Martin) O'Donnell, June 1994.

RPC Runtime Support For I18N Characters — Functional Specification, OSF DCE SIG RFC 41.2, M. Romagna, R. Mackey, November 1994.

X/Open System Interface Definitions, Issue 4 Version 2, 1995.

Interoperability Overview 12

Contents

This chapter contains the following sections.

Section Title	Page
“Elements of Interoperability”	12-1
“Relationship to Previous Versions of CORBA”	12-4
“Examples of Interoperability Solutions”	12-5
“Motivating Factors”	12-8
“Interoperability Design Goals”	12-9

ORB interoperability specifies a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant ORBs. The approach to “interORBability” is universal, because its elements can be combined in many ways to satisfy a very broad range of needs.

12.1 Elements of Interoperability

The elements of interoperability are as follows:

- ORB interoperability architecture
- Inter-ORB bridge support
- General and Internet inter-ORB Protocols (GIOPs and IIOPs)

In addition, the architecture accommodates environment-specific inter-ORB protocols (ESIOPs) that are optimized for particular environments such as DCE.

12.1.1 ORB Interoperability Architecture

The ORB Interoperability Architecture provides a conceptual framework for defining the elements of interoperability and for identifying its compliance points. It also characterizes new mechanisms and specifies conventions necessary to achieve interoperability between independently produced ORBs.

Specifically, the architecture introduces the concepts of *immediate* and *mediated bridging* of ORB domains. The Internet Inter-ORB Protocol (IIOP) forms the common basis for broad-scope mediated bridging. The inter-ORB bridge support can be used to implement both immediate bridges and to build "half-bridges" to mediated bridge domains.

By use of bridging techniques, ORBs can interoperate without knowing any details of that ORB's implementation, such as what particular IPC or protocols (such as ESIOPs) are used to implement the *CORBA* specification.

The IIOP may be used in bridging two or more ORBs by implementing "half bridges" that communicate using the IIOP. This approach works for both stand-alone ORBs, and networked ones that use an ESIOP.

The IIOP may also be used to implement an ORB's internal messaging, if desired. Since ORBs are not required to use the IIOP internally, the goal of not requiring prior knowledge of each others' implementation is fully satisfied.

12.1.2 Inter-ORB Bridge Support

The interoperability architecture clearly identifies the role of different kinds of domains for ORB-specific information. Such domains can include object reference domains, type domains, security domains (e.g., the scope of a *Principal* identifier), a transaction domain, and more.

Where two ORBs are in the same domain, they can communicate directly. In many cases, this is the preferable approach. This is not always true, however, since organizations often need to establish local control domains.

When information in an invocation must leave its domain, the invocation must traverse a bridge. The role of a bridge is to ensure that content and semantics are mapped from the form appropriate to one ORB to that of another, so that users of any given ORB only see their appropriate content and semantics.

The inter-ORB bridge support element specifies ORB APIs and conventions to enable the easy construction of interoperability bridges between ORB domains. Such bridge products could be developed by ORB vendors, Sieves, system integrators, or other third-parties.

Because the extensions required to support Inter-ORB Bridges are largely general in nature, do not impact other ORB operation, and can be used for many other purposes besides building bridges, they are appropriate for all ORBs to support. Other applications include debugging, interposing of objects, implementing objects with interpreters and scripting languages, and dynamically generating implementations.

The inter-ORB bridge support can also be used to provide interoperability with non-CORBA systems, such as Microsoft's Component Object Model (COM). The ease of doing this will depend on the extent to which those systems conform to the CORBA Object Model.

12.1.3 General Inter-ORB Protocol (GIOP)

The General Inter-ORB Protocol (GIOP) element specifies a standard transfer syntax (low-level data representation) and a set of message formats for communications between ORBs. The GIOP is specifically built for ORB to ORB interactions and is designed to work directly over any connection-oriented transport protocol that meets a minimal set of assumptions. It does not require or rely on the use of higher level RPC mechanisms. The protocol is simple, scalable and relatively easy to implement. It is designed to allow portable implementations with small memory footprints and reasonable performance, with minimal dependencies on supporting software other than the underlying transport layer.

While versions of the GIOP running on different transports would not be directly interoperable, their commonality would allow easy and efficient bridging between such networking domains.

12.1.4 Internet Inter-ORB Protocol (IIOP)

The Internet Inter-ORB Protocol (IIOP) element specifies how GIOP messages are exchanged using TCP/IP connections. The IIOP specifies a standardized interoperability protocol for the Internet, providing "out of the box" interoperation with other compatible ORBs based on the most popular product- and vendor-neutral transport layer. It can also be used as the protocol between half-bridges (see below).

The protocol is designed to be suitable and appropriate for use by any ORB to interoperate in Internet Protocol domains unless an alternative protocol is necessitated by the specific design center or intended operating environment of the ORB. In that sense it represents the basic inter-ORB protocol for TCP/IP environments, a most pervasive transport layer.

The IIOP's relationship to the GIOP is similar to that of a specific language mapping to OMG IDL; the GIOP may be mapped onto a number of different transports, and specifies the protocol elements that are common to all such mappings. The GIOP by itself, however, does not provide complete interoperability, just as IDL cannot be used to build complete programs. The IIOP and other similar mappings to different transports, are concrete realizations of the abstract GIOP definitions, as shown in Figure 12-1 on page 12-4.

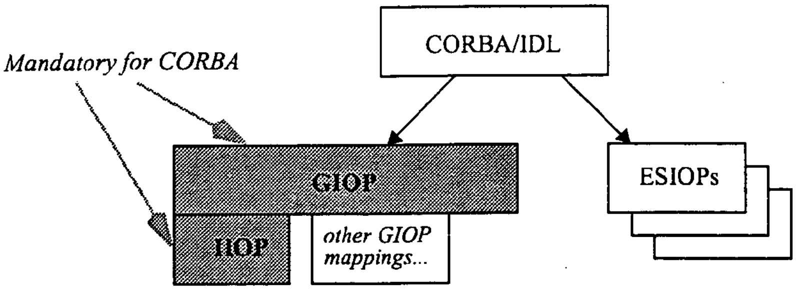


Figure 12-1 Inter-ORB Protocol Relationships.

12.1.5 Environment-Specific Inter-ORB Protocols (ESIOPs)

This specification also makes provision for an open-ended set of Environment-Specific Inter-ORB Protocols (ESIOPs). Such protocols would be used for "out of the box" interoperation at user sites where a particular networking or distributing computing infrastructure is already in general use.

Because of the opportunity to leverage and build on facilities provided by the specific environment, ESIOPs might support specialized capabilities such as those relating to security and administration.

While ESIOPs may be optimized for particular environments, all ESIOP specifications will be expected to conform to the general ORB interoperability architecture conventions to enable easy bridging. The inter-ORB bridge support enables bridges to be built between ORB domains that use the IIOP and ORB domains that use a particular ESIOP.

12.2 Relationship to Previous Versions of CORBA

The ORB Interoperability Architecture builds on Common Object Request Broker Architecture by adding the notion of ORB Services and their domains. (ORB Services are described in Section 13.2, "ORBs and ORB Services," on page 13-3). The architecture defines the problem of ORB interoperability in terms of bridging between those domains, and defines several ways in which those bridges can be constructed. The bridges can be internal (in-line) and external (request-level) to ORBs.

APIs included in the interoperability specifications include compatible extensions to previous versions of *CORBA* to support request-level bridging:

- A Dynamic Skeleton Interface (DSI) is the basic support needed for building request-level bridges. It is the server-side analogue of the Dynamic Invocation Interface and in the same way it has general applicability beyond bridging. For information about the Dynamic Skeleton Interface, refer to the Dynamic Skeleton Interface chapter.

- APIs for managing object references have been defined, building on the support identified for the Relationship Service. The APIs are defined in Object Reference Operations in the ORB Interface chapter of this book. The Relationship Service is described in the Relationship Service specification; refer to the *CosObjectIdentity Module* section of that specification.

12.3 Examples of Interoperability Solutions

The elements of interoperability (Inter-ORB Bridges, General and Internet Inter-ORB Protocols, Environment-Specific Inter-ORB Protocols) can be combined in a variety of ways to satisfy particular product and customer needs. This section provides some examples.

12.3.1 Example 1

ORB product A is designed to support objects distributed across a network and provide "out of the box" interoperability with compatible ORBs from other vendors. In addition it allows bridges to be built between it and other ORBs that use environment-specific or proprietary protocols. To accomplish this, ORB A uses the IIOP and provides inter-ORB bridge support.

12.3.2 Example 2

ORB product B is designed to provide highly optimized, very high-speed support for objects located on a single machine. For example, to support thousands of Fresco GUI objects operated on at near function-call speeds. In addition, some of the objects will need to be accessible from other machines and objects on other machines will need to be infrequently accessed. To accomplish this, ORB A provides a half-bridge to support the Internet IOP for communication with other "distributed" ORBs.

12.3.3 Example 3

ORB product C is optimized to work in a particular operating environment. It uses a particular environment-specific protocol based on distributed computing services that are commonly available at the target customer sites. In addition, ORB C is expected to interoperate with other arbitrary ORBs from other vendors. To accomplish this, ORB C provides inter-ORB bridge support and a companion half-bridge product (supplied by the ORB vendor or some third-party) provides the connection to other ORBs. The half-bridge uses the IIOP to enable interoperability with other compatible ORBs.

12.3.4 Interoperability Compliance

An ORB is considered to be interoperability-compliant when it meets the following requirements:

- In the CORBA Core part of this specification, standard APIs are provided by an ORB to enable the construction of request-level inter-ORB bridges. APIs are defined by the Dynamic Invocation Interface, the Dynamic Skeleton Interface, and by the object identity operations described in the Interface Repository chapter of this book.
- An Internet Inter-ORB Protocol (IIOP) (explained in the Building Inter-ORB Brdiges chapter) defines a transfer syntax and message formats (described independently as the General Inter-ORB Protocol), and defines how to transfer messages via TCP/IP connections. The IIOP can be supported natively or via a half-bridge.

Support for additional ESIOPs and other proprietary protocols is optional in an interoperability-compliant system. However, any implementation that chooses to use the other protocols defined by the CORBA interoperability specifications must adhere to those specifications to be compliant with CORBA interoperability.

Figure 12-2 on page 12-7 shows examples of interoperable ORB domains that are CORBA-compliant.

These compliance points support a range of interoperability solutions. For example, the standard APIs may be used to construct "half bridges" to the IIOP, relying on another "half bridge" to connect to another ORB. The standard APIs also support construction of "full bridges," without using the Internet IOP to mediate between separated bridge components. ORBs may also use the Internet IOP internally. In addition, ORBs may use GIOP messages to communicate over other network protocol families (such as Novell or OSI), and provide transport-level bridges to the IIOP.

The GIOP is described separately from the IIOP to allow future specifications to treat it as an independent compliance point.

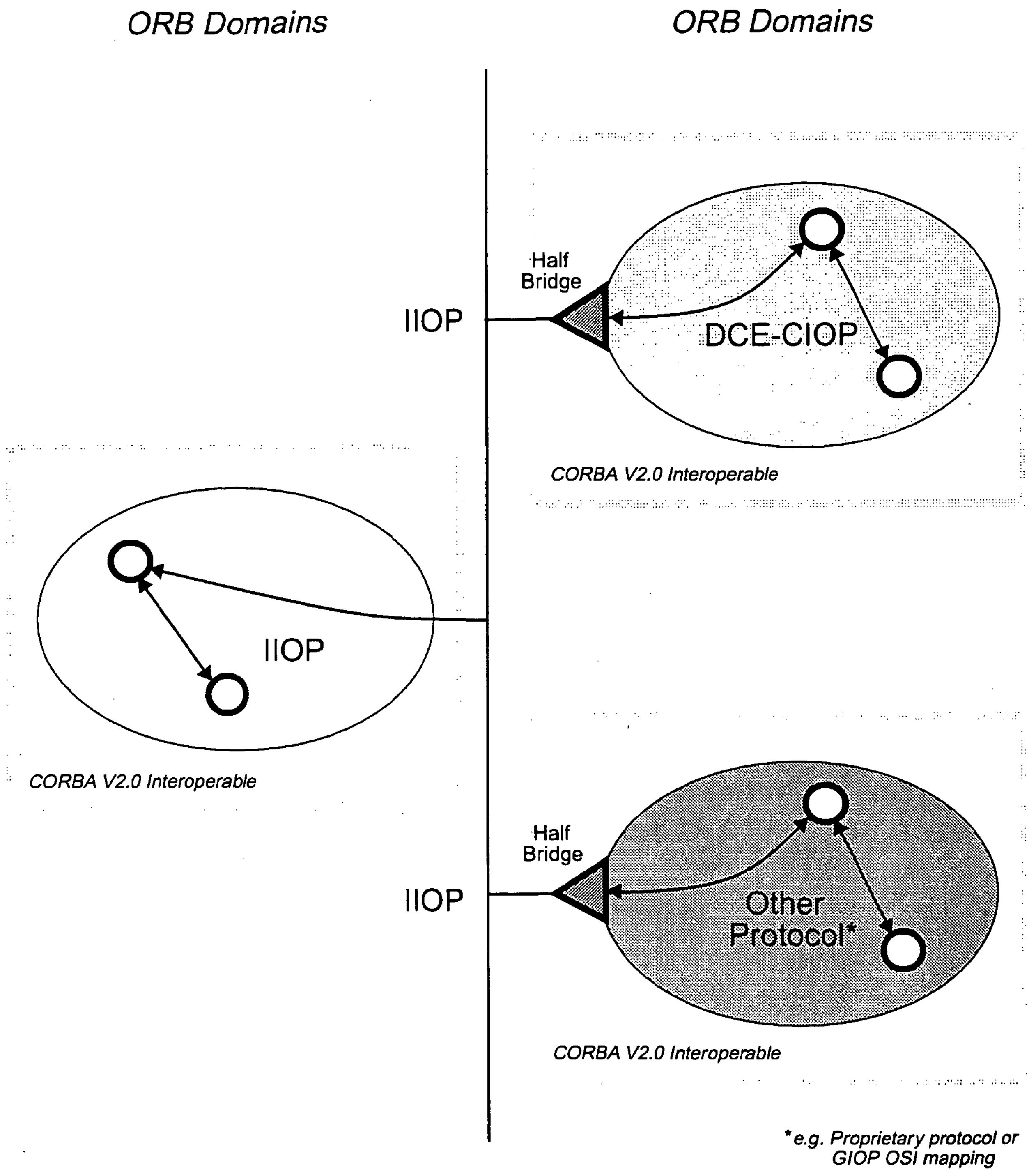

ORB Domains
ORB Domains
Half Bridge
IIOP
DCE-CIOP
CORBA V2.0 Interoperable
IIOP
CORBA V2.0 Interoperable
Half Bridge
IIOP
Other Protocol*
CORBA V2.0 Interoperable
*e.g. Proprietary protocol or GIOP OSI mapping

Figure 12-2 Examples of CORBA Interoperability Compliance

12.4 Motivating Factors

This section explains the factors that motivated the creation of interoperability specifications.

12.4.1 ORB Implementation Diversity

Today, there are many different ORB products that address a variety of user needs. A large diversity of implementation techniques is evident. For example, the time for a request ranges over at least 5 orders of magnitude, from a few microseconds to several seconds. The scope ranges from a single application to enterprise networks. Some ORBs have high levels of security, others are more open. Some ORBs are layered on a particular widely used protocol, others use highly optimized, proprietary protocols.

The market for object systems and applications that use them will grow as object systems are able to be applied to more kinds of computing. From application integration to process control, from loosely coupled operating systems to the information superhighway, CORBA-based object systems can be the common infrastructure.

12.4.2 ORB Boundaries

Even when it is not required by implementation differences, there are other reasons to partition an environment into different ORBs.

For security reasons, it may be important to know that it is not generally possible to access objects in one domain from another. For example, an "internet ORB" may make public information widely available, but a "company ORB" will want to restrict what information can get out. Even if they used the same ORB implementation, these two ORBs would be separate, so that the company could allow access to public objects from inside the company without allowing access to private objects from outside. Even though individual objects should protect themselves, prudent system administrators will want to avoid exposing sensitive objects to attacks from outside the company.

Supporting multiple ORBs also helps handle the difficult problem of testing and upgrading the object system. It would be unwise to test new infrastructure without limiting the set of objects that might be damaged by bugs, and it may be impractical to replace "the ORB" everywhere simultaneously. A new ORB might be tested and deployed in the same environment, interoperating with the existing ORB until either a complete switch is made or it incrementally displaces the existing one.

Management issues may also motivate partitioning an ORB. Just as networks are subdivided into domains to allow decentralized control of databases, configurations, resources, management of the state in an ORB (object reference location and translation information, interface repositories, per-object data) might also be done by creating sub-ORBs.

12.4.3 ORBs Vary in Scope, Distance, and Lifetime

Even in a single computing environment produced by a single vendor, there are reasons why some of the objects an application might use would be in one ORB, and others in another ORB. Some objects and services are accessed over long distances, with more global visibility, longer delays, and less reliable communication. Other objects are nearby, are not accessed from elsewhere, and provide higher quality service. By deciding which ORB to use, an implementer sets expectations for the clients of the objects.

One ORB might be used to retain links to information that is expected to accumulate over decades, such as library archives. Another ORB might be used to manage a distributed chess program in which the objects should all be destroyed when the game is over. Although while it is running, it makes sense for "chess ORB" objects to access the "archives ORB," we would not expect the archives to try to keep a reference to the current board position.

12.5 Interoperability Design Goals

Because of the diversity in ORB implementations, multiple approaches to interoperability are required. Options identified in previous versions of *CORBA* include:

- *Protocol Translation*, where a gateway residing somewhere in the system maps requests from the format used by one ORB to that used by another.
- *Reference Embedding*, where invocation using a native object reference delegates to a special object whose job is to forward that invocation to another ORB.
- *Alternative ORBs*, where ORB implementations agree to coexist in the same address space so easily that a client or implementation can transparently use any of them, and pass object references created by one ORB to another ORB without losing functionality.

In general, there is no single protocol that can meet everyone's needs, and there is no single means to interoperate between two different protocols. There are many environments in which multiple protocols exist, and there are ways to bridge between environments that share no protocols.

This specification adopts a flexible architecture that allows a wide variety of ORB implementations to interoperate and that includes both bridging and common protocol elements.

The following goals guided the creation of interoperability specifications:

- The architecture and specifications should allow high-performance, small footprint, lightweight interoperability solutions.
- The design should scale, should not be unduly difficult to implement, and should not unnecessarily restrict implementation choices.

- Interoperability solutions should be able to work with any vendors' existing ORB implementations with respect to their CORBA-compliant core feature set; those implementations are diverse.
- All operations implied by the CORBA object model (i.e., the stringify and destringify operations defined on the **CORBA:ORB** pseudo-object and all the operations on **CORBA:Object**) as well as type management (e.g., narrowing, as needed by the C++ mapping) should be supported.

12.5.1 Non-Goals

The following were taken into account, but were not goals:

- Support for security
- Support for future ORB Services

ORB Interoperability Architecture 13

Contents

This chapter contains the following sections.

Section Title	Page
"Overview"	13-1
"ORBs and ORB Services"	13-3
"Domains"	13-5
"Interoperability Between ORBs"	13-7
"Object Addressing"	13-11
"An Information Model for Object References"	13-14
"Service Context"	13-28
"Coder/Decoder Interfaces"	13-31
"Feature Support and GIOP Versions"	13-35
"Code Set Conversion"	13-36

13.1 Overview

The original Interoperability RFP defines interoperability as the ability for a client on ORB A to invoke an OMG IDL-defined operation on an object on ORB B, where ORB A and ORB B are independently developed. It further identifies general requirements including in particular:

- Ability for two vendors' ORBs to interoperate without prior knowledge of each other's implementation.

- Support of all ORB functionality.
- Preservation of content and semantics of ORB-specific information across ORB boundaries (for example, security).

In effect, the requirement is for invocations between client and server objects to be independent of whether they are on the same or different ORBs, and not to mandate fundamental modifications to existing ORB products.

13.1.1 Domains

The CORBA Object Model identifies various distribution transparencies that must be supported within a single ORB environment, such as location transparency. Elements of ORB functionality often correspond directly to such transparencies. Interoperability can be viewed as extending transparencies to span multiple ORBs.

In this architecture a *domain* is a distinct scope, within which certain common characteristics are exhibited and common rules are observed over which a distribution transparency is preserved. Thus, interoperability is fundamentally involved with transparently crossing such domain boundaries.

Domains tend to be either administrative or technological in nature, and need not correspond to the boundaries of an ORB installation. Administrative domains include naming domains, trust groups, resource management domains and other "run-time" characteristics of a system. Technology domains identify common protocols, syntaxes and similar "build-time" characteristics. In many cases, the need for technology domains derives from basic requirements of administrative domains.

Within a single ORB, most domains are likely to have similar scope to that of the ORB itself: common object references, network addresses, security mechanisms, and more. However, it is possible for there to be multiple domains of the same type supported by a given ORB: internal representation on different machine types, or security domains. Conversely, a domain may span several ORBs: similar network addresses may be used by different ORBs, type identifiers may be shared.

13.1.2 Bridging Domains

The abstract architecture describes ORB interoperability in terms of the translation required when an object request traverses domain boundaries. Conceptually, a mapping or *bridging mechanism* resides at the boundary between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain.

The concrete architecture identifies two approaches to inter-ORB bridging:

- At application level, allowing flexibility and portability.
- At ORB level, built into the ORB itself.

13.2 ORBs and ORB Services

The ORB Core is that part of the ORB which provides the basic representation of objects and the communication of requests. The ORB Core therefore supports the minimum functionality to enable a client to invoke an operation on a server object, with (some of) the distribution transparencies required by *CORBA*.

An object request may have implicit attributes which affect the way in which it is communicated - though not the way in which a client makes the request. These attributes include security, transactional capabilities, recovery, and replication. These features are provided by "ORB Services," which will in some ORBs be layered as internal services over the core, or in other cases be incorporated directly into an ORB's core. It is an aim of this specification to allow for new ORB Services to be defined in the future, without the need to modify or enhance this architecture.

Within a single ORB, ORB services required to communicate a request will be implemented and (implicitly) invoked in a private manner. For interoperability between ORBs, the ORB services used in the ORBs, and the correspondence between them, must be identified.

13.2.1 The Nature of ORB Services

ORB Services are invoked implicitly in the course of application-level interactions. ORB Services range from fundamental mechanisms such as reference resolution and message encoding to advanced features such as support for security, transactions, or replication.

An ORB Service is often related to a particular transparency. For example, message encoding – the marshaling and unmarshaling of the components of a request into and out of message buffers – provides transparency of the representation of the request. Similarly, reference resolution supports location transparency. Some transparencies, such as security, are supported by a combination of ORB Services and Object Services while others, such as replication, may involve interactions between ORB Services themselves.

ORB Services differ from Object Services in that they are positioned below the application and are invoked transparently to the application code. However, many ORB Services include components which correspond to conventional Object Services in that they are invoked explicitly by the application.

Security is an example of service with both ORB Service and normal Object Service components, the ORB components being those associated with transparently authenticating messages and controlling access to objects while the necessary administration and management functions resemble conventional Object Services.

13.2.2 ORB Services and Object Requests

Interoperability between ORBs extends the scope of distribution transparencies and other request attributes to span multiple ORBs. This requires the establishment of relationships between supporting ORB Services in the different ORBs.

In order to discuss how the relationships between ORB Services are established, it is necessary to describe an abstract view of how an operation invocation is communicated from client to server object.

1. The client generates an operation request, using a reference to the server object, explicit parameters, and an implicit invocation context. This is processed by certain ORB Services on the client path.
2. On the server side, corresponding ORB Services process the incoming request, transforming it into a form directly suitable for invoking the operation on the server object.
3. The server object performs the requested operation.
4. Any result of the operation is returned to the client in a similar manner.

The correspondence between client-side and server-side ORB Services need not be one-to-one and in some circumstances may be far more complex. For example, if a client application requests an operation on a replicated server, there may be multiple server-side ORB service instances, possibly interacting with each other.

In other cases, such as security, client-side or server-side ORB Services may interact with Object Services such as authentication servers.

13.2.3 Selection of ORB Services

The ORB Services used are determined by:

- Static properties of both client and server objects; for example, whether a server is replicated.
- Dynamic attributes determined by a particular invocation context; for example, whether a request is transactional.
- Administrative policies (e.g., security).

Within a single ORB, private mechanisms (and optimizations) can be used to establish which ORB Services are required and how they are provided. Service selection might in general require negotiation to select protocols or protocol options. The same is true between different ORBs: it is necessary to agree which ORB Services are used, and how each transforms the request. Ultimately, these choices become manifest as one or more protocols between the ORBs or as transformations of requests.

In principle, agreement on the use of each ORB Service can be independent of the others and, in appropriately constructed ORBs, services could be layered in any order or in any grouping. This potentially allows applications to specify selective transparencies according to their requirements, although at this time CORBA provides no way to penetrate its transparencies.

A client ORB must be able to determine which ORB Services must be used in order to invoke operations on a server object. Correspondingly, where a client requires dynamic attributes to be associated with specific invocations, or administrative policies dictate, it must be possible to cause the appropriate ORB Services to be used on client and server sides of the invocation path. Where this is not possible - because, for example, one ORB does not support the full set of services required - either the interaction cannot proceed or it can only do so with reduced facilities or transparencies.

13.3 Domains

From a computational viewpoint, the OMG Object Model identifies various distribution transparencies which ensure that client and server objects are presented with a uniform view of a heterogeneous distributed system. From an engineering viewpoint, however, the system is not wholly uniform. There may be distinctions of location and possibly many others such as processor architecture, networking mechanisms and data representations. Even when a single ORB implementation is used throughout the system, local instances may represent distinct, possibly optimized scopes for some aspects of ORB functionality.

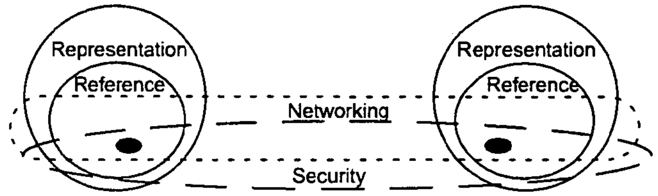


Figure 13-1 Different Kinds of Domains can Coexist.

Interoperability, by definition, introduces further distinctions, notably between the scopes associated with each ORB. To describe both the requirements for interoperability and some of the solutions, this architecture introduces the concept of *domains* to describe the scopes and their implications.

Informally, a domain is a set of objects sharing a common characteristic or abiding by common rules. It is a powerful modelling concept which can simplify the analysis and description of complex systems. There may be many types of domains (e.g., management domains, naming domains, language domains, and technology domains).

13.3.1 Definition of a Domain

Domains allow partitioning of systems into collections of components which have some characteristic in common. In this architecture a domain is a scope in which a collection of objects, said to be members of the domain, is associated with some common characteristic; any object for which the association does not exist, or is undefined, is not a member of the domain. A domain can be modeled as an object and may be itself a member of other domains.

It is the scopes themselves and the object associations or bindings defined within them which characterize a domain. This information is disjoint between domains. However, an object may be a member of several domains, of similar kinds as well as of different kinds, and so the sets of members of domains may overlap.

The concept of a domain boundary is defined as the limit of the scope in which a particular characteristic is valid or meaningful. When a characteristic in one domain is translated to an equivalent in another domain, it is convenient to consider it as traversing the boundary between the two domains.

Domains are generally either administrative or technological in nature. Examples of domains related to ORB interoperability issues are:

- Referencing domain – the scope of an object reference
- Representation domain – the scope of a message transfer syntax and protocol
- Network addressing domain – the scope of a network address
- Network connectivity domain – the potential scope of a network message
- Security domain – the extent of a particular security policy
- Type domain – the scope of a particular type identifier
- Transaction domain – the scope of a given transaction service

Domains can be related in two ways: containment, where a domain is contained within another domain, and federation, where two domains are joined in a manner agreed to and set up by their administrators.

13.3.2 Mapping Between Domains: Bridging

Interoperability between domains is only possible if there is a well-defined mapping between the behaviors of the domains being joined. Conceptually, a mapping mechanism or bridge resides at the boundary between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain. Note that the use of the term "bridge" in this context is conceptual and refers only to the functionality which performs the required mappings between distinct domains. There are several implementation options for such bridges and these are discussed elsewhere.

For full interoperability, it is essential that all the concepts used in one domain are transformable into concepts in other domains with which interoperability is required, or that if the bridge mechanism filters such a concept out, nothing is lost as far as the supported objects are concerned. In other words, one domain may support a superior service to others, but such a superior functionality will not be available to an application system spanning those domains.

A special case of this requirement is that the object models of the two domains need to be compatible. This specification assumes that both domains are strictly compliant with the CORBA Object Model and the *CORBA* specifications. This includes the use of OMG IDL when defining interfaces, the use of the CORBA Core Interface Repository, and other modifications that were made to *CORBA*. Variances from this model could easily compromise some aspects of interoperability.

13.4 Interoperability Between ORBs

An ORB "provides the mechanisms by which objects transparently make and receive requests and responses. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments..." ORB interoperability extends this definition to cases in which client and server objects on different ORBs "transparently make and receive requests."

Note that a direct consequence of this transparency requirement is that bridging must be bidirectional: that is, it must work as effectively for object references passed as parameters as for the target of an object invocation. Were bridging unidirectional (e.g., if one ORB could only be a client to another) then transparency would not have been provided, because object references passed as parameters would not work correctly: ones passed as "callback objects," for example, could not be used.

Without loss of generality, most of this specification focuses on bridging in only one direction. This is purely to simplify discussions, and does not imply that unidirectional connectivity satisfies basic interoperability requirements.

13.4.1 ORB Services and Domains

In this architecture, different aspects of ORB functionality - ORB Services - can be considered independently and associated with different domain types. The architecture does not, however, prescribe any particular decomposition of ORB functionality and interoperability into ORB Services and corresponding domain types. There is a range of possibilities for such a decomposition:

1. The simplest model, for interoperability, is to treat all objects supported by one ORB (or, alternatively, all ORBs of a given type) as comprising one domain. Interoperability between any pair of different domains (or domain types) is then achieved by a specific all-encompassing bridge between the domains. (This is all *CORBA* implies.)
2. More detailed decompositions would identify particular domain types - such as referencing, representation, security, and networking. A core set of domain types would be pre-determined and allowance made for additional domain types to be defined as future requirements dictate (e.g., for new ORB Services).

13.4.2 ORBs and Domains

In many respects, issues of interoperability between ORBs are similar to those which can arise with a single type of ORB (e.g., a product). For example:

- Two installations of the ORB may be installed in different security domains, with different Principal identifiers. Requests crossing those security domain boundaries will need to establish locally meaningful Principals for the caller identity, and for any Principals passed as parameters.
- Different installations might assign different type identifiers for equivalent types, and so requests crossing type domain boundaries would need to establish locally meaningful type identifiers (and perhaps more).

Conversely, not all of these problems need to appear when connecting two ORBs of a different type (e.g., two different products). Examples include:

- They could be administered to share user visible naming domains, so that naming domains do not need bridging.
- They might reuse the same networking infrastructure, so that messages could be sent without needing to bridge different connectivity domains.

Additional problems can arise with ORBs of different types. In particular, they may support different concepts or models, between which there are no direct or natural mappings. CORBA only specifies the application level view of object interactions, and requires that distribution transparencies conceal a whole range of lower level issues. It follows that within any particular ORB, the mechanisms for supporting transparencies are not visible at the application-level and are entirely a matter of implementation choice. So there is no guarantee that any two ORBs support similar internal models or that there is necessarily a straightforward mapping between those models.

These observations suggest that the concept of an ORB (instance) is too coarse or superficial to allow detailed analysis of interoperability issues between ORBs. Indeed, it becomes clear that an ORB instance is an elusive notion: it can perhaps best be characterized as the intersection or coincidence of ORB Service domains.

13.4.3 Interoperability Approaches

When an interaction takes place across a domain boundary, a mapping mechanism, or bridge, is required to transform relevant elements of the interaction as they traverse the boundary. There are essentially two approaches to achieving this: mediated bridging and immediate bridging. These approaches are described in the following subsections.

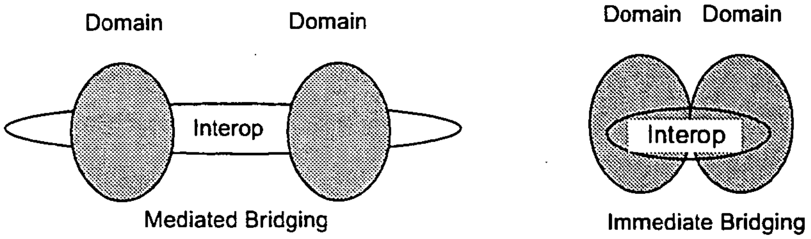


Figure 13-2 Two bridging techniques, different uses of an intermediate form agreed on between the two domains.

13.4.3.1 Mediated Bridging

With mediated bridging, elements of the interaction relevant to the domain are transformed, at the boundary of each domain, between the internal form of that domain and an agreed, common form.

Observations on mediated bridging are as follows:

- The scope of agreement of a common form can range from a private agreement between two particular ORB/domain implementations to a universal standard.

- There can be more than one common form, each oriented or optimized for a different purpose.
- If there is more than one possible common form, then which is used can be static (e.g., administrative policy agreed between ORB vendors, or between system administrators) or dynamic (e.g., established separately for each object, or on each invocation).
- Engineering of this approach can range from in-line specifically compiled (compare to stubs) or generic library code (such as encryption routines), to intermediate bridges to the common form.

13.4.3.2 Immediate Bridging

With immediate bridging, elements of the interaction relevant to the domain are transformed, at the boundary of each domain, directly between the internal form of one domain and the internal form of the other.

Observations on immediate bridging are as follows:

- This approach has the potential to be optimal (in that the interaction is not mediated via a third party, and can be specifically engineered for each pair of domains) but sacrifices flexibility and generality of interoperability to achieve this.
- This approach is often applicable when crossing domain boundaries which are purely administrative (i.e., there is no change of technology). For example, when crossing security administration domains between similar ORBs, it is not necessary to use a common intermediate standard.

As a general observation, the two approaches can become almost indistinguishable when private mechanisms are used between ORB/domain implementations.

13.4.3.3 Location of Inter-Domain Functionality

Logically, an inter-domain bridge has components in both domains, whether the mediated or immediate bridging approach is used. However, domains can span ORB boundaries and ORBs can span machine and system boundaries; conversely, a machine may support, or a process may have access to more than one ORB (or domain of a given type). From an engineering viewpoint, this means that the components of an inter-domain bridge may be dispersed or co-located, with respect to ORBs or systems. It also means that the distinction between an ORB and a bridge can be a matter of perspective: there is a duality between viewing inter-system messaging as belonging to ORBs, or to bridges.

For example, if a single ORB encompasses two security domains, the inter-domain bridge could be implemented wholly within the ORB and thus be invisible as far as ORB interoperability is concerned. A similar situation arises when a bridge between two ORBs or domains is implemented wholly within a process or system which has access to both. In such cases, the engineering issues of inter-domain bridging are confined, possibly to a single system or process. If it were practical to implement all bridging in this way, then interactions between systems or processes would be solely within a single domain or ORB.

13.4.3.4 Bridging Level

As noted at the start of this section, bridges may be implemented both internally to an ORB and as layers above it. These are called respectively "in-line" and "request-level" bridges.

Request-level bridges use the CORBA APIs, including the Dynamic Skeleton Interface, to receive and issue requests. However, there is an emerging class of "implicit context" which may be associated with some invocations, holding ORB Service information such as transaction and security context information, which is not at this time exposed through general purpose public APIs. (Those APIs expose only OMG IDL-defined operation parameters, not implicit ones.) Rather, the precedent set with the Transaction Service is that special purpose APIs are defined to allow bridging of each kind of context. This means that request-level bridges must be built to specifically understand the implications of bridging such ORB Service domains, and to make the appropriate API calls.

13.4.4 Policy-Mediated Bridging

An assumption made through most of this specification is that the existence of domain boundaries should be transparent to requests: that the goal of interoperability is to hide such boundaries. However, if this were always the goal, then there would be no real need for those boundaries in the first place.

Realistically, administrative domain boundaries exist because they reflect ongoing differences in organizational policies or goals. Bridging the domains will in such cases require *policy mediation*. That is, inter-domain traffic will need to be constrained, controlled, or monitored; fully transparent bridging may be highly undesirable. Resource management policies may even need to be applied, restricting some kinds of traffic during certain periods.

Security policies are a particularly rich source of examples: a domain may need to audit external access, or to provide domain-based access control. Only a very few objects, types of objects, or classifications of data might be externally accessible through a "firewall."

Such policy-mediated bridging requires a bridge that knows something about the traffic being bridged. It could in general be an application-specific policy, and many policy-mediated bridges could be parts of applications. Those might be organization-specific, off-the-shelf, or anywhere in between.

Request-level bridges, which use only public ORB APIs, easily support the addition of policy mediation components, without loss of access to any other system infrastructure that may be needed to identify or enforce the appropriate policies.

13.4.5 Configurations of Bridges in Networks

In the case of network-aware ORBs, we anticipate that some ORB protocols will be more frequently bridged to than others, and so will begin to serve the role of "backbone ORBs." (This is a role that the IIOP is specifically expected to serve.) This use of "backbone topology" is true both on a large scale and a small scale. While a large scale public data network provider could define its own backbone ORB, on a smaller scale, any given institution will probably designate one commercially available ORB as its backbone.

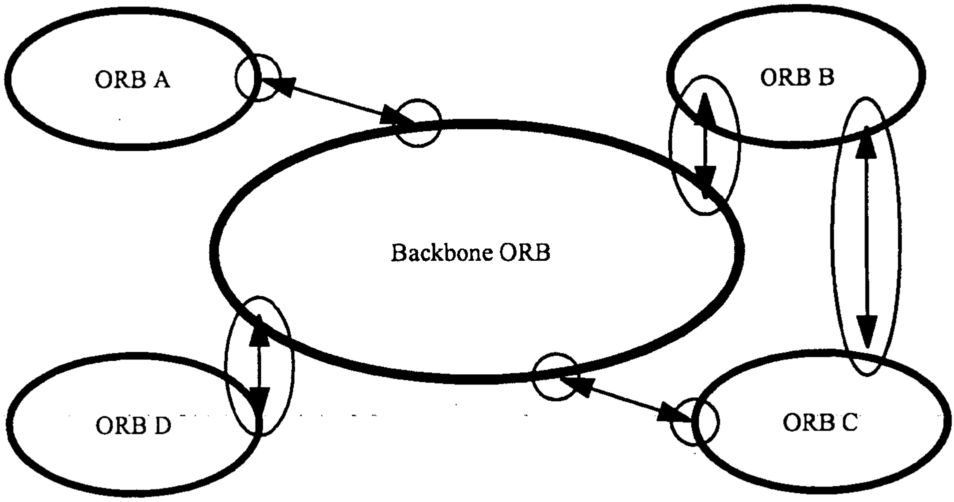


Figure 13-3 An ORB chosen as a backbone will connect other ORBs through bridges, both full-bridges and half-bridges.

Adopting a backbone style architecture is a standard administrative technique for managing networks. It has the consequence of minimizing the number of bridges needed, while at the same time making the ORB topology match typical network organizations. (That is, it allows the number of bridges to be proportional to the number of protocols, rather than combinatorial.)

In large configurations, it will be common to notice that adding ORB bridges doesn't even add any new "hops" to network routes, because the bridges naturally fit in locations where connectivity was already indirect, and augment or supplant the existing network firewalls.

13.5 Object Addressing

The Object Model (see Chapter 1, Requests) defines an object reference as an object name that reliably denotes a particular object. An object reference identifies the same object each time the reference is used in a request, and an object may be denoted by multiple, distinct references.

The fundamental ORB interoperability requirement is to allow clients to use such object names to invoke operations on objects in other ORBs. Clients do not need to distinguish between references to objects in a local ORB or in a remote one. Providing this transparency can be quite involved, and naming models are fundamental to it.

This section discusses models for naming entities in multiple domains, and transformations of such names as they cross the domain boundaries. That is, it presents transformations of object reference information as it passes through networks of inter-ORB bridges. It uses the word "ORB" as synonymous with referencing domain; this is purely to simplify the discussion. In other contexts, "ORB" can usefully denote other kinds of domain.

13.5.1 Domain-relative Object Referencing

Since CORBA does not require ORBs to understand object references from other ORBs, when discussing object references from multiple ORBs one must always associate the object reference's domain (ORB) with the object reference. We use the notation *D0.R0* to denote an object reference *R0* from domain *D0*; this is itself an object reference. This is called "domain-relative" referencing (or addressing) and need not reflect the implementation of object references within any ORB.

At an implementation level, associating an object reference with an ORB is only important at an inter-ORB boundary; that is, inside a bridge. This is simple, since the bridge knows from which ORB each request (or response) came, including any object references embedded in it.

13.5.2 Handling of Referencing Between Domains

When a bridge hands an object reference to an ORB, it must do so in a form understood by that ORB: the object reference must be in the recipient ORB's native format. Also, in cases where that object originated from some other ORB, the bridge must associate each newly created "proxy" object reference with (what it sees as) the original object reference.

Several basic schemes to solve these two problems exist. These all have advantages in some circumstances; all can be used, and in arbitrary combination with each other, since CORBA object references are opaque to applications. The ramifications of each scheme merits attention, with respect to scaling and administration. The schemes include:

1. *Object Reference Translation Reference Embedding*: The bridge can store the original object reference itself, and pass an entirely different proxy reference into the new domain. The bridge must then manage state on behalf of each bridged object reference, map these references from one ORB's format to the other's, and vice versa.

2. *Reference Encapsulation:* The bridge can avoid holding any state at all by conceptually concatenating a domain identifier to the object name. Thus if a reference *D0.R*, originating in domain *D0*, traversed domains *D1... D4* it could be identified in *D4* as proxy reference *d3.d2.d1.d0.R*, where *dn* is the address of *Dn* relative to *Dn+1*.

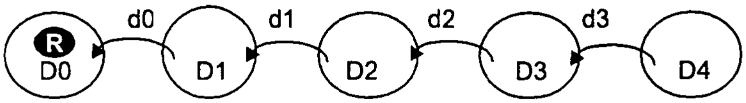


Figure 13-4 Reference encapsulation adds domain information during bridging.

3. *Domain Reference Translation:* Like object reference translation, this scheme holds some state in the bridge. However, it supports sharing that state between multiple object references by adding a domain-based route identifier to the proxy (which still holds the original reference, as in the reference encapsulation scheme). It achieves this by providing encoded domain route information each time a domain boundary is traversed; thus if a reference *D0.R*, originating in domain *D0*, traversed domains *D1...D4* it would be identified in *D4* as *(d3, x3).R*, and in *D2* as *(d1,x1).R*, and so on, where *dn* is the address of *Dn* relative to *Dn+1*, and *xn* identifies the pair *(dn-1, xn-1)*.

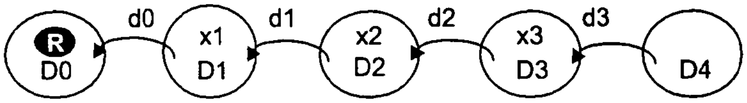


Figure 13-5 Domain Reference Translation *substitutes domain references* during bridging.

4. *Reference Canonicalization:* This scheme is like domain reference translation, except that the proxy uses a "well-known" (e.g., global) domain identifier rather than an encoded path. Thus a reference *R*, originating in domain *D0* would be identified in other domains as *D0.R*.

Observations about these approaches to inter-domain reference handling are as follows:

- Naive application of reference encapsulation could lead to arbitrarily large references. A "topology service" could optimize cycles within any given encapsulated reference and eliminate the appearance of references to local objects as alien references.
- A topology service could also optimize the chains of routes used in the domain reference translation scheme. Since the links in such chains are re-used by any path traversing the same sequence of domains, such optimization has particularly high leverage.

- With the general purpose APIs defined in *CORBA*, object reference translation can be supported even by ORBs not specifically intended to support efficient bridging, but this approach involves the most state in intermediate bridges. As with reference encapsulation, a topology service could optimize individual object references. (APIs are defined by the Dynamic Skeleton Interface and Dynamic Invocation Interface)
- The chain of addressing links established with both object and domain reference translation schemes must be represented as state within the network of bridges. There are issues associated with managing this state.
- Reference canonicalization can also be performed with managed hierarchical name spaces such as those now in use on the Internet and X.500 naming.

13.6 An Information Model for Object References

This section provides a simple, powerful information model for the information found in an object reference. That model is intended to be used directly by developers of bridging technology, and is used in that role by the IIOP, described in the *General Inter-ORB Protocol* chapter, *Object References* section.

13.6.1 What Information Do Bridges Need?

The following potential information about object references has been identified as critical for use in bridging technologies:

- *Is it null?* Nulls only need to be transmitted and never support operation invocation.
- *What type is it?* Many ORBs require knowledge of an object's type in order to efficiently preserve the integrity of their type systems.
- *What protocols are supported?* Some ORBs support objrefs that in effect live in multiple referencing domains, to allow clients the choice of the most efficient communications facilities available.
- *What ORB Services are available?* As noted in Section 13.2.3, "Selection of ORB Services" on page 13-4, several different ORB Services might be involved in an invocation. Providing information about those services in a standardized way could in many cases reduce or eliminate negotiation overhead in selecting them.

13.6.2 Interoperable Object References: IORs

To provide the information above, an "Interoperable Object Reference," (IOR) data structure has been provided. This data structure need not be used internally to any given ORB, and is not intended to be visible to application-level ORB programmers. It should be used only when crossing object reference domain boundaries, within bridges.

This data structure is designed to be efficient in typical single-protocol configurations, while not penalizing multiprotocol ones.

```
module IOP {                                               // IDL

    // Standard Protocol Profile tag values

    typedef unsigned long             ProfileId;

    struct TaggedProfile {
        ProfileId                     tag;
        sequence <octet>              profile_data;
    };

    // an Interoperable Object Reference is a sequence of
    // object-specific protocol profiles, plus a type ID.

    struct IOR {
        string                        type_id;
        sequence <TaggedProfile>      profiles;
    };

    // Standard way of representing multicomponent profiles.
    // This would be encapsulated in a TaggedProfile.

    typedef unsigned long ComponentId;
    struct TaggedComponent {
        ComponentId                   tag;
        sequence <octet>              component_data;
    };
    typedef sequence<TaggedComponent> TaggedComponentSeq;
};
```

13.6.3 IOR Profiles

Object references have at least one *tagged profile*. Each profile supports one or more protocols and encapsulates all the basic information the protocols it supports need to identify an object. Any single profile holds enough information to drive a complete invocation using any of the protocols it supports; the content and structure of those profile entries are wholly specified by these protocols.

When a specific protocol is used to convey an object reference passed as a parameter in an IDL operation invocation (or reply), an IOR which reflects, in its contained profiles, the full protocol understanding of the operation client (or server in case of reply) may be sent. A receiving ORB which operates (based on topology and policy information available to it) on profiles rather than the received IOR as a whole, to create a derived reference for use in its own domain of reference, is placing itself as a bridge between reference domains. Interoperability inhibiting situations can arise when an orb sends an IOR with multiple profiles (using one of its supported protocols) to a receiving orb, and that receiving orb later returns a derived reference to that object, which has had profiles or profile component data removed or transformed from the original IOR contents.

To assist in classifying behavior of ORBS in such bridging roles, two classes of IOR conformance may be associated with the conformance requirements for a given ORB interoperability protocol:

- Full IOR conformance requires that an orb which receives an IOR for an object passed to it through that ORB interoperability protocol, shall recover the original IOR, in its entirety, for passing as a reference to that object from that orb through that same protocol
- Limited-Profile IOR conformance requires that an orb which receives an IOR passed to it through a given ORB interoperability protocol, shall recover all of the standard information contained in the IOR profile for that protocol, whenever passing a reference to that object, using that same protocol, to another ORB.

Note – Conformance to IIOP versions 1.0, 1.1 and 1.2 only requires support of limited-Profile IOR conformance, specifically for the IIOP IOR profile. However, due to interoperability problems induced by Limited-Profile IOR conformance, it is now deprecated by the CORBA 2.4 specification for an orb to not support Full IOR conformance. Some future IIOP versions could require Full IOR conformance.

An ORB may be unable to use any of the profiles provided in an IOR for various reasons which may be broadly categorized as transient ones like temporary network outage, and non-transient ones like unavailability of appropriate protocol software in the ORB. The decision about the category of outage that causes an ORB to be unable to use any profile from an IOR is left up to the ORB. At an appropriate point, when an ORB discovers that it is unable to use any profile in an IOR, depending on whether it considers the reason transient or non-transient, it should raise the standard system exception TRANSIENT with standard minor code 2, or IMP_LIMIT with the standard minor code 1.

Each profile has a unique numeric tag, assigned by the OMG. The ones defined here are for the IIOP (see Section 15.7.3, "IIOP IOR Profile Components" on page 15-54) and for use in "multiple component profiles." Profile tags in the range **0x80000000** through **0xffffffff** are reserved for future use, and are not currently available for assignment.

Null object references are indicated by an empty set of profiles, and by a "Null" type ID (a string which contains only a single terminating character). Type IDs may only be "Null" in any message, requiring the client to use existing knowledge or to consult the object, to determine interface types supported. The type ID is a Repository ID identifying the interface type, and is provided to allow ORBs to preserve strong typing. This identifier is agreed on within the bridge and, for reasons outside the scope of this interoperability specification, needs to have a much broader scope to address various problems in system evolution and maintenance. Type IDs support detection of type equivalence, and in conjunction with an Interface Repository, allow processes to reason about the relationship of the type of the object referred to and any other type.

The type ID, if provided by the server, indicates the most derived type that the server wishes to publish, at the time the reference is generated. The object's actual most derived type may later change to a more derived type. Therefore, the type ID in the IOR can only be interpreted by the client as a hint that the object supports at least the indicated interface. The client can succeed in narrowing the reference to the indicated interface, or to one of its base interfaces, based solely on the type ID in the IOR, but must not fail to narrow the reference without consulting the object via the "_is_a" or "_get_interface" pseudo-operations.

ORBs claiming to support the Full-IOR conformance are required to preserve all the semantic content of any IOR (including the ordering of each profile and its components), and may only apply transformations which preserve semantics (e.g., changing Byte order for encapsulation).

For example, consider an echo operation for object references:

```
interface Echoer {Object echo(in Object o);};
```

Assume that the method body implementing this "echo" operation simply returns its argument. When a client application invokes the echo operation and passes an arbitrary object reference, if both the client and server ORBs claim support to Full IOR conformance, the reference returned by the operation is guaranteed to have not been semantically altered by either client or server ORB. That is, all its profiles will remain intact and in the same order as they were present when the reference was sent. This requirement for ORBs which claim support for Full-IOR conformance, ensures that, for example, a client can safely store an object reference in a naming service and get that reference back again later without losing information inside the reference.

13.6.4 Standard IOR Profiles

```
module IOP {
    const ProfileId         TAG_INTERNET_IOP = 0;
    const ProfileId         TAG_MULTIPLE_COMPONENTS = 1;
    const ProfileId         TAG_SCCP_IOP = 2;

    typedef sequence <TaggedComponent> MultipleComponentProfile;
};
```

13.6.4.1 The TAG_INTERNET_IOP Profile

The **TAG_INTERNET_IOP** tag identifies profiles that support the Internet Inter-ORB Protocol. The **ProfileBody** of this profile, described in detail in Section 15.7.2, "IIOP IOR Profiles" on page 15-51, contains a CDR encapsulation of a structure containing addressing and object identification information used by IIOP. Version 1.1 of the **TAG_INTERNET_IOP** profile also includes a **sequence<TaggedComponent>** that can contain additional information supporting optional IIOP features, ORB services such as security, and future protocol extensions.

Protocols other than IIOP (such as ESIOPs and other GIOPs) can share profile information (such as object identity or security information) with IIOP by encoding their additional profile information as components in the **TAG_INTERNET_IOP** profile. All **TAG_INTERNET_IOP** profiles support IIOP, regardless of whether they also support additional protocols. Interoperable ORBs are not required to create or understand any other profile, nor are they required to create or understand any of the components defined for other protocols that might share the **TAG_INTERNET_IOP** profile with IIOP.

The **profile_data** for the **TAG_INTERNET_IOP** profile is a CDR encapsulation of the **IIOP::ProfileBody_1_1** type, described in Section 15.7.2, "IIOP IOR Profiles" on page 15-51.

13.6.4.2 The TAG_MULTIPLE_COMPONENTS Profile

The **TAG_MULTIPLE_COMPONENTS** tag indicates that the value encapsulated is of type **MultipleComponentProfile**. In this case, the profile consists of a list of protocol components, the use of which must be specified by the protocol using this profile. This profile may be used to carry IOR components, as specified in Section 13.6.5, "IOR Components" on page 13-18.

The **profile_data** for the **TAG_MULTIPLE_COMPONENTS** profile is a CDR encapsulation of the **MultipleComponentProfile** type shown above.

13.6.4.3 The TAG_SCCP_IOP Profile

See the CORBA/IN Interworking specification (dtc/2000-02-02).

13.6.5 IOR Components

TaggedComponents contained in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles are identified by unique numeric tags using a namespace distinct form that is used for profile tags. Component tags are assigned by the OMG.

Specifications of components must include the following information:

- *Component ID:* The compound tag that is obtained from OMG.
- *Structure and encoding:* The syntax of the component data and the encoding rules. If the component value is encoded as a CDR encapsulation, the IDL type that is encapsulated and the GIOP version which is used for encoding the value, if different than GIOP 1.0, must be specified as part of the component definition.
- *Semantics:* How the component data is intended to be used.
- *Protocols:* The protocol for which the component is defined, and whether it is intended that the component be usable by other protocols.
- *At most once:* whether more than one instance of this component can be included in a profile.

Specifications of protocols must describe how the components affect the protocol. In addition, a protocol definition must specify, for each TaggedComponent, whether inclusion of the component in profiles supporting the protocol is required (MANDATORY PRESENCE) or not required (OPTIONAL PRESENCE). An ORB claiming to support Full-IOR conformance shall not drop optional components, once they have been added to a profile.

13.6.6 Standard IOR Components

The following are standard IOR components that can be included in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles, and may apply to IIOP, other GIOPs, ESIOPs, or other protocols. An ORB must not drop these components from an existing IOR.

```
module IOP {
    const ComponentId    TAG_ORB_TYPE = 0;
    const ComponentId    TAG_CODE_SETS = 1;
    const ComponentId    TAG_POLICIES = 2;
    const ComponentId    TAG_ALTERNATE_IIOP_ADDRESS = 3;

    const ComponentId    TAG_ASSOCIATION_OPTIONS = 13;
    const ComponentId    TAG_SEC_NAME = 14;
    const ComponentId    TAG_SPKM_1_SEC_MECH = 15;
    const ComponentId    TAG_SPKM_2_SEC_MECH = 16;
    const ComponentId    TAG_KerberosV5_SEC_MECH = 17;
    const ComponentId    TAG_CSI_ECMA_Secret_SEC_MECH = 18;
    const ComponentId    TAG_CSI_ECMA_Hybrid_SEC_MECH = 19;
    const ComponentId    TAG_SSL_SEC_TRANS = 20;
    const ComponentId    TAG_CSI_ECMA_Public_SEC_MECH = 21;
    const ComponentId    TAG_ GENERIC_SEC_MECH = 22;
    const ComponentId    TAG_FIREWALL_TRANS = 23;
    const ComponentId    TAG_SCCP_CONTACT_INFO = 24;
    const ComponentId    TAG_JAVA_CODEBASE = 25;
    const ComponentId    TAG_TRANSACTION_POLICY = 26;
    const ComponentId    TAG_MESSAGE_ROUTERS = 30;
    const ComponentId    TAG_OTS_POLICY = 31;
    const ComponentId    TAG_INV_POLICY = 32;
    const ComponentId    TAG_INET_SEC_TRANS = 123;
};
```

The following additional components that can be used by other protocols are specified in the DCE ESIOP chapter of this document and *CORBAServices*, Security Service, in the Security Service for DCE ESIOP section:

```
    const ComponentId    TAG_COMPLETE_OBJECT_KEY = 5;
    const ComponentId    TAG_ENDPOINT_ID_POSITION = 6;
    const ComponentId    TAG_LOCATION_POLICY = 12;
    const ComponentId    TAG_DCE_STRING_BINDING = 100;
    const ComponentId    TAG_DCE_BINDING_NAME = 101;
    const ComponentId    TAG_DCE_NO_PIPES = 102;
```

```
const ComponentId    TAG_DCE_SEC_MECH = 103; // Security Service
```

13.6.6.1 TAG_ORB_TYPE Component

It is often useful in the real world to be able to identify the particular kind of ORB an object reference is coming from, to work around problems with that particular ORB, or exploit shared efficiencies.

The **TAG_ORB_TYPE** component has an associated value of type **unsigned long**, encoded as a CDR encapsulation, designating an ORB type ID allocated by the OMG for the ORB type of the originating ORB. Anyone may register any ORB types by submitting a short (one-paragraph) description of the ORB type to the OMG, and will receive a new ORB type ID in return. A list of ORB type descriptions and values will be made available on the OMG web server.

The **TAG_ORB_TYPE** component can appear at most once in any IOR profile. For profiles supporting IIOP 1.1 or greater, it is optionally present.

13.6.6.2 TAG_ALTERNATE_IIOP_ADDRESS Component

In cases where the same object key is used for more than one internet location, the following standard IOR Component is defined for support in IIOP version 1.2.

The **TAG_ALTERNATE_IIOP_ADDRESS** component has an associated value of type

```
struct {
    string HostID,
    unsigned short Port
};
```

encoded as a CDR encapsulation.

Zero or more instances of the **TAG_ALTERNATE_IIOP_ADDRESS** component type may be included in a version 1.2 **TAG_INTERNET_IOP** Profile. Each of these alternative addresses may be used by the client orb, in addition to the host and port address expressed in the body of the Profile. In cases where one or more **TAG_ALTERNATE_IIOP_ADDRESS** components are present in a **TAG_INTERNET_IOP** Profile, no order of use is prescribed by Version 1.2 of IIOP.

13.6.6.3 Other Components

The following standard components are specified in various OMG specifications:

- **TAG_CODE_SETS** - See Section 13.10.2.4, "CodeSet Component of IOR Multi-Component Profile" on page 13-42.
- **TAG_POLICIES** - See CORBA Messaging - chapter 22.
- **TAG_SEC_NAME** - See the Security Service specification, Mechanism Tags section.

- **TAG_ASSOCIATION_OPTIONS** - See the Security Service specification, Tag Association Options section.
- **TAG_SSL_SEC_TRANS** - See the Security Service specification, Mechanism Tags section.
- **TAG_GENERIC_SEC_MECH** and all other tags with names in the form **TAG_*_SEC_MECH** - See the Security Service specification, Mechanism Tags section.
- **TAG_FIREWALL_SEC** - See the Firewall specification (orbos/98-05-04).
- **TAG_SCCP_CONTACT_INFO** - See the CORBA/IN Interworking specification (telecom/98-10-03).
- **TAG_JAVA_CODEBASE** - See the Java to IDL Language Mapping specification (formal/99-07-59), Codebase Transmission section.
- **TAG_TRANSACTION_POLICY** - See the Object Transaction Service specification (formal/00-06-28).
- **TAG_MESSAGE_ROUTERS** - See CORBA Messaging (chapter 22).
- **TAG_OTS_POLICY** - See the Object Transaction Service specification (formal/00-06-28).
- **TAG_INV_POLICY** - See the Object Transaction Service specification (formal/00-06-28).
- **TAG_INET_SEC_TRANS** - See the Security Service specification (formal/00-06-25).
- **TAG_COMPLETE_OBJECT_KEY** (See Section 16.5.4, "Complete Object Key Component" on page 16-19).
- **TAG_ENDPOINT_ID_POSITION** (See Section 16.5.5, "Endpoint ID Position Component" on page 16-20).
- **TAG_LOCATION_POLICY** (See Section 16.5.6, "Location Policy Component" on page 16-20).
- **TAG_DCE_STRING_BINDING** (See Section 16.5.1, "DCE-CIOP String Binding Component" on page 16-17).
- **TAG_DCE_BINDING_NAME** (See Section 16.5.2, "DCE-CIOP Binding Name Component" on page 16-18).
- **TAG_DCE_NO_PIPES** (See Section 16.5.3, "DCE-CIOP No Pipes Component" on page 16-19).

13.6.7 Profile and Component Composition in IORs

The following rules augment the preceding discussion:

1. Profiles must be independent, complete, and self-contained. Their use shall not depend on information contained in another profile.
2. Any invocation uses information from exactly one profile.