Ignoring the difference between the performance of local and remote invocations can lead to designs whose implementations are virtually assured of having performance problems because the design requires a large amount of communication between components that are in different address spaces and on different machines. Ignoring the difference in the time it takes to make a remote object invocation and the time it takes to make a local object invocation is to ignore one of the major design areas of an application. A properly designed application will require determining, by understanding the application being designed, what objects can be made remote and what objects must be clustered together.

The vision outlined earlier, however, has an answer to this objection. The answer is two-pronged. The first prong is to rely on the steadily increasing speed of the underlying hardware to make the difference in latency irrelevant. This, it is often argued, is what has happened to efficiency concerns having to do with everything from high level languages to virtual memory. Designing at the cutting edge has always required that the hardware catch up before the design is efficient enough for the real world. Arguments from efficiency seem to have gone out of style in software engineering, since in the past such concerns have always been answered by speed increases in the underlying hardware.

The second prong of the reply is to admit to the need for tools that will allow one to see what the pattern of communication is between the objects that make up an application. Once such tools are available, it will be a matter of tuning to bring objects that are in constant contact to the same address space, while moving those that are in relatively infrequent contact to wherever is most convenient. Since the vision allows all objects to communicate using the same underlying mechanism, such tuning will be possible by simply altering the implementation details (such as object location) of the relevant objects. However, it is important to get the application correct first, and after that one can worry about efficiency.

Whether or not it will ever become possible to mask the efficiency difference between a local object invocation and a distributed object invocation is not answerable *a priori.* Fully masking the distinction would require not only advances in the technology underlying remote object invocation, but would also require changes to the general programming model used by developers.

If the only difference between local and distributed object invocations was the difference in the amount of time it took to make the call, one could strive for a future in which the two kinds of calls would be conceptually indistinguishable. Whether the technology of distributed computing has moved far enough along to allow one to plan products based on such technology would be a matter of judgement, and rational people could disagree as to the wisdom of such an approach.

However, the difference in latency between the two kinds of calls is only the most obvious difference. Indeed, this difference is not really the fundamental difference between the two kinds of calls, and that even if it were possible to develop

the technology of distributed calls to an extent that the difference in latency between the two sorts of calls was minimal, it would be unwise to construct a programming paradigm that treated the two calls as essentially similar. In fact, the difference in latency between local and remote calls, because it is so obvious, has been the only difference most see between the two, and has tended to mask the more irreconcilable differences.

A.4.2 Memory Access

A more fundamental (but still obvious) difference between local and remote computing concerns the access to memory in the two cases—specifically in the use of pointers. Simply put, pointers in a local address space are not valid in another (remote) address space. The system can paper over this difference, but for such an approach to be successful, the transparency must be complete. Two choices exist: either all memory access must be controlled by the underlying system, or the programmer must be aware of the different types of access—local and remote. There is no inbetween.

If the desire is to completely unify the programming model—to make remote accesses behave as if they were in fact local—the underlying mechanism must totally control all memory access. Providing distributed shared memory is one way of completely relieving the programmer from worrying about remote memory access (or the difference between local and remote). Using the object-oriented paradigm to the fullest, and requiring the programmer to build an application with "objects all the way down," (that is, only object references or values are passed as method arguments) is another way to eliminate the boundary between local and remote computing. The layer underneath can exploit this approach by marshalling and unmarshalling method arguments and return values for intra-address space transmission.

But adding a layer that allows the replacement of all pointers to objects with object references only *permits* the developer to adopt a unified model of object interaction. Such a unified model cannot be *enforced* unless one also removes the ability to get address-space-relative pointers from the language used by the developer. Such an approach erects a barrier to programmers who want to start writing distributed applications, in that it requires that those programmers learn a new style of programming which does not use address-space-relative pointers. In requiring that programmers learn such a language, moreover, one gives up the complete transparency between local and distributed computing.[A]

Even if one were to provide a language that did not allow obtaining address-space-relative pointers to objects (or returned an object reference whenever such a pointer was requested), one would need to provide an equivalent way of making

cross-address space reference to entities other than objects. Most programmers use pointers as references for many different kinds of entities. These pointers must either be replaced with something that can be used in cross-address space calls or the programmer will need to be aware of the difference between such calls (which will either not allow pointers to such entities, or do something special with those pointers) and local calls. Again, while this could be done, it does violate the doctrine of complete unity between local and remote calls. Because of memory access constraints, the two *have* to differ.

The danger lies in promoting the myth that "remote access and local access are exactly the same" and not enforcing the myth. An underlying mechanism that does not unify all memory accesses while still promoting this myth is both misleading and prone to error. Programmers buying into the myth may believe that they do not have to change the way they think about programming. The programmer is therefore quite likely to make the mistake of using a pointer in the wrong context, producing incorrect results. "Remote is just like local," such programmers think, "so we have just one unified programming model." Seemingly, programmers need not change their style of programming. In an incomplete implementation of the underlying mechanism, or one that allows an implementation language that in turn allows direct access to local memory, the system does not take care of all memory accesses, and errors are bound to occur. These errors occur because the programmer is not aware of the difference between local and remote access and what is actually happening "under the covers."

The alternative is to explain the difference between local and remote access, making the programmer aware that remote address space access is very different from local access. Even if some of the pain is taken away by using an interface definition language like that specified in [1] and having it generate an intelligent language mapping for operation invocation on distributed objects, the programmer aware of the difference will not make the mistake of using pointers for cross-address space access. The programmer will know it is incorrect. By not masking the difference, the programmer is able to learn when to use one method of access and when to use the other.

Just as with latency, it is logically possible that the difference between local and remote memory access could be completely papered over and a single model of both presented to the programmer. When we turn to the problems introduced to distributed computing by partial failure and concurrency, however, it is not clear that such a unification is even conceptually possible.

A.5 Partial Failure and Concurrency

While unlikely, it is at least logically possible that the differences in latency and memory access between local computing and distributed computing could be masked. It is not clear that such a masking could be done in such a way that the local computing paradigm could be used to produce distributed applications, but it might still be possible to allow some new programming technique to be used for both activities. Such a masking does not even seem to be logically possible, however, in the case of partial failure and concurrency. These aspects appear to be different in kind in the case of distributed and local computing.[2]

Partial failure is a central reality of distributed computing. Both the local and the distributed world contain components that are subject to periodic failure. In the case of local computing, such failures are either total, affecting all of the entities that are working together in an application, or detectable by some central resource allocator (such as the operating system on the local machine).

This is not the case in distributed computing, where one component (machine, network link) can fail while the others continue. Not only is the failure of the distributed components independent, but there is no common agent that is able to determine what component has failed and inform the other components of that failure, no global state that can be examined that allows determination of exactly what error has occurred. In a distributed system, the failure of a network link is indistinguishable from the failure of a processor on the other side of that link.

These sorts of failures are not the same as mere exception raising or the inability to complete a task, which can occur in the case of local computing. This type of failure is caused when a machine crashes during the execution of an object invocation or a network link goes down, occurrences that cause the target object to simply disappear rather than return control to the caller. A central problem in distributed computing is insuring that the state of the whole system is consistent after such a failure; this is a problem that simply does not occur in local computing.

The reality of partial failure has a profound effect on how one designs interfaces and on the semantics of the operations in an interface. Partial failure requires that programs deal with indeterminacy. When a local component fails, it is possible to know the state of the system that caused the failure and the state of the system after the failure. No such determination can be made in the case of a distributed system. Instead, the interfaces that are used for the communication must be designed in such a way that it is possible for the objects to react in a consistent way to possible partial failures.

---

[2] In fact, authors such as Schroeder[12] and Hadzilacos and Toueg[13] take partial failure and concurrency to be the defining problems of distributed computing.

Being robust in the face of partial failure requires some expression at the interface level. Merely improving the implementation of one component is not sufficient. The interfaces that connect the components must be able to state whenever possible the cause of failure, and there must be interfaces that allow reconstruction of a reasonable state when failure occurs and the cause cannot be determined.

If an object is co-resident in an address space with its caller, partial failure is not possible. A function may not complete normally, but it always completes. There is no indeterminism about how much of the computation completed. Partial completion can occur only as a result of circumstances that will cause the other components to fail.

The addition of partial failure as a possibility in the case of distributed computing does not mean that a single object model cannot be used for both distributed computing and local computing. The question is not "can you make remote method invocation look like local method invocation?" but rather "what is the price of making remote method invocation identical to local method invocation?" One of two paths must be chosen if one is going to have a unified model.

The first path is to treat all objects as if they were local and design all interfaces as if the objects calling them, and being called by them, were local. The result of choosing this path is that the resulting model, when used to produce distributed systems, is essentially indeterministic in the face of partial failure and consequently fragile and non-robust. This path essentially requires ignoring the extra failure modes of distributed computing. Since one can't get rid of those failures, the price of adopting the model is to require that such failures are unhandled and catastrophic.

The other path is to design all interfaces as if they were remote. That is, the semantics and operations are all designed to be deterministic in the face of failure, both total and partial. However, this introduces unnecessary guarantees and semantics for objects that are never intended to be used remotely. Like the approach to memory access that attempts to require that all access is through system-defined references instead of pointers, this approach must also either rely on the discipline of the programmers using the system or change the implementation language so that all of the forms of distributed indeterminacy are forced to be dealt with on all object invocations.

This approach would also defeat the overall purpose of unifying the object models. The real reason for attempting such a unification is to make distributed computing more like local computing and thus make distributed computing easier. This second approach to unifying the models makes local computing as complex as distributed computing. Rather than encouraging the production of distributed applications, such a model will discourage its own adoption by making all object-based computing more difficult.

Similar arguments hold for concurrency. Distributed objects by their nature must handle concurrent method invocations. The same dichotomy applies if one insists on a unified programming model. Either all objects must bear the weight of concurrency semantics, or all objects must ignore the problem and hope for the best when distributed. Again, this is an interface issue and not solely an implementation issue, since dealing with concurrency can take place only by passing information from one object to another through the agency of the interface. So either the overall programming model must ignore significant modes of failure, resulting in a fragile system; or the overall programming model must assume a worst-case complexity model for all objects within a program, making the production of any program, distributed or not, more difficult.

One might argue that a multi-threaded application needs to deal with these same issues. However, there is a subtle difference. In a multi-threaded application, there is no real source of indeterminacy of invocations of operations. The application programmer has complete control over invocation order when desired. A distributed system by its nature introduces truly asynchronous operation invocations. Further, a non-distributed system, even when multi-threaded, is layered on top of a single operating system that can aid the communication between objects and can be used to determine and aid in synchronization and in the recovery of failure. A distributed system, on the other hand, has no single point of resource allocation, synchronization, or failure recovery, and thus is conceptually very different.

A.6 The Myth of "Quality of Service"

One could take the position that the way an object deals with latency, memory access, partial failure, and concurrency control is really an aspect of the implementation of that object, and is best described as part of the "quality of service" provided by that implementation. Different implementations of an interface may provide different levels of reliability, scalability, or performance. If one wants to build a more reliable system, one merely needs to choose more reliable implementations of the interfaces making up the system.

On the surface, this seems quite reasonable. If I want a more robust system, I go to my catalog of component vendors. I quiz them about their test methods. I see if they have ISO9000 certification, and I buy my components from the one I trust the most. The components all comply with the defined interfaces, so I can plug them right in; my system is robust and reliable, and I'm happy.

Let us imagine that I build an application that uses the (mythical) queue interface to enqueue work for some component. My application dutifully enqueues records that represent work to be done. Another application dutifully dequeues them and performs the work. After a while, I notice that my application crashes

due to time-outs. I find this extremely annoying, but realize that it's my fault. My application just isn't robust enough. It gives up too easily on a time-out. So I change my application to retry the operation until it succeeds. Now I'm happy. I almost never see a time-out. Unfortunately, I now have another problem. Some of the requests seem to get processed two, three, four, or more times. How can this be? The component I bought which implements the queue has allegedly been rigorously tested. It shouldn't be doing this. I'm angry. I call the vendor and yell at him. After much fingerpointing and research, the culprit is found. The problem turns out to be the way I'm using the queue. Because of my handling of partial failures (which in my naivete, I had thought to be total), I have been enqueuing work requests multiple times.

Well, I yell at the vendor that it is still their fault. Their queue should be detecting the duplicate entry and removing it. I'm not going to continue using this software unless this is fixed. But, since the entities being enqueued are just values, there is no way to do duplicate elimination. The only way to fix this is to change the protocol to add request IDs. But since this is a standardized interface, there is no way to do this.

The moral of this tale is that robustness is not simply a function of the implementations of the interfaces that make up the system. While robustness of the individual components has some effect on the robustness of the overall systems, it is not the sole factor determining system robustness. Many aspects of robustness can be reflected only at the protocol/interface level.

Similar situations can be found throughout the standard set of interfaces. Suppose I want to reliably remove a name from a context. I would be tempted to write code that looks like:

```
while (true) {
    try {
        context->remove(name);
        break;
    }
    catch (NotFoundInContext) {
        break;
    }
    catch (NetworkServerFaliure) {
        continue;
    }
}
```

That is, I keep trying the operation until it succeeds (or until I crash). The problem is that my connection to the name server may have gone down, but another client's may have stayed up. I may have, in fact, successfully removed the name but not

discovered it because of a network disconnection. The other client then adds the same name, which I then remove. Unless the naming interface includes an operation to lock a naming context, there is no way that I can make this operation completely robust. Again, we see that robustness/reliability needs to be expressed at the interface level. In the design of any operation, the question has to be asked: What happens if the client chooses to repeat this operation with the exact same parameters as previously? What mechanisms are needed to ensure that they get the desired semantics? These are things that can be expressed only at the interface level. These are issues that can't be answered by supplying a "more robust implementation" because the lack of robustness is inherent in the interface and not something that can be changed by altering the implementation.

Similar arguments can be made about performance. Suppose an interface describes an object which maintains sets of other objects. A defining property of sets is that there are no duplicates. Thus, the implementation of this object needs to do duplicate elimination. If the interfaces in the system do not provide a way of testing equality of reference, the objects in the set must be queried to determine equality. Thus, duplicate elimination can be done only by interacting with the objects in the set. It doesn't matter how fast the objects in the set implement the equality operation. The overall performance of eliminating duplicates is going to be governed by the latency in communicating over the slowest communications link involved. There is no change in the set implementations that can overcome this. An interface design issue has put an upper bound on the performance of this operation.

A.7 Lessons From NFS

We do not need to look far to see the consequences of ignoring the distinction between local and distributed computing at the interface level. NFS®, Sun's distributed computing file system[14,15] is an example of a non-distributed application programer interface (API) (open, read, write, close, etc.) re-implemented in a distributed way.

Before NFS and other network file systems, an error status returned from one of these calls indicated something rare: a full disk, or a catastrophe such as a disk crash. Most failures simply crashed the application along with the file system. Further, these errors generally reflected a situation that was either catastrophic for the program receiving the error or one that the user running the program could do something about.

NFS opened the door to partial failure within a file system. It has essentially two modes for dealing with an inaccessible file server: soft mounting and hard mounting. But since the designers of NFS were unwilling (for easily understand-

able reasons) to change the interface to the file system to reflect the new, distributed nature of file access, neither option is particularly robust.

Soft mounts expose network or server failure to the client program. Read and write operations return a failure status much more often than in the single-system case, and programs written with no allowance for these failures can easily corrupt the files used by the program. In the early days of NFS, system administrators tried to tune various parameters (time-out length, number of retries) to avoid these problems. These efforts failed. Today, soft mounts are seldom used, and when they are used, their use is generally restricted to read-only file systems or special applications.

Hard mounts mean that the application hangs until the server comes back up. This generally prevents a client program from seeing partial failure, but it leads to a malady familiar to users of workstation networks: one server crashes, and many workstations—even those apparently having nothing to do with that server—freeze. Figuring out the chain of causality is very difficult, and even when the cause of the failure can be determined, the individual user can rarely do anything about it but wait. This kind of brittleness can be reduced only with strong policies and network administration aimed at reducing interdependencies. Nonetheless, hard mounts are now almost universal.

Note that because the NFS protocol is stateless, it assumes clients contain no state of interest with respect to the protocol; in other words, the server doesn't care what happens to the client. NFS is also a "pure" client-server protocol, which means that failure can be limited to three parties: the client, the server, or the network. This combination of features means that failure modes are simpler than in the more general case of peer-to-peer distributed object-oriented applications where no such limitation on shared state can be made and where servers are themselves clients of other servers. Such peer-to-peer distributed applications can and will fail in far more intricate ways than are currently possible with NFS.

The limitations on the reliability and robustness of NFS have nothing to do with the implementation of the parts of that system. There is no "quality of service" that can be improved to eliminate the need for hard mounting NFS volumes. The problem can be traced to the interface upon which NFS is built, an interface that was designed for non-distributed computing where partial failure was not possible. The reliability of NFS cannot be changed without a change to that interface, a change that will reflect the distributed nature of the application.

This is not to say that NFS has not been successful. In fact, NFS is arguably the most successful distributed application that has been produced. But the limitations on the robustness have set a limitation on the scalability of NFS. Because of the intrinsic unreliability of the NFS protocol, use of NFS is limited to fairly small numbers of machines, geographically co-located and centrally administered. The way NFS has dealt with partial failure has been to informally require a centralized

resource manager (a system administrator) who can detect system failure, initiate resource reclamation and insure system consistency. But by introducing this central resource manager, one could argue that NFS is no longer a genuinely distributed application.

A.8 Taking the Difference Seriously

Differences in latency, memory access, partial failure, and concurrency make merging of the computational models of local and distributed computing both unwise to attempt and unable to succeed. Merging the models by making local computing follow the model of distributed computing would require major changes in implementation languages (or in how those languages are used) and make local computing far more complex than is otherwise necessary. Merging the models by attempting to make distributed computing follow the model of local computing requires ignoring the different failure modes and basic indeterminacy inherent in distributed computing, leading to systems that are unreliable and incapable of scaling beyond small groups of machines that are geographically colocated and centrally administered.

A better approach is to accept that there are irreconcilable differences between local and distributed computing, and to be conscious of those differences at all stages of the design and implementation of distributed applications. Rather than trying to merge local and remote objects, engineers need to be constantly reminded of the differences between the two, and know when it is appropriate to use each kind of object.

Accepting the fundamental difference between local and remote objects does not mean that either sort of object will require its interface to be defined differently. An interface definition language such as IDL[B] can still be used to specify the set of interfaces that define objects. However, an additional part of the definition of a class of objects will be the specification of whether those objects are meant to be used locally or remotely. This decision will need to consider what the anticipated message frequency is for the object, and whether clients of the object can accept the indeterminacy implied by remote access. The decision will be reflected in the interface to the object indirectly, in that the interface for objects that are meant to be accessed remotely will contain operations that allow reliability in the face of partial failure.

It is entirely possible that a given object will often need to be accessed by some objects in ways that cannot allow indeterminacy, and by other objects relatively rarely and in a way that does allow indeterminacy. Such cases should be split into two objects (which might share an implementation) with one having an

interface that is best for local access and the other having an interface that is best for remote access.

A compiler for the interface definition language used to specify classes of objects will need to alter its output based on whether the class definition being compiled is for a class to be used locally or a class being used remotely. For interfaces meant for distributed objects, the code produced might be very much like that generated by RPC stub compilers today. Code for a local interface, however, could be much simpler, probably requiring little more than a class definition in the target language.

While writing code, engineers will have to know whether they are sending messages to local or remote objects, and access those objects differently. While this might seem to add to the programming difficulty, it will in fact aid the programmer by providing a framework under which he or she can learn what to expect from the different kinds of calls. To program completely in the local environment, according to this model, will not require any changes from the programmer's point of view. The discipline of defining classes of objects using an interface definition language will insure the desired separation of interface from implementation, but the actual process of implementing an interface will be no different than what is done today in an object-oriented language.

Programming a distributed application will require the use of different techniques than those used for non-distributed applications. Programming a distributed application will require thinking about the problem in a different way than before it was thought about when the solution was a non-distributed application. But that is only to be expected. Distributed objects are different from local objects, and keeping that difference visible will keep the programmer from forgetting the difference and making mistakes. Knowing that an object is outside of the local address space, and perhaps on a different machine, will remind the programmer that he or she needs to program in a way that reflects the kinds of failures, indeterminacy, and concurrency constraints inherent in the use of such objects. Making the difference visible will aid in making the difference part of the design of the system.

Accepting that local and distributed computing are different in an irreconcilable way will also allow an organization to allocate its research and engineering resources more wisely. Rather than using those resources in attempts to paper over the differences between the two kinds of computing, resources can be directed at improving the performance and reliability of each.

One consequence of the view espoused here is that it is a mistake to attempt to construct a system that is "objects all the way down" if one understands the goal as a distributed system constructed of the *same kind* of objects all the way down. There will be a line where the object model changes; on one side of the line will be distributed objects, and on the other side of the line there will (perhaps) be

local objects. On either side of the line, entities on the other side of the line will be opaque; thus one distributed object will not know (or care) if the implementation of another distributed object with which it communicates is made up of objects or is implemented in some other way. Objects on different sides of the line will differ in kind and not just in degree; in particular, the objects will differ in the kinds of failure modes with which they must deal.

A.9 A Middle Ground

As noted in Section A.2, the distinction between local and distributed objects as we are using the terms is not exhaustive. In particular, there is a third category of objects made up of those that are in different address spaces but are guaranteed to be on the same machine. These are the sorts of objects, for example, that appear to be the basis of systems such as Spring[16] or Clouds[4]. These objects have some of the characteristics of distributed objects, such as increased latency in comparison to local objects and the need for a different model of memory access. However, these objects also share characteristics of local objects, including sharing underlying resource management and failure modes that are more nearly deterministic.

It is possible to make the programming model for such "local-remote" objects more similar to the programming model for local objects than can be done for the general case of distributed objects. Even though the objects are in different address spaces, they are managed by a single resource manager. Because of this, partial failure and the indeterminacy that it brings can be avoided. The programming model for such objects will still differ from that used for objects in the same address space with respect to latency, but the added latency can be reduced to generally acceptable levels. The programming models will still necessarily differ on methods of memory access and concurrency, but these do not have as great an effect on the construction of interfaces as additional failure modes.

The other reason for treating this class of objects separately from either local objects or generally distributed objects is that a compiler for an interface definition language can be significantly optimized for such cases. Parameter and result passing can be done via shared memory if it is known that the objects communicating are on the same machine. At the very least, marshalling of parameters and the unmarshalling of results can be avoided.

The class of locally distributed objects also forms a group that can lead to significant gains in software modularity. Applications made up of collections of such objects would have the advantage of forced and guaranteed separation between the interface to an object and the implementation of that object, and would allow the replacement of one implementation with another without affecting other parts of the system. Because of this, it might be advantageous to investigate the uses of

such a system. However, this activity should not be confused with the unification of local objects with the kinds of distributed objects we have been discussing.

A.10 References

[1] The Object Management Group. "Common Object Request Broker: Architecture and Specification." *OMG Document Number 91.12.1* (1991).

[2] Parrington, Graham D. "Reliable Distributed Programming in C++: The Arjuna Approach." *USENIX 1990 C++ Conference Proceedings* (1991).

[3] Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald." *IEEE Transactions on Software Engineering* SE-13, no. 1, (January 1987).

[4] Dasgupta, P., R. J. Leblanc, and E. Spafford. "The Clouds Project: Designing and Implementing a Fault Tolerant Distributed Operating System." *Georgia Institute of Technology Technical Report GIT-ICS-85/29. (*1985).

[5] Microsoft Corporation. *Object Linking and Embedding Programmers Reference*. version 1. Microsoft Press, 1992.

[6] Linton, Mark. "A Taste of Fresco." Tutorial given at the *8th Annual X Technical Conference (*January 1994).

[7] Jaayeri, M., C. Ghezzi, D. Hoffman, D. Middleton, and M. Smotherman. "CSP/80: A Language for Communicating Sequential Processes." *Proceedings: Distributed Computing CompCon* (Fall 1980).

[8] Cook, Robert. "MOD— A Language for Distributed Processing." *Proceedings of the 1st International Conference on Distributed Computing Systems (*October 1979).

[9] Birrell, A. D. and B. J. Nelson. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems* 2 (1978).

[10] Hutchinson, N. C., L. L. Peterson, M. B. Abott, and S. O'Malley. "RPC in the x-Kernel: Evaluating New Design Techniques." *Proceedings of the Twelfth Symposium on Operating Systems Principles* 23, no. 5 (1989).

[11] Zahn, L., T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, and G. Wyant. *Network Computing Architecture*. Prentice Hall, 1990.

[12] Schroeder, Michael D. "A State-of-the-Art Distributed System: Computing with BOB." In *Distributed Systems*, 2nd ed., S. Mullender, ed., ACM Press, 1993.

[13] Hadzilacos, Vassos and Sam Toueg. "Fault-Tolerant Broadcasts and Related Problems." In *Distributed Systems*, 2nd ed., S. Mullendar, ed., ACM Press, 1993.

[14] Walsh, D., B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. "Overview of the SUN Network File System." *Proceedings of the Winter Usenix Conference* (1985).

[15] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. "Design and Implementation of the SUN Network File System." *Proceedings of the Summer Usenix Conference (*1985).

[16] Khalidi, Yousef A. and Michael N. Nelson. "An Implementation of UNIX on an Object-Oriented Operating System." *Proceedings of the Winter Usenix Conference* (1993). Also *Sun Microsystems Laboratories, Inc. Technical Report SMLI TR-92-3* (December 1992).

A.11 Observations for this Reprinting

[A] When this note was written, the major system programming languages (C, C++, Modula3, etc.) all allowed direct access, to a greater or lesser degree, to pointers to internal memory. This paragraph points out that adding indirect references to such languages would allow two kinds of reference, one of which was distribution transparent while the other was not. Java, of course, does not have direct access to pointers. Because of the Java use of references within the language, it does provide a platform in which address-space-relative pointers are missing. Thus Java not only permits a unified addressing scheme, it enforces that scheme.

[B] There are actually a number of interface definition languages that are referred to by the initials IDL. When this note was originally written, we were referring to the CORBA interface definition language. However, the other languages that use this name share the characteristics discussed here, so the argument presented would apply equally to them.

APPENDIX B

The Example Code

The first rule of magic is simple:
Don't waste your time waving your hands and hoping
when a rock or a club will do.
—McCloctnik the Lucid

THE following pages contain the complete code for the examples used in the introductory chapters of this book. The sources are listed in alphabetical order by the full name, including the package name. For your convenience, here is a mapping from the simple class name to its fully-qualified class name:

`ChatMessage`	`chat.ChatMessage`	328
`ChatProxy`	`chat.ChatProxy`	330
`ChatServer`	`chat.ChatServer`	332
`ChatServerAdmin`	`chat.ChatServerAdmin`	333
`ChatServerImpl`	`chat.ChatServerImpl`	337
`ChatSpeaker`	`chat.ChatSpeaker`	344
`ChatStream`	`chat.ChatStream`	345
`ChatSubject`	`chat.ChatSubject`	347
`Chatter`	`chatter.Chatter`	348
`ChatterThread`	`chatter.ChatterThread`	350
`FortuneAdmin`	`fortune.FortuneAdmin`	360
`FortuneStream`	`fortune.FortuneStream`	362
`FortuneStreamImpl`	`fortune.FortuneStreamImpl`	363
`FortuneTheme`	`fortune.FortuneTheme`	368
`MessageStream`	`message.MessageStream`	369
`ParseUtil`	`util.ParseUtil`	370
`StreamReader`	`client.StreamReader`	352

You can also find the code at `http://java.sun.com/docs/books/jini/`

328

chat.ChatMessage

```
package chat;

import java.io.Serializable;

/**
 * A single message in the <CODE>ChatStream</CODE>.  This is the
 * type of <CODE>Object</CODE> returned by <CODE>ChatStream.nextMessage</CODE>.
 *
 * @see ChatStream
 */
public class ChatMessage implements Serializable {
    /**
     * The speaker of the message.
     * @serial
     */
    private String speaker;

    /**
     * The contents of the message.
     * @serial
     */
    private String[] content;

    /**
     * The serial version UID.  Stating it explicitly is good.
     *
     * @see fortune.FortuneTheme#serialVersionUID
     */
    static final long serialVersionUID =
                            -1852351967189107571L;

    /**
     * Create a new <CODE>ChatMessage</CODE> with the given
     * <CODE>speaker</CODE> and <CODE>content</CODE>.
     */
    public ChatMessage(String speaker, String[] content) {
        this.speaker = speaker;
        this.content = content;
    }

    /**
     * Return the speaker of the message.
     */
    public String getSpeaker() { return speaker; }

    /**
     * Return the content of the message.  Each string in the array
     * represents a single line of content.
```

```
     */
    public String[] getContent() { return content; }

    // inherit doc comment from superclass
    public String toString() {
        StringBuffer buf = new StringBuffer(speaker);
        buf.append(": ");
        for (int i = 0; i < content.length; i++)
            buf.append(content[i]).append('\n');
        buf.setLength(buf.length() - 1); // strip newline
        return buf.toString();
    }
}
```

330
chat.ChatProxy

```
    package chat;

import java.io.EOFException;
import java.io.Serializable;
import java.rmi.RemoteException;

/**
 * The client-side proxy for a <CODE>ChatServer</CODE>-based
 * <CODE>ChatStream</CODE> service.  This forwards most requests to the
 * server, remembering the last successfuly retrieved message index.
 */
class ChatProxy implements ChatStream, Serializable {
    /**
     * Reference to the remote server.
     * @serial
     */
    private final ChatServer server;

    /**
     * The index of the last entry successfully received.
     * @serial
     */
    private int lastIndex = -1;

    /**
     * Cache of the subject of the chat.
     */
    private transient String subject;

    /**
     * Create a new proxy that will talk to the given server object.
     */
    ChatProxy(ChatServer server) {
        this.server = server;
    }

    // inherit doc comment from ChatStream
    public synchronized Object nextMessage()
        throws RemoteException, EOFException
    {
        ChatMessage msg = server.nextInLine(lastIndex);
        lastIndex++;
        return msg;
    }

    // inherit doc comment from ChatStream
    public void add(String speaker, String[] msg)
        throws RemoteException
```

```
    {
        server.add(speaker, msg);
    }

    // inherit doc comment from ChatStream
    public synchronized String getSubject()
        throws RemoteException
    {
        if (subject == null)
            subject = server.getSubject();
        return subject;
    }

    // inherit doc comment from ChatStream
    public String[] getSpeakers() throws RemoteException {
        return server.getSpeakers();
    }
}
```

```
package chat;

import util.ParseUtil;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationException;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupID;
import java.rmi.activation.ActivationSystem;
import java.rmi.MarshalledObject;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Properties;

/**
 * The administrative program that creates a new <CODE>ChatServerImpl</CODE>
 * chat stream service.  It's invocation is:
 * <pre>
 *      java [<i>java-options</i>] chat.ChatServerAdmin <i>dir subject</i>
 *              [<i>groups|lookupURL classpath codebase policy-file</i>]
 * </pre>
 * Where the options are:
 * <dl>
 * <dt><i><CODE>java-options</CODE></i>
 * <dd>Options to the Java VM that will run the admin program.  Typically
 * this includes a security policy property.
 * <p>
 * <dt><i><CODE>dir</CODE></i>
 * <dd>The directory in which all the chats in the same group will live.
 * <p>
 * <dt><i><CODE>subject</CODE></i>
 * <dd>The subject of the chat.  This must be unique within the group.
 * <p>
 * <dt><i><CODE>groups</CODE></i>|<i><CODE>lookupURL</CODE></i>
 * <dd>Either a comma-separated list of groups in which all the services
 * in the group will be regsitered or a URL to a specific lookup service.
 * <p>
```

```
 * <dt><i><CODE>classpath</CODE></i>
 * <dd>The classpath for the activated service (<CODE>ChatServerImpl</CODE>
 * will be loaded from this).
 * <p>
 * <dt><i><CODE>codebase</CODE></i>
 * <dd>The codebase for users of the service (<CODE>ChatProxy</CODE> will
 * be loaded from this).
 * <p>
 * <dt><i><CODE>policy-file</CODE></i>
 * <dd>The policy file for the activated service's virtual machine.
 * </dl>
 * <p>The last four parameters imply creation of a new group.  If any
 * are specified they must all be specified.  If none are specified the
 * new chat stream will be in the same activation group as the others
 * who use the same storage directory, and so will use the same values
 * for the last four parameters.
 */
public class ChatServerAdmin {
    /**
     * The main program for <CODE>ChatServerAdmin</CODE>.
     */
    public static void main(String[] args) throws Exception
    {
        if (args.length != 2 && args.length != 6) {
            usage();            // print usage message
            System.exit(1);
        }

        File dir = new File(args[0]);
        String subject = args[1];

        ActivationGroupID group = null;
        if (args.length == 2)
            group = getGroup(dir);
        else {
            String[] groups = ParseUtil.parseGroups(args[2]);
            String lookupURL =
                (args[2].indexOf(':') > 0 ? args[2] : null);
            String classpath = args[3];
            String codebase = args[4];
            String policy = args[5];
            group = createGroup(dir, groups, lookupURL,
                classpath, codebase, policy);
        }

        File data = new File(dir, subject);
        MarshalledObject state = new MarshalledObject(data);
        ActivationDesc desc =
```

```
        new ActivationDesc(group, "chat.ChatServerImpl",
                           null, state, true);
    Remote newObj = Activatable.register(desc);
    ChatServer server = (ChatServer)newObj;
    String s = server.getSubject(); // force server up
    System.out.println("server created for " + s);
}

/**
 * Print a usage message for the user.
 */
private static void usage() {
    System.out.println("usage: java [java-options] " +
        ChatServerAdmin.class + " dir subject " +
        " [groups|lookupURL classpath codebase policy-file]\n");
}

/**
 * Create a new group with the given parameters.
 */
private static ActivationGroupID
    createGroup(File dir, String[] groups, String lookupURL,
                String classpath, String codebase,
                String policy)
    throws IOException, ActivationException
{
    if (!dir.isDirectory())
        dir.mkdirs();

    Properties props = new Properties();
    props.put("java.rmi.server.codebase", codebase);
    props.put("java.security.policy", policy);
    String[] argv = new String[] { "-cp", classpath };
    CommandEnvironment cmd =
        new CommandEnvironment("java", argv);
    ActivationSystem actSys = ActivationGroup.getSystem();
    ActivationGroupDesc groupDesc =
        new ActivationGroupDesc(props, cmd);
    ActivationGroupID id = actSys.registerGroup(groupDesc);

    FileOutputStream fout =
        new FileOutputStream(groupFile(dir));
    ObjectOutputStream out = new ObjectOutputStream(
        new BufferedOutputStream(fout));
    out.writeObject(id);
    out.writeObject(groups);
    out.writeObject(lookupURL);
    out.flush();            // force bits out of buffer
```

```
            fout.getFD().sync();    // force bits to the disk
            out.close();

            return id;
        }

        /**
         * Return a <CODE>File</CODE> object contains the group description.
         * This assumes that nobody will create a group with the subject
         * <CODE>"grpdesc"</CODE>.  This is probably a bad assumption -- a
         * fully robust implementation should either check this and forbid it
         * or figure out a way to store this someplace that does not conflict
         * with subject names.
         */
        static File groupFile(File dir) {
            return new File(dir, "grpdesc");
        }

        /**
         * Get the ActivationGroupID for the existing group in the given
         * directory.
         */
        private static ActivationGroupID getGroup(File dir)
            throws IOException, ClassNotFoundException
        {
            ObjectInputStream in = null;
            try {
                in = new ObjectInputStream(new BufferedInputStream(
                    new FileInputStream(groupFile(dir))));
                return (ActivationGroupID)in.readObject();
            } finally {
                if (in != null)
                    in.close();
            }
        }
    }
```

```
package chat;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceID;

import com.sun.jini.lease.LeaseRenewalManager;
import com.sun.jini.lookup.JoinManager;
import com.sun.jini.lookup.ServiceIDListener;
import com.sun.jini.reliableLog.LogHandler;
import com.sun.jini.reliableLog.ReliableLog;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

/**
 * The implementation of <CODE>ChatServer</CODE>.  This runs inside an
 * activation group defined by the persistent state from the activation
 * service.
 */
public class ChatServerImpl implements ChatServer {
    /**
     * The join manager we're using.
     */
    private JoinManager joinMgr;

    /**
     * Our subject of discussion.
     */
    private String subject;

    /**
     * The set of known speakers.
     */
    private Set speakers = new HashSet();
```

```
    /**
     * The list of messages.
     */
    private List messages = new ArrayList();

    /**
     * The list of service attributes.
     */
    private List attrs;

    /**
     * The service ID (or <CODE>null</CODE>).
     */
    private ServiceID serviceID;

    /**
     * Our persistent storage.
     */
    private ChatStore store;

    /**
     * Groups to register with (or an empty array).
     */
    private String[] groups = new String[0];

    /**
     * URL to specific join manager (or <CODE>null</CODE>).
     */
    private String lookupURL;

    /**
     * The lease renewal manager for all servers in our group.
     * We share it because this gives it more leases it might be
     * able to compress into single renewal messages.
     */
    private static LeaseRenewalManager renewer;

    /**
     * The storage for a <CODE>ChatServerImpl</CODE>.
     */
    class ChatStore extends LogHandler
        implements ServiceIDListener
    {
        /**
         * The reliable log in which we store our state.
         */
        private ReliableLog log;
```

```
/**
 * Create a new <CODE>ChatStore</CODE> object for the given
 * directory.  The directory is the full path for the specific
 * storage for this chat on the subject.  The parent directory
 * is the one for the group.
 */
ChatStore(File dir) throws IOException {
    // If the directory exists, recover from it.  Otherwise
    // create it as a a new subject.
    if (dir.exists()) {
        log = new ReliableLog(dir.toString(), this);
        log.recover();
    } else {
        subject = dir.getName();
        log = new ReliableLog(dir.toString(), this);
        attrs = new ArrayList();
        attrs.add(new ChatSubject(subject));
        log.snapshot();
    }

    // Read in the lookup groups and lookupURL for our service
    ObjectInputStream in = null;
    try {
        in = new ObjectInputStream(
            new FileInputStream(
                ChatServerAdmin.groupFile(dir.getParentFile())));
        in.readObject();        // skip over the group ID
        groups = (String[])in.readObject();
        lookupURL = (String)in.readObject();
    } catch (ClassNotFoundException e) {
        unexpectedException(e);
    } catch (IOException e) {
        unexpectedException(e);
    } finally {
        if (in != null)
            in.close();
    }
}

/**
 * Stores the current information in storage.  In our case only
 * the start state is snapshoted -- everything else is added
 * incrementally anyway and so the log of changes is the
 * state.  Part of <CODE>ReliableLogHandler</CODE>.
 */
public void snapshot(OutputStream out) throws Exception {
    ObjectOutputStream oo = new ObjectOutputStream(out);
    oo.writeObject(subject);
```

```
        oo.writeObject(attrs);
    }

    /**
     * Recovers the information from storage.  Part of
     * <CODE>ReliableLogHandler</CODE>.
     *
     * @see #snapshot
     */
    public void recover(InputStream in) throws Exception {
        ObjectInputStream oi = new ObjectInputStream(in);
        subject = (String)oi.readObject();
        attrs = (List)oi.readObject();
    }

    /**
     * Apply an update from the log during recovery.  The types
     * of data we add happen to all be distinct so we know exactly
     * what something is based on its type alone (lucky us).  Part
     * of <CODE>ReliableLogHandler</CODE>.
     */
    public void applyUpdate(Object update) throws Exception {
        if (update instanceof ChatMessage) {
            messages.add(update);
            addSpeaker(((ChatMessage)update).getSpeaker());
        } else if (update instanceof Entry) {
            attrs.add(update);
        } else if (update instanceof ServiceID) {
            serviceID = (ServiceID)update;
        } else {
            throw new IllegalArgumentException(
                "Internal error: update type " +
                update.getClass().getName() + ", " + update);
        }
    }

    /**
     * Invoked when the serviceID is first assigned to the service.
     * Part of <CODE>ServiceIDListener</CODE>.
     */
    public void serviceIDNotify(ServiceID serviceID) {
        try {
            log.update(serviceID);
        } catch (IOException e) {
            unexpectedException(e);
        }
        ChatServerImpl.this.serviceID = serviceID;
    }
```

```
    /**
     * Add a new speaker to the persistent storage log.
     */
    synchronized void add(ChatMessage msg) {
        try {
            log.update(msg, true);
        } catch (IOException e) {
            unexpectedException(e);
        }
    }
}

/**
 * The activation constructor for <CODE>ChatServerImpl</CODE>.  The
 * <CODE>state</CODE> object contains the directory which is our
 * reliable log directory.
 */
public ChatServerImpl(ActivationID actID,
                      MarshalledObject state)
    throws IOException, ClassNotFoundException
{
    File dir = (File) state.get();
    store = new ChatStore(dir);
    ChatProxy proxy = new ChatProxy(this);

    LookupLocator[] locators = null;
    if (lookupURL != null) {
        LookupLocator loc = new LookupLocator(lookupURL);
        locators = new LookupLocator[] { loc };
    }
    joinMgr = new JoinManager(proxy, getAttrs(), groups,
        locators, store, renewer);
    Activatable.exportObject(this, actID, 0);
}

/**
 * Return the attributes as an array for use in JoinManager.
 */
private Entry[] getAttrs() {
    return (Entry[])attrs.toArray(new Entry[attrs.size()]);
}

// inherit doc comment from ChatServer
public String getSubject() {
    return subject;
}
```

```
    // inherit doc comment from ChatServer
    public String[] getSpeakers() {
        return (String[])speakers.toArray(new String[speakers.size()]);
    }

    // inherit doc comment from ChatServer
    public synchronized void add(String speaker, String[] lines)
    {
        ChatMessage msg = new ChatMessage(speaker, lines);
        store.add(msg);
        addSpeaker(speaker);
        messages.add(msg);
        notifyAll();
    }

    /**
     * Add a speaker to the known list.  If the speaker is already
     * known, this does nothing.
     */
    private synchronized void addSpeaker(String speaker) {
        if (speakers.contains(speaker))
            return;
        speakers.add(speaker);
        Entry speakerAttr = new ChatSpeaker(speaker);
        attrs.add(speakerAttr);
        joinMgr.addAttributes(new Entry[] { speakerAttr });
    }

    // inherit doc comment from ChatServer
    public synchronized ChatMessage nextInLine(int index) {
        try {
            int nextIndex = index + 1;
            while (nextIndex >= messages.size())
                wait();
            return (ChatMessage)messages.get(nextIndex);
        } catch (InterruptedException e) {
            unexpectedException(e);
            return null; // keeps the compiler happy
        }
    }

    /**
     * Turn any unexpected exception into a runtime exception reflected
     * back to the client.  These are both unexpected and unrecoverable
     * exception (such as "file system full").
     */
    private static void unexpectedException(Throwable e) {
```

```
            throw new RuntimeException("unexpected exception: " + e);
        }
    }
```

344
chat.ChatSpeaker

```
package chat;

import net.jini.entry.AbstractEntry;
import net.jini.lookup.entry.ServiceControlled;

/**
 * An attribute for the <CODE>ChatStream</CODE> service that marks a
 * speaker as being present in a particular stream.
 *
 * @see ChatStream
 */
public class ChatSpeaker extends AbstractEntry
    implements ServiceControlled
{
    /**
     * The serial version UID.  Stating it explicitly is good.
     *
     * @see fortune.FortuneTheme#serialVersionUID
     */
    static final long serialVersionUID =
                            6748592884814857788L;

    /**
     * The speaker's name.
     * @serial
     */
    public String speaker;

    /**
     * Public no-arg constructor.  Required for all <CODE>Entry</CODE>
     * objects.
     */
    public ChatSpeaker() { }

    /**
     * Create a new <CODE>ChatSpeaker</CODE> with the given speaker.
     */
    public ChatSpeaker(String speaker) {
        this.speaker = speaker;
    }
}
```

```
package chat;

import message.MessageStream;

import java.rmi.RemoteException;

/**
 * A type of <CODE>MessageStream</CODE> whose contents are a chat
 * session.  The <CODE>nextMessage</CODE> method blocks if there is
 * as yet no next message in the stream.  The messages in the stream
 * are ordered, so <CODE>nextMessage</CODE> must be idempotent -- should
 * the client receive a <CODE>RemoteException</CODE>, the next invocation
 * must return the next message that the client has not yet seen.
 * <p>
 * Each message returned by <CODE>nextMessage</CODE> is a
 * <CODE>ChatMessage</CODE> object that has a speaker and what they
 * said.
 *
 * @see ChatMessage
 * @see ChatSpeaker
 * @see ChatSubject
 */
public interface ChatStream extends MessageStream {
    /**
     * Add a new message to the stream.  If the speaker is previously
     * unknown in the stream, a <CODE>ChatSpeaker</CODE> attribute
     * will be added to the service.
     *
     * @see ChatSpeaker
     */
    public void add(String speaker, String[] message)
        throws RemoteException;

    /**
     * Return the subject of the chat.  This does not change during the
     * lifetime of the service.  This subject will also exist as a
     * <CODE>ChatSubject</CODE> attribute on the service.
     *
     * @see ChatSubject
     */
    public String getSubject() throws RemoteException;

    /**
     * Return the list of speakers currently known in the stream.
     * The order is not significant.
     *
     * @see ChatSpeaker
```

```
         */
        public String[] getSpeakers() throws RemoteException;
    }
```

```
    package chat;

import net.jini.entry.AbstractEntry;
import net.jini.lookup.entry.ServiceControlled;

/**
 * An attribute for the <CODE>ChatStream</CODE> service that marks the
 * subject of discussion.
 *
 * @see ChatStream
 */
public class ChatSubject extends AbstractEntry
    implements ServiceControlled
{
    /**
     * The serial version UID.  Stating it explicitly is good.
     *
     * @see fortune.FortuneTheme#serialVersionUID
     */
    static final long serialVersionUID =
                            -4036337828321897774L;

    /**
     * The subject of the discussion.
     * @serial
     */
    public String subject;

    /**
     * Public no-arg constructor.  Required for all <CODE>Entry</CODE>
     * objects.
     */
    public ChatSubject() { }

    /**
     * Create a new <CODE>ChatSubject</CODE> with the given subject.
     */
    public ChatSubject(String subject) {
        this.subject = subject;
    }
}
```

348
chatter.Chatter

```
package chatter;

import chat.ChatStream;
import chat.ChatMessage;
import client.StreamReader;
import message.MessageStream;

import java.rmi.RemoteException;

/**
 * A client that talks to a <CODE>ChatStream</CODE>, allowing the user
 * to add messages as well as read them.  The user's login name is used
 * as their name in the chat.  The usage is:
 * <pre>
 *      java [java-options] chatter.Chatter args...
 * </pre>
 * The arguments are the same as those for <CODE>client.StreamReader</CODE>
 * except that you cannot specify the <CODE>-c</CODE> option.  The stream
 * used will be at least a <CODE>chat.ChatStream</CODE> service.
 *
 * @see client.StreamReader
 * @see ChatterThread
 */
public class Chatter extends StreamReader {
    /**
     * Start up the service.
     */
    public static void main(String[] args) throws Exception
    {
        String[] fullargs = new String[args.length + 3];
        fullargs[0] = "-c";
        fullargs[1] = String.valueOf(Integer.MAX_VALUE);
        System.arraycopy(args, 0, fullargs, 2, args.length);
        fullargs[fullargs.length - 1] = "chat.ChatStream";
        Chatter chatter = new Chatter(fullargs);
        chatter.execute();
    }

    /**
     * Create a new <CODE>Chatter</CODE>.  The <CODE>args</CODE> are
     * passed to the superclass.
     */
    private Chatter(String[] args) {
        super(args);
    }

    /**
     * Overrides <CODE>readStream</CODE> to start up a
```

```
     * <CODE>ChatterThread</CODE> when the stream is found.  The
     * <CODE>ChatterThread</CODE> lets the user type messages, while this
     * thread continually reads them.
     */
    public void readStream(MessageStream msgStream)
        throws RemoteException
    {
        ChatStream stream = (ChatStream)msgStream;
        new ChatterThread(stream).start();
        super.readStream(stream);
    }

    /**
     * Print out a message, marking the speaker for easy reading.
     */
    public void printMessage(int msgNum, Object msg) {
        if (!(msg instanceof ChatMessage))
            super.printMessage(msgNum, msg);
        else {
            ChatMessage cmsg = (ChatMessage)msg;
            System.out.println(cmsg.getSpeaker() + ":");
            String[] lines = cmsg.getContent();
            for (int i = 0; i < lines.length; i++) {
                System.out.print("    ");
                System.out.println(lines[i]);
            }
        }
    }
}
```

350
chatter.ChatterThread

```
package chatter;

import chat.ChatStream;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.rmi.RemoteException;
import java.util.ArrayList;
import java.util.List;

/**
 * The thread that <CODE>Chatter</CODE> uses to let the user type
 * new messages.
 */
class ChatterThread extends Thread {
    /**
     * The stream to which we're adding.
     */
    private ChatStream stream;

    /**
     * Create a new <CODE>ChatterThread</CODE> to write to the given stream.
     */
    ChatterThread(ChatStream stream) {
        this.stream = stream;
    }

    /**
     * The thread's workhorse.  Read what the user types and put it into
     * the stream as messages from the user.  The user's name is read from
     * the <CODE>user.name</CODE> property.  A message consists of a series
     * of lines ending in backslash until one that doesn't.
     */
    public void run() {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        String user = System.getProperty("user.name");
        List msg = new ArrayList();
        String[] msgArray = new String[0];
        for (;;) {
            try {
                String line = in.readLine();
                if (line == null)
                    System.exit(0);

                boolean more = line.endsWith("\\");
                if (more) {     // strip trailing backslash
```

```
                int stripped = line.length() - 1;
                line = line.substring(0, stripped);
            }
            msg.add(line);
            if (!more) {
                msgArray = (String[])
                    msg.toArray(new String[msg.size()]);
                stream.add(user, msgArray);
                msg.clear();
            }
        } catch (RemoteException e) {
            System.out.println("RemoteException:retry");
            for (;;) {
                try {
                    Thread.sleep(1000);
                    stream.add(user, msgArray);
                    msg.clear();
                    break;
                } catch (RemoteException re) {
                    continue;       // try again
                } catch (InterruptedException ie) {
                    System.exit(1);
                }
            }
        } catch (IOException e) {
            System.exit(1);
        }
    }
}
}
```

```
package client;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.LookupDiscovery;

import message.MessageStream;

import java.io.BufferedReader;
import java.io.EOFException;
import java.io.InputStreamReader;
import java.io.Reader;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import java.util.StringTokenizer;

/**
 * This class provides a client that reads messages from a
 * <code>MessageStream</code> service.  It's use is:
 * <pre>
 *      java [<i>java-options</i>] client.StreamReader [-c <i>count</i>]
 *               <i>groups|lookupURL</i>
 *               [<i>service-type</i>|<i>attribute</i> ...]
 * </pre>
 * Where the options are:
 * <dl>
 * <dt><i><CODE>java-options</CODE></i>
 * <dd>Options to the Java VM that will run the admin program.  Typically
 * this includes a security policy property.
 * <p>
 * <dt><i><CODE>-c <i>count</i></CODE></i>
 * <dd>The number of messages to print.
 * <p>
 * <dt><i><CODE>groups</CODE></i>|<i><CODE>lookupURL</CODE></i>
 * <dd>Either a comma-separated list of groups in which all the services
 * in the group will be regsitered or a URL to a specific lookup service.
 * <p>
 * <dt><i><CODE>service-type</CODE></i>|<i><CODE>attribute</CODE></i>
```

```
 * <dd>A combination (in any order) of service types and attribute definitions.
 * Service types are specfied as types that the service must be an instance of.
 * Attribute definitions are either <CODE>Entry</CODE> type names,
 * which declare that the service must have an attribute of that type,
 * or <CODE>Entry</CODE> type names with a single <CODE>String</CODE>
 * parameter for the constructor, as in
 * <CODE><i>AttributeType</i>:<i>stringArg</i></CODE>.
 * </dl>
 * <p>The lookups are searched for a <CODE>MessageStream</CODE> that
 * supports any additional service types specfied and that matches all
 * specified attributes.  If one is found, then <CODE><i>count</i></CODE>
 * messages are printed from it.  If a <CODE>RemoteException</CODE>
 * occurs the <CODE>nextMessage</CODE> invocation is retried up to
 * a maximum number of times.
 * <P>
 * This class is designed to be subclassed.  As an example, see
 * <CODE>chatter.Chatter</CODE>.
 *
 * @see message.MessageStream
 * @see chatter.Chatter
 */
public class StreamReader implements DiscoveryListener {
    /**
     * The number of messages to print.
     */
    private int count;

    /**
     * The lookup groups (or an empty array).
     */
    private String[] groups = new String[0];

    /**
     * The lookup URL (or <code>null</code>).
     */
    private String lookupURL;

    /**
     * The stream and attribute types.
     */
    private String[] typeArgs;

    /**
     * The list of unexamined registrars.
     */
    private List registrars = new LinkedList();

    /**
```

354
client.StreamReader

```
     * How long to wait for matches before giving up.
     */
    private final static int MAX_WAIT = 5000;   // five seconds

    /**
     * Maximum number of retries of <code>nextMessage</code>.
     */
    private final static int MAX_RETRIES = 5;

    /**
     * Run the program.
     *
     * @param args      The command-line arguments
     *
     * @see #StreamReader
     */
    public static void main(String[] args) throws Exception
    {
        StreamReader reader = new StreamReader(args);
        reader.execute();
    }

    /**
     * Create a new <code>StreamReader</code> object from the
     * given command line arguments.
     */
    public StreamReader(String[] args) {
        // parse command into the fields count, groups,
        // lookupURL, and typesArgs...
        if (args.length == 0) {
            usage();
            throw new IllegalArgumentException();
        }

        int start;
        if (!args[0].equals("-c")) {
            count = 1;
            start = 0;
        } else {
            count = Integer.parseInt(args[1]);
            start = 2;
        }

        if (args[start].indexOf(':') < 0)
            groups = util.ParseUtil.parseGroups(args[start]);
        else
            lookupURL = args[start];
        typeArgs = new String[args.length - start - 1];
```

```
        System.arraycopy(args, start + 1, typeArgs, 0, typeArgs.length);
    }

    /**
     * Print out a usage message.
     */
    private void usage() {
        System.err.println("usage: java [java-options] " + StreamReader.class +
            " [-c count] groups|lookupURL [service-type|attribute ...]");
    }

    /**
     * Execute the program by consuming messages.
     */
    public void execute() throws Exception {
        if (System.getSecurityManager() == null)
            System.setSecurityManager(new RMISecurityManager());

        // Create lookup discovery object and have it notify us
        LookupDiscovery ld = new LookupDiscovery(groups);
        ld.addDiscoveryListener(this);

        searchDiscovered(); // search discovered lookup services
    }

    /**
     * Search through an discovered lookup services.
     */
    private synchronized void searchDiscovered()
        throws Exception
    {
        ServiceTemplate serviceTmpl = buildTmpl(typeArgs);

        // Loop searching in discovered lookup services
        long end = System.currentTimeMillis() + MAX_WAIT;
        for (;;) {
            // wait until a lookup is discovered or time expires
            long timeLeft = end - System.currentTimeMillis();
            while (timeLeft > 0 && registrars.isEmpty()) {
                wait(timeLeft);
                timeLeft = end - System.currentTimeMillis();
            }
            if (timeLeft <= 0)
                break;

            // Check out the next lookup service
            ServiceRegistrar reg =
                (ServiceRegistrar)registrars.remove(0);
```

```
            try {
                MessageStream stream =
                    (MessageStream)reg.lookup(serviceTmpl);
                if (stream != null) {
                    readStream(stream);
                    return;
                }
            } catch (RemoteException e) {
                continue;               // skip on to next
            }
        }
        System.err.println("No service found");
        System.exit(1);         // nothing happened in time
    }

    /**
     * Build up a <code>ServiceTemplate</code> object for
     * matching based on the types listed on the command line.
     */
    private ServiceTemplate buildTmpl(String[] typeNames)
        throws ClassNotFoundException, IllegalAccessException,
               InstantiationException, NoSuchMethodException,
               InvocationTargetException
    {
        Set typeSet = new HashSet();    // service types
        Set attrSet = new HashSet();    // attribute objects

        // MessageStream class is always required
        typeSet.add(MessageStream.class);

        for (int i = 0; i < typeNames.length; i++) {
            // break the type name up into name and argument
            StringTokenizer tokens =    // breaks up string
                new StringTokenizer(typeNames[i], ":");
            String typeName = tokens.nextToken();
            String arg = null;          // string argument
            if (tokens.hasMoreTokens())
                arg = tokens.nextToken();
            Class cl = Class.forName(typeName);

            // test if it is a type of Entry (an attribute)
            if (Entry.class.isAssignableFrom(cl))
                attrSet.add(attribute(cl, arg));
            else
                typeSet.add(cl);
        }

        // create the arrays from the sets
```

```
        Entry[] attrs = (Entry[])
            attrSet.toArray(new Entry[attrSet.size()]);
        Class[] types = (Class[])
            typeSet.toArray(new Class[typeSet.size()]);

        return new ServiceTemplate(null, types, attrs);
    }

    /**
     * Create an attribute from the class name and optional argument.
     */
    private Object attribute(Class cl, String arg)
        throws IllegalAccessException, InstantiationException,
               NoSuchMethodException, InvocationTargetException
    {
        if (arg == null)
            return cl.newInstance();
        else {
            Class[] argTypes = new Class[] { String.class };
            Constructor ctor = cl.getConstructor(argTypes);
            Object[] args = new Object[] { arg };
            return ctor.newInstance(args);
        }
    }


    /**
     * Notified by <code>LookupDiscovery</code> code when it finds one
     * or more registries.  This implementation adds it to the list of
     * known registries and notifies any waiting thread.
     */
    public synchronized void discovered(DiscoveryEvent ev) {
        ServiceRegistrar[] regs = ev.getRegistrars();
        for (int i = 0; i < regs.length; i++)
            registrars.add(regs[i]);
        notifyAll(); // notify waiters that the list has changed
    }

    /**
     * Notified by <code>LookupDiscovery</code> code when one or more
     * found registries vanishes.  This implementation removes it from
     * the list of known registries.  No notification is necessary
     * since the only waiting threads are waiting for additions, not
     * subtractions.
     */
    public synchronized void discarded(DiscoveryEvent ev) {
        ServiceRegistrar[] regs = ev.getRegistrars();
        for (int i = 0; i < regs.length; i++)
```

```
            registrars.remove(regs[i]);
        notifyAll(); // notify waiters that the list has changed
    }

    /**
     * Read the required number of messages from the given stream.
     */
    public void readStream(MessageStream stream)
        throws RemoteException
    {
        int errorCount = 0;     // # of errors seen this message
        int msgNum = 0;         // # of messages
        while (msgNum < count) {
            try {
                Object msg = stream.nextMessage();
                printMessage(msgNum, msg);
                msgNum++;               // successful read
                errorCount = 0;         // clear error count
            } catch (EOFException e) {
                System.out.println("---EOF---");
                break;
            } catch (RemoteException e) {
                e.printStackTrace();
                if (++errorCount > MAX_RETRIES) {
                    if (msgNum == 0)    // got no messages
                        throw e;
                    else {
                        System.err.println("too many errors");
                        System.exit(1);
                    }
                }
                try {
                    Thread.sleep(1000); // wait 1 second, retry
                } catch (InterruptedException ie) {
                    System.err.println("---Interrupted---");
                    System.exit(1);
                }
            }
        }
    }

    /**
     * Print out the message in a reasonable format.
     */
    public void printMessage(int msgNum, Object msg) {
        if (msgNum > 0) // print separator
            System.out.println("---");
```

```
            System.out.println(msg);
        }
    }
```

```
package fortune;

import message.MessageStream;

import java.io.DataOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.ArrayList;
import java.util.List;

import java.rmi.activation.ActivationException;

/**
 * Administer a <code>FortuneStreamImpl</code>.
 * <pre>
 *      java [<i>java options</i>] fortune.FortuneAdmin <i>database-dir</i>
 * </pre>
 * The database is initialized from the fortune set in the directory's
 * <code>fortunes</code> file, creating a file named <code>pos</code> that
 * contains each fortune's starting position.  The <code>fortunes</code>
 * file must be present.  The <code>pos</code> file, if it exists, will
 * be overwritten.
 *
 * @see FortuneStreamImpl
 */
public class FortuneAdmin {
    /**
     * Run the FortuneAdmin utility.  The class comment describes the
     * possibilities.
     *
     * @param args
     *          The arguments passed on the command line
     *
     * @see FortuneAdmin
     */
    public static void main(String[] args) throws Exception {
        if (args.length != 1)
            usage();
        else
            setup(args[0]);
    }

    /**
     * Set up a directory, reading its <code>fortunes</code> file and
     * creating a correct <code>pos</code> file.
     *
```

```
     * @param dir
     *          The fortune database directory.
     * @throws java.io.IOException
     *          Some error accessing the database files.
     */
    private static void setup(String dir) throws IOException {
        File fortuneFile = new File(dir, "fortunes");
        File posFile = new File(dir, "pos");
        if (posFile.lastModified() > fortuneFile.lastModified()) {
            System.out.println("positions up to date");
            return;
        }

        System.out.print("positions out of date, updating");
        // Open the fortunes file
        RandomAccessFile fortunes =
            new RandomAccessFile(new File(dir, "fortunes"), "r");

        // Remember the start of each fortune
        List positions = new ArrayList();
        positions.add(new Long(0));
        String line;
        while ((line = fortunes.readLine()) != null)
            if (line.startsWith("%%"))
                positions.add(new Long(fortunes.getFilePointer()));
        fortunes.close();

        // Write the pos file
        DataOutputStream pos =
            new DataOutputStream(new FileOutputStream(new File(dir, "pos")));
        int size = positions.size();
        pos.writeLong(size);
        for (int i = 0; i < size; i++)
            pos.writeLong(((Long) positions.get(i)).longValue());
        pos.close();
        System.out.println();
    }

    /**
     * Print out a usage message.
     */
    private static void usage() {
        System.out.println("usage: java [java-options] " + FortuneAdmin.class +
            " database-dir");
    }
}
```

362

fortune.FortuneStream

```
package fortune;

import message.MessageStream;

import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * A <CODE>FortuneStream</CODE> is a <CODE>MessageStream</CODE> whose
 * <CODE>nextMessage</CODE> method returns a random saying on some theme.
 * The theme is returned by the <CODE>getTheme</CODE> method.
 *
 * @see FortuneTheme
 */
interface FortuneStream extends MessageStream, Remote {
    /**
     * Return the theme of the stream.  This is also represented in the
     * lookup service as a <CODE>FortuneTheme</CODE> object.
     */
    String getTheme() throws RemoteException;
}
```

```
package fortune;

import message.MessageStream;
import util.ParseUtil;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceID;

import com.sun.jini.lease.LeaseRenewalManager;
import com.sun.jini.lookup.JoinManager;

import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.rmi.Remote;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;
import java.util.Random;

/**
 * Implement a <code>MessageStream</code> whose
 * <code>nextMessage</code> method returns ''fortune cookie'' selected
 * at random.  The stream is an activatable remote object.  It requires
 * no special proxy because there is no client-side state or smarts --
 * the simple RMI stub works perfectly for this use.
 *
 * <code>FortuneStreamImpl</code> objects are created using the
 * <code>create</code>.  It's only public constructor is designed for
 * use by the activation system itself.  The class
 * <code>FortuneAdmin</code> provides a program that will invoke
 * <code>create</code>.
 *
 * @see FortuneAdmin
 */
public class FortuneStreamImpl implements FortuneStream {
    /**
     * Groups to register with (or an empty array).
     */
    private String[] groups = new String[0];

    /**
     * URL to specific join manager (or <CODE>null</CODE>).
```

```
     */
    private String lookupURL;

    /**
     * The directory we work in.
     */
    private String dir;

    /**
     * The theme of this stream.
     */
    private String theme;

    /**
     * The random number generator we use.
     */
    private Random random = new Random();

    /**
     * The positions of the start of each fortune in the file.
     */
    private long[] positions;

    /**
     * The file that contains the fortunes.
     */
    private RandomAccessFile fortunes;

    /**
     * The join manager does most work required of services in Jini systems.
     */
    private JoinManager joinMgr;

    /**
     * @param args      The command line arguments.
     */
    public static void main(String[] args) throws Exception
    {
        FortuneStreamImpl f = new FortuneStreamImpl(args);
        f.execute();
    }

    /**
     * Create a stream that reads from the given directory.
     *
     * @param dir       The directory name.
     */
    private FortuneStreamImpl(String args[])
```

```
    throws IOException
{
    // Set the groups, lookupURL, dir, and theme
    // fields...
    if (args.length != 3) {
        usage();
        throw new IllegalArgumentException();
    }
    if (args[0].indexOf(':') < 0)
        groups = util.ParseUtil.parseGroups(args[0]);
    else
        lookupURL = args[0];
    dir = args[1];
    theme = args[2];
}

/**
 * Print out a usage message.
 */
private void usage() {
    System.err.println("usage: java " + FortuneStreamImpl.class +
        " groups|lookupURL database-dir theme");
}

/**
 * Export this service as a UnicastRemoteObject for debugging purposes.
 *
 * @see #main
 */
private void execute() throws IOException {
    System.setSecurityManager(new RMISecurityManager());
    UnicastRemoteObject.exportObject(this);

    // Set up the fortune database
    setupFortunes();

    // set our FortuneTheme attribute
    FortuneTheme themeAttr = new FortuneTheme(theme);
    Entry[] initialAttrs = new Entry[] { themeAttr };

    LookupLocator[] locators = null;
    if (lookupURL != null) {
        LookupLocator loc = new LookupLocator(lookupURL);
        locators = new LookupLocator[] { loc };
    }
    joinMgr = new JoinManager(this, initialAttrs,
        groups, locators, null, null);
}
```

```
/**
 * Called when the database needs to be set up.  This can be called
 * multiple times, for example if the database has been modified while
 * the service is running.
 *
 * @throws java.io.IOException
 *          Some problem occurred accessing the database files.
 */
private synchronized void setupFortunes() throws IOException {
    // Read in the position of each fortune
    File posFile = new File(dir, "pos");
    DataInputStream in = new DataInputStream(
        new BufferedInputStream(new FileInputStream(posFile)));
    int count = (int) in.readLong();
    positions = new long[count];
    for (int i = 0; i < positions.length; i++)
        positions[i] = in.readLong();
    in.close();

    // Close the fortune file if previously opened
    if (fortunes != null)
        fortunes.close();
    // Open up the fortune file
    fortunes = new RandomAccessFile(new File(dir, "fortunes"), "r");
}

/**
 * Return the next message from the stream.  Since messages are
 * selected at random, any message is as good as any other and so
 * this is idempotent by contract: there will be no violation of
 * the contract if the client calls it a second time after getting
 * a <code>RemoteException</code>.  The <CODE>Object</CODE> returned
 * is a <CODE>String</CODE> with embeded newlines, but no trailing
 * newline.
 *
 * @throws java.io.EOFException
 *          The database has been corrupted -- no more messages
 *          from this stream.
 */
public synchronized Object nextMessage() throws EOFException {
    try {
        int which = random.nextInt(positions.length);
        fortunes.seek(positions[which]);
        StringBuffer buf = new StringBuffer();
        String line;
        while ((line = fortunes.readLine()) != null && !line.equals("%%")) {
            if (buf.length() > 0)
```

```
                buf.append('\n');
            buf.append(line);
        }
        return buf.toString();
    } catch (IOException e) {
        throw new EOFException("directory not available:" + e.getMessage());
    }
}

// inherit doc comment from interface
public String getTheme() {
    return theme;
}
}
```

368
fortune.FortuneTheme

```
package fortune;

import net.jini.entry.AbstractEntry;
import net.jini.lookup.entry.ServiceControlled;

/**
 * This class is used as an attribute in the lookup system to tell
 * the user what theme of fortunes a stream generates.
 */
public class FortuneTheme extends AbstractEntry
    implements ServiceControlled
{
    /**
     * The serial version UID.  Stating it explicitly allows future
     * evolution with a guaranteed consistency of the UID itself.  It
     * is also more efficient since otherwise the UID must be calculated
     * when the class is serialized.  A good specification should include
     * the serial version UID of each class.
     */
    static final long serialVersionUID =
                            -1696813496901296488L;

    /**
     * The theme of this collection of fortunes.
     *
     * @see fortune.FortuneStream#getTheme
     * @serial
     */
    public String theme;

    /**
     * Public no-arg constructor.  Required for all <CODE>Entry</CODE>
     * objects.
     */
    public FortuneTheme() { }

    /**
     * Create a new <CODE>FortuneTheme</CODE> with the given theme.
     */
    public FortuneTheme(String theme) {
        this.theme = theme;
    }
}
```

```
package message;

import java.io.EOFException;
import java.rmi.RemoteException;

/**
 * This interface defines a message stream service.  Successive
 * invocations of <code>nextMessage</code> return the next message in
 * turn.  Subinterfaces may add methods to rewind the stream or
 * otherwise move around within the stream if appropriate.
 */
public interface MessageStream {
    /**
     * Return the next message in the stream.  Each message is an
     * object whose default method of display is a string returned by
     * its <CODE>toString</CODE> method.  This method is idempotent: if
     * the client receives a <code>RemoteException</code>, the next
     * invocation from the client should return an equivalent message.
     * A service may specify which kinds of messages will be returned.
     *
     * @returns The next message as an <CODE>Object</CODE>.
     * @throws  java.io.EOFException
     *          The end of the stream has been reached.
     * @throws  java.rmi.RemoteException
     *          A remote exception has occurred.
     */
    Object nextMessage()
        throws EOFException, RemoteException;
}
```

370
util.ParseUtil

```
package util;

import java.util.HashSet;
import java.util.Set;
import java.util.StringTokenizer;

/**
 * This class holds the static <CODE>parseGroups</CODE> method.
 */
public class ParseUtil {
    /**
     * Break up a comma-separated list of groups into an array of strings.
     *
     * @param groupDesc A comma-separated list of groups.
     * @returns         An array of strings (empty if none were specified).
     */
    public static String[] parseGroups(String groupDesc) {
        if (groupDesc.equals(""))
            return new String[] {""};
        Set groups = new HashSet();
        StringTokenizer strs = new StringTokenizer(groupDesc, ", \t\n");
        while (strs.hasMoreTokens())
            groups.add(strs.nextToken());
        return (String[]) groups.toArray(new String[groups.size()]);
    }
}
```

Index

It's a d–mn poor mind that can only think of one way to spell a word!
—Andrew Jackson

A

aborted votes, 187
`ABSOLUTE` constant, 143
`AbstractEntry` class
  `equals`, `hashCode`, and `toString` functions of, 133–134
  serialized forms of, 134
access control list, 67
ACID properties
  atomicity, 188, 270
  consistency, 188, 270
  durability, 188, 270
  isolation, 188, 270
  in JavaSpaces, 270–271
activation, 43–45
  activatable classes and objects, 43–44
  in chat server, 48
  definition of, 293
  lazy activation and, 297
activation constructor, 44
activation descriptor, 293
activation group
  creating, 45
  definition of, 293
  function of, 43–44
activation system, 43–45
activator, 293
active object, 294
`Address` class, 247–248, 249
algorithms, distributed, 254–256
`ALL_GROUPS` constants, 115
ancestor transactions
  definition of, 294
  execution of, 212
`ANY` constant, 143
architecture specification, 61–82
  components of, 68–71
  environmental assumptions of, 63
  goals of, 61–62
  infrastructure component of, 61, 68
  key concepts of, 65–67
  printing service example of, 77–80
  programming model component of, 62, 68
  service architecture in, 72–76
  services component of, 62, 68
atomicity, 188, 270
attribute classes, 243–250
  adding comments with, 246–247
  getting service information with, 243–245
  getting status information with, 248–249
  modifying, 243
  naming a service with, 246
  physical location and, 247–248
  serialized forms of, 249–250
attributes. *See also* lookup attribute schema specification
  definition of, 294
  `FortuneStream` service and, 31
  human access to, 234–235, 237
  interoperability with JavaBeans, 235
  localization of, 235
  lookup services and, 29, 101, 217–218
  matching, 11
  modifying, 103, 235, 239
  overview of, 10–11
  registering and querying based on, 233
  service items and, 218–219
  single views of, 237
  specifying, 20, 73–74
  static quality of, 234
  using as service properties, 11

using names as, 14

B

bridging calls, 105–106
buildTmpl method, 24–25

C

CannotNestException class, 212
channel, 294
chat room service. ***See*** **ChatStream**
chat server, 43–50
activation and, 43–45, 48
classes and methods of, 45–50
implementation of, 43
improvements to, 51–52
registration in, 48
ChatMessage, 327–329
ChatProxy, 330–331
ChatServer, 332–333
ChatServerAdmin, 45, 49, 334–337
ChatServerImpl, 43, 45, 338–343
ChatSpeaker, 344–345
ChatStore object, 49
ChatStream, 37–55
chat server and, 43–50, 51–52
clients for, 52–55
complete code for, 345–346
creating, 41–43
getSubject and, 40, 43
lastIndex field and, 43
nextInLine method and, 41–43, 50–51
nextMessage method and, 38, 41
overview of, 37–41
public service interface for, 39
toStringMethod and, 40
ChatSubject, 347
Chatter, 52–55, 348–349
ChatterThread, 350–351
Ciardi, John
quotation, 57
classes
Comment class, 246–247, 249
Constants class, 122–123
DiscoveryEvent class, 116
entry classes, 128–129, 239–241
EntryBeans class, 242
event interfaces and, 161–162, 163–168
EventRegistration class, 162, 168
IncomingMulticastAnnouncement class, 122
IncomingMulticastRequest class, 120–121
IncomingUnicastRequest class, 123–124
IncomingUnicastResponse class, 124–125
JavaBeans and, 241–242
Location class, 247, 249
LookupDiscovery class, 113–115
LookupLocator class, 107–109
Name class, 246
packages and, 16–17
RemoteEvent class, 162, 164–165
ServerTransaction class, 209–212
ServiceInfo attribute class, 243
ServiceMatches class, 224
ServiceType class, 245
Status class, 250
StatusBean class, 249
StatusType class, 249
TransactionFactory class, 187, 209
clients
ChatStream service and, 52–55
completing transactions and, 197–198
definition of, 234
locating services and, 73
service interfaces and, 74–75
specifying attributes for, 73–74
clients, writing, 19–28
buildTmpl method and, 24–25
creating search template for, 20
execute method and, 22
LookupDiscovery and, 22–23
main method and, 21
MessageStream interface and, 19–28
readStream method and, 26–27
searchDiscovered method and, 22–24
setting security manager for, 22
specifying attributes for, 20
users specifications for, 20–21
code
downloading, 7, 63
Java application environment and, 62
notes on, 16
passing with RMI, 66
code, examples, 327–370
ChatMessage, 327–329
ChatProxy, 330–331

ChatServer, 332–333
ChatServerAdmin, 334–337
ChatServerImpl, 338–343
ChatSpeaker, 344–345
ChatStream, 345–346
ChatSubject, 347
Chatter, 348–349
ChatterThread, 350–351
FortuneAdmin, 360–362
FortuneStreamImpl, 363–367
FortuneTheme, 368
MessageStream, 369
ParseUtil, 370
StreamReader, 352–359
collaboration
quotation, 385
Comment class, 246–247, 249
commit points, 204
Common Object Request Broker Architecture (CORBA), 288–289, 308
com.sun.jini, 16–18
com.sun.jini.lookup.JoinManager, 30
concurrency problems, 316–318
connection, 294
consistency, ACID property, 188, 270
constants, 115
ABSOLUTE, 143
ALL_GROUPS, 115
ANY constant, 143
DURATION constant, 143
FOREVER constant, 143
lease interface and, 143
NO_GROUPS, 115
protocol utilities and, 122–123
CORBA (Common Object Request Broker Architecture), 288–289
core packages, 16–17
crash recovery, 204–205
commit points and, 204
roll decisions and, 205
createGroup, 47

D

data
Java application environment and, 62
passing with RMI, 66
databases, 257
delegation event model, 179
designing lookup services, 234–235. ***See also*** **lookup services**
automated matching and, 234
changing attributes and, 235
human understanding and, 234–235
JavaBeans and, 235
static nature of attributes and, 234
de Saint-Exupery, Antoine
quotation, *xix*
device architecture specification, 277–289
combining hardware and software applications and, 277–278
devices connected via IIOP streams, 288–289
devices using specialized virtual machines, 283–284
devices with resident JVMs, 281–283
devices with shared virtual machines (network option), 286–289
devices with shared virtual machines (physical option0, 284–286
introduction to, 277–279
Java programming language and, 278
participating in discovery protocol and, 278
registering with lookup services and, 278
requirements of, 278–279
devices connected via IIOP streams, 288–289
advantages and disadvantages of, 289
directly interpreting IIOP streams and, 289
requirements of, 289
using CORBA ORBs and, 288
devices using specialized virtual machines
advantages and disadvantages of, 283–284
simplifying JVM structure for, 284
devices with resident Java Virtual Machines, 281–283
costs of, 283
design illustration of, 282
functionality of, 282
Java programming language and, 283
utilizing RMI and, 283
devices with shared virtual machines (network option), 286–289
advantages and disadvantages of, 288
building gateways between devices with, 288
complexity of individual devices in, 288
design illustration of, 287
network proxy for, 286
protocols needed for, 287
requirements for, 287

devices with shared virtual machines (physical option)
co-location of JVM and, 284
costs and savings with, 286
design illustration of, 285
"device bay" functionality of, 284–285
directory service, 13–14
discovering entity, 83, 294
discovery and join specification, 228–229. ***See also*** **discovery protocols; join protocols**
discovery protocols, 85–100
definition of, 5
device architecture specification and, 278
finding lookup services with, 9–10, 66, 72–75
in Jini infrastructure, 69
multicast announcement protocol, 85, 87, 95–97
multicast request protocol, 85, 86–87, 89–95
network issues of, 105–109
registering printing services and, 77
unicast discovery protocol, 85, 88
discovery request service, 294
discovery response service, 295
discovery utilities specification. ***See*** **multicast discovery utilities; protocol utilities; utilities specification**
`DiscoveryEvent` class
`LookupDiscovery` and, 113
methods of, 116
serialized forms of, 118
`DiscoveryListener` interface, 22, 114, 116–117
`DiscoveryPermission`, 117–118
distributed algorithms
design of, 253
JavaSpaces and, 254–256
distributed computing. ***See also*** **distributed vs. local computing**
compared with centralized networks, 62
dealing with out of date information in, 138–139
dealing with partial failure problems in, 138
definition of, 308
difficulties of, 253, 307-325
Java application environment and, 62
Jini system and, 61
distributed event adapters, 171–177
notification composition and, 176–177
notification filters and, 173–175
notification mailboxes and, 175–176
store-and-forward agents and, 171–173
distributed event model, 179, 180
distributed event specification, 155–182. ***See also*** **events**
distributed event adapters for, 171–177
goals and requirements for, 156–157
integrating with JavaBeans, 179–182
interfaces for, 159–170
overview of, 155–156
registration methods in, 267
distributed leasing specification, 137–153. ***See also*** **leasing**
distributed systems and, 137–139
goals and requirements of, 140
interfaces for, 141–148
supporting classes for, 149–152
distributed notification
compared with local notification, 179
third-party objects for, 179
distributed persistence, 254
distributed systems. ***See*** **distributed computing**
distributed vs. local computing, 307–326
historical view of, 311–312
introduction to, 307–308
latency problems in, 312–314
lessons from NFS, 320–322
memory access problems in, 314–315
middle ground situations, 324–325
partial failure and concurrency problems in, 316–318
quality of service myth and, 318–320
taking the differences into account, 322–324
unified objects vision for, 308–310
djinns
definition of, 295
handling responses from multiple djinns, 95
host requirements for, 84
Jini system and, 83
DNS names, 108
durability, ACID property, 188, 270
`DURATION` constant, 143
dynamic class loading, 295
dynamic stub loading, 295

E

encapsulation
object-oriented programming and, 6
proxy objects and, 8
RMI and, 66
endpoint, 295
entities
definition of, 84
in event interfaces, 159–161
entries. ***See also*** **attributes**
aggregating attributes with, 233
definition of, 296
`FortuneTheme` and, 31
JavaSpaces services and, 261
overview of, 128–129
semantics of, 11
entry classes, 128–129, 239–241
entry specification, 127–131
constructors for, 128
entries defined, 127
fields and, 128
Jini utility for, 132
operations of, 127
serialized forms of, 131
serializing `entry` objects, 128–129
templates and matching in, 127, 131
types and, 128
`UnusableEntryException` and, 129–130
entry utilities specification, 133–135
`EntryBeans` class, 242
environmental prerequisites, Jini systems
Java programming language compliance, 63
memory and processing capacity, 63
reasonable network latency, 63
`equals`
`AbstractEntry` class, 133–134
`LookupLocator` class, 107
event generators, 160, 296
event interfaces, 159–170
entities involved in, 159–161
functions of, 159
interfaces and classes of, 161–168
leasing and, 169–170
sequence numbers and, 169–170
serialized forms of, 170
transactions and, 169–170
event listener, 296
event models, 179–180
event registration, 169
`EventGenerator` interface, 166–167
`EventRegistration` class, 162, 168
events. ***See also*** **distributed event specification**
definition of, 159, 296
event generator and, 160
local events, 298
registration of, 159, 160
remote events, 160
support for distributed events and, 67
types of, 159
exception types, 145–147
exclusive leases, 67
`execute` method, 22
exporting, 296
`exportObject`, 47
extended packages, 16–17

F

faulting remote reference, 296–297
federated groups, 65
fetch operations, 127, 129
"flow of objects" approach, JavaSpaces, 254–256
`FOREVER` constant, 143
fortune cookie service. ***See*** **`FortuneStream` service**
`FortuneAdmin`, 33, 360–362
`FortuneStream` service, 30–36
administration program for, 33
attributes of, 31
creating, 32–33
entry and, 31
implementation design for, 32
overview of, 30–31
running, 34–36
security options for, 34
`FortuneStreamImp`, 32–33, 363–367
`FortuneTheme`, 31, 35, 368
Fuller, R. Buckminster
quotation, 29

G

gateways, devices, 288
`getHost` method, 107
`getPort` method, 107
`getRegistrar` method, 108

getSubject, 43
getSubject method, 40
getTheme method, 30
glossary, 293–307
goals, Jini system
easy and portable network access, 62
erasing hardware/software distinctions, 4
plug-and-work functionality, 4
service-based architecture, 4
sharing resources, 62
simple network administration, 62
simplicity and reusable code, 4–5
spontaneous networking, 4
groups
chat server and, 47
discovery process and, 85–86
djinns and, 84
join protocols and, 29, 102, 103
limiting scope with, 13
lookup services and, 12–13
modifying, 115–116
object groups and, 75
public groups and, 101

H

hard mounts, 321
hardware
device architecture specification and, 277–278
implementing within Jini architecture, 281
hashCode, 133–134
"here I am" messages, 29
host requirements, 84
hosts, 83, 296–297

I

idempotent methods, 38, 297
IDL (Interface Definition Language), 322
IIOP (Internet Inter-Operability Protocol), 288–289
IncomingMulticastAnnouncement class, 122
IncomingMulticastRequest class, 120–121
IncomingUnicastRequest class, 123–124
IncomingUnicastResponse class, 124–125
indeterminacy, 316
inferior transactions, 297
infrastructure
discovery and join protocols in, 69
distributed security system in, 69
Jini architecture and, 61, 68
lookup service in, 69
interface definition languages, 322
interfaces, 228
client/server interactions with, 74–75
for core, standard, and extended packages, 16–17
designing for distributed systems, 317–318
event interfaces, 161–162, 163–168
for event specification, 159–170
finder-style visual interfaces, 95
Java programming language and, 69–70
for JavaBeans, 241–242
service protocols as, 66
for services, 71
for store-and-forward agents, 173
InternalSpaceException, 263–264, 268
Internet Inter-Operability Protocol (IIOP), 288–289
interposition, 281
IP addresses
assigning to hosts, 84
URL syntax and, 108
IP broadcast protocols, 106
IP multicast protocols, 106
IP networks, 84
isolation, ACID property, 188, 270
item matching, 223–224

J

Jackson, Andrew
quotation, 371
Java application environment, 62
Java Development Kit (JDK), 118
Java Foundation Classes (JFC), 179
Java objects, 5
Java programming language
device architecture specification and, 278, 283
Jini system and, 63, 69–70
security options of, 34
service types and, 73
using for matching, 10
Java Remote Method Invocation (RMI). ***See*** **Remote Method Invocation (RMI)**

Java Virtual Machines (JVMs)
devices with full versions of, 281–283
devices with specialized versions of, 283–284
hosts and, 83
in Jini systems, 63
properties of, 7
RMI system and, 279
sharing between devices, 284–288
JavaBeans component event model, 179–182
characteristics of, 180
distributed event model and, 180–182
JavaBeans components
displaying and modifying attributes with, 239
supporting interfaces and classes with, 241–242
using with entry classes, 239–241
JavaBeans specification, 237
`JavaSpace` interface, 262–263
JavaSpaces application model, 253–256
compared with Linda systems, 258–259
design issues of, 258–259
distributed algorithms as flow objects in, 254–256
distributed persistence in, 254
goals and requirements of, 259–260
JavaSpaces specification, 253–274
benefits of, 256–257
compared with databases, 257
dependency on other specifications, 260
distributed object persistence in, 257
entries and, 261
further reading on, 273–274
handling concurrent access with, 256
introduction to, 253–260
methods of, 263–268
notify operation of, 261
order of operations in, 268
read operation of, 261
reliable distributed storage in, 256
replication of, 259
services of, 71
take operation of, 261
transactions and, 269–271
write operation of, 261
Jini system, introduction, 3–18
architectural features of, 5–7
flexibility of, 15
goals of, 4–5
lookup service in, 9–14
overview of, 3–4
package structure in, 16–18
properties of, 7
robust nature of, 14–15
value of a proxy in, 7–8
Johnson, S.C.
quotation, 19
join configuration, 29
join protocols, 101–109
attribute modification and, 103
definition of, 297
initial discovery and registration with, 102
in Jini infrastructure, 69
joining or leaving groups with, 102, 103
joining with lookup services, 66, 72–75
lease renewal and handling communication with, 102
making changes and performing updates with, 103
order of discovery and, 102
registering and unregistering with lookup services, 103
registering printing service with, 77–78
joining entities, 83, 297
`JoinManager`
`FortuneStream` service example and, 35–36
managing lookup membership with, 30

L

`lastIndex` field, 43
latency problems, 312–314
efficiency disparities due to, 312–313
masking with increased speed, 313
lazy activation, 297
lease grantors, 298
lease holders, 298
`Lease` interfaces, 141–148
constants used with, 143
exceptions and, 145–147
methods of, 143–145
operations of, 142–147
overview of, 137
serialized forms of, 148
time grants for, 147–148
`LeaseDeniedException`, 145, 148
`LeaseException`, 146, 148
`LeaseMapException`, 148
`LeaseRenew` class, 149–151
`LeaseRenewService` interface, 151–152

leases. ***See also*** **distributed leasing specification**
accessing services and, 67
benefits of, 12
characteristics of, 141–142
definition of, 297–298
event registration transactions and, 169
exclusive or non-exclusive, 67
JavaSpaces and, 254
lookup citizenship and, 29
for printing services, 78
registering services and, 11
renewing, 102, 149–152
store-and-forward agents and, 173
Linda systems, 258–259
live references, 298
local area networks (LANs), 89, 93
local computing, 308. ***See also*** **distributed vs. local computing**
local event model, 179
local events, 298
local notification, 179
local object invocation. ***See*** **local computing**
local objects
with remote characteristics, 324
in unified object system, 308
`Location` class, 247, 249
lookup attribute schema specification, 233–250. ***See also*** **attributes**
attribute standards in, 219
dependency on other specifications, 235
generic attribute classes and, 243–250
human access to attributes and, 237–238
introduction to, 233–235
JavaBeans components and, 239–242
lookup citizenship, 29–30
lookup discovery protocol, 298
lookup protocols
invoking services with, 72–75
in Jini infrastructure, 69
lookup service model, 217–218
administrative uses of, 218
imposing hierarchical views on, 218
service items in, 217
lookup service specification, 217–230
dependency on other specifications, 219
introduction to, 212–219
`ServiceRegistrar` and, 225–229
`ServiceRegistration` and, 229–230
types defined in, 221–224
lookup services, ***vii,*** **371.** ***See also*** **services**
attributes of, 10–11, 217–218, 218–219
available services list in, 9
compared to directory services, 13–14
definition of, 5, 299
design issues of, 234
device architecture specification and, 278
discovery process and, 9–10
functions of, 217
good standing of, 29–30
groups and, 12–13
Java languages rules for, 10
matching services with, 66, 218, 223–224
membership management in, 11–12
multicast request protocol and, 89
RMI interface and, 279
`LookupDiscovery` class, 113–115
methods of, 114–115
registering with, 79
use of, 113
writing a client and, 22–23
`LookupLocator` class, 107–109
as interface for unicast discovery, 107
methods of, 107–108
specifying lookup services by URL with, 20

M

`main` method
writing a client and, 21
writing a service and, 34–35
managers
commit point and, 204
completing a transaction, 202–204
roll decision of, 205
marshall streams, 299
marshalled objects, 299
`MarshalledObject`, 47
match operations
of entries, 127, 131
item matching and, 223–224
Java programming language and, 10
of lookup services, 66, 218, 223–224
membership management
with `JoinManager` utility, 30
leases and, 11–12
in lookup services, 11–12

memory access problems, 314–315
illusion of unified programming model and, 315
transparency and, 314
MessageStream, 37–55
complete code for, 369
FortuneStream example of, 30–36
writing a client and, 19–20
method-invocation-style design, 255
methods
of DiscoveryEvent class, 116
execute method, 22
getHost method, 107
getPort method, 107
getRegistrar method, 108
getSubject method, 40
getTheme method, 30
of JavaSpaces specification, 263–268
of Lease interfaces, 143–145
of LookupDiscovery class, 114–115
of LookupLocator class, 107–108
nextInLine method, 41–43
nextMessage method, 36
Register method, 78
searchDiscovered method, 22–24
of ServiceRegistrar interface, 225–227
of ServiceRegistration, 229–230
toStringMethod, 40
multicast announcement protocol
announcing service availability with, 95
definition of, 85
discovery process in, 87
steps in process of, 97
multicast announcement service, 95–97
address for, 106
fields of, 96
multicast UDP and, 95
packet requirements of, 96
size of, 97
multicast discovery utilities, 113–118
DiscoveryEvent class, 116
DiscoveryListener interface, 116–117
LookupDiscovery class, 113–115
modifying groups with, 115–116
security methods of, 117–118
serialized form of, 118
useful constants of, 115
multicast network protocols. ***See*** **network protocols**
multicast request client, 89–90
multicast request packet format
contents of, 91–92
size of, 92–93
specifications of, 91
variables in, 92
multicast request protocol, 89–95
definition of, 85
discovering lookup services with, 89
discovery process in, 86–87
handling responses from multiple djinns, 95
multicast request service and, 90–91
multicast response service and, 93
net.jini.core.lookup.ServiceRegistrar and, 86
protocol participants in, 89–90
request packet format for, 91–93
steps taken by the discovering entity, 93–94
steps taken by the multicast request server, 94–95
multicast request server, 90, 94–95
multicast response client, 90
multicast response server, 90
multicast response service, 93
multicast UDP, 84, 95

N

Name class, 246, 249
naming service, 13–14
NestableServerTransaction class, 209–212
NestableTransactionManager
starting a nested transaction and, 193–194
two-phase commit and, 191
net.core.entry.Entry interface, 239
net.jini, 16–17
net.jini.core, 16–17
net.jini.core.entry, 219
net.jini.core.entry.Entry, 253
net.jini.core.entry.UnusableEntryException, 129–130
net.jini.core.event, 161
net.jini.core.lease, 142
net.jini.core.lookup, 221
net.jini.discovery.LookupDiscovery. ***See*** **LookupDiscovery class**
net.jini.core.lookup.ServiceRegistrar
multicast request protocol and, 86
unicast discovery protocol and, 88

net.jini.core.transaction, 269
net.jini.discovery.DiscoveryEvent, 116
net.jini.discovery.DiscoveryListener, 116–117
net.jini.entry.AbsrtactEntry, 133
net.jini.spaace.JavaSpace, 262–264
network access, 62
network administration, 62
network protocols, 105–107
address and port mappings for TCP and UDP, 106
bridging calls with, 105–106
limiting the scope of multicasts in, 106
multicast IP and, 106
packet size limitations of, 105
transport properties of, 105
networking
centralized, 62
distributed computing and, 62
IP networks and, 84
in Jini systems, 4
nextInLine method, 41–43, 50–51
nextMessage method, 36, 38, 41
NFS, 320–322
limitation on scalability in, 321–322
stateless protocol of, 321
use of soft and hard mounts in, 321
NO_GROUPS constant, 115
non-exclusive leases, 67
notification composition, 176–177
notification filters, 173–175
definition of, 299
functions of, 173–174
notification multiplexing with, 174–175
notification, local and distributed, 179
notification mailboxes, 175–176
definition of, 299–230
delivery to, 175
purpose of, 175
use of, 175–176
notification multiplexing, 174
notify, 261, 266–267
in transactions, 270

O

object groups, 75
Object Linking and Embedding (OLE), 309
object-oriented programming, 6. ***See also*** **distributed computing**
Object Request Broker (ORB), 288–289
object serialization, 300
orphans, 213
OutgoingMulticastAnnouncement class, 121
OutgoingMulticastRequest class, 119–120
OutgoingUnicastRequest class, 123
OutgoingUnicastResponse class, 124

P

package structure
core, standard, and extended categories of, 16–17
in Jini systems, 16–18
lookup packages of, 16–17
packets
in multicast announcement service, 96
in multicast request protocol, 91–92
size limitations on, 105
ParseUtil, 370
partial failure problems, 138, 316–318
participants
commit point and, 204
completing a transaction and, 199–201
roll decision of, 205
passive objects, 300
peer lookup, 75
permissions, 117
persistence of information, 139
plug-and-work, 4
port mapping, 106
prepared votes, 187
principal, security, 67
printing service, 71
example using, 77–80
printing with, 78–80
registering, 77–78
programming model, 69–71
ability to move code in, 69
combining with infrastructure and services, 71
for distributed services, 62
interfaces in, 69–70
as segment of Jini architecture, 62, 68
properties, Jini architecture, 7
protocol stack requirements, 84

protocol utilities
`Constants` class, 122–123
`IncomingMulticastAnnouncement` class, 122
`IncomingMulticastRequest` class, 120–121
`IncomingUnicastRequest` class, 123–124
`IncomingUnicastResponse` class, 124–125
`OutgoingMulticastAnnouncement` class, 121
`OutgoingMulticastRequest` class, 119–120
`OutgoingUnicastRequest` class, 123
`OutgoingUnicastResponse` class, 124
protocols
discovery protocols. *See* discovery protocols
IP broadcast protocols, 106
IP multicast protocols, 106
join protocols. *See* join protocols
lookup discovery protocols, 298
lookup protocols, 69, 72–75
multicast announcement protocols, 95
multicast request protocols, 89–95
network protocols, 105–107
for proxies, 6
roles of, 89
service protocols, 66
two-phase commit protocol, 191–206
unicast discovery protocol, 85
proxy objects
clustered devices and, 286
defining service type with, 7–8
for devices, 281
encapsulation and, 6, 8
functioning as downloadable drivers, 7
as Java object, 5
protocol definition and, 6
representing devices to Jini system with, 63
smart proxies and, 75
value of, 7–8
writing, 8
public groups
lookup services in, 86
services and, 101
pure transactions, 300

Q

Quotations
Ciardi, John, 57
collaboration, 385
de Saint-Exupery, Antoine, *xix*
Fuller, R. Buckminster, 29
Jackson, Andrew, 371
Twain, Mark, 3

R

`read`, 261, 264–265
in transactions, 269–270
`readIfExists`, 264–265
`readStream` method, 26–27
reference lists, 300
`Register` method, 78
registration
chat server and, 48
of events, 159, 160, 169
join protocols and, 102
`notify` and, 266–267
registering and unregistering with lookup services, 103
of services, 29, 77–78, 226
registry, 300
remote event generators, 301
remote event listeners, 160, 301. ***See also*** **events**
remote events, 160, 300–301. ***See also*** **events**
remote interfaces, 301
Remote Method Invocation (RMI)
communicating between services with, 66
definition of, 301
downloading, 84
encapsulation and, 66
lookup services and, 279
using with devices, 278, 283
remote object invocation. ***See*** **distributed computing**
remote objects
definition of, 301
unified objects vision and, 308
remote procedure calls (RPC), 308
remote reference layers (RRL), 301
`RemoteEvent` class, 162, 164–165
`RemoteEventListener`
enabling features for third-party entities, 176–177
event interface, 161–162
implementation of, 163–164
request format, 99

resource allocation, 140
response format, 100
rmic, 301–302
rmid, 302
rmiregistry, 302
`RMISecurityManger`, 22
roll back transaction, 187
roll decisions, 205
roll forward transaction, 187

S

safety and security, 7
search templates, 20
`searchDiscovered` method, 22–24
security
  access control list and, 67
  distributed security system and, 69
  in Java application environment, 62
  JDK model for, 118
  principal and, 67
  safety and, 7
security manager, 22
security methods, 117–118
security policy file
  for multicast discovery, 117
  settings for, 20
semantic transactions, 302
sequence numbers, 169, 228
servers, 41
`ServerTransaction` class, 209–212
service architecture, 72–76
  discovery protocol and, 72–75
  join protocol and, 72–75
  lookup protocol and, 72–75
  service implementation, 75–76
service ID, 30
service implementation, 75–76
service items
  attributes of, 218–219
  definition of, 303
  in lookup service model, 217
service protocols, 66
service registrars, 303
service types, 73
  defined by proxy objects, 7–8
  serialized form of, 230
`ServiceEvent` class, 224, 230
`ServiceID`, 221–222, 228, 230
`ServiceInfo` attribute class, 243
`ServiceItem`, 230
`ServiceItem` and, 222–223
`ServiceMatches`, 230
`ServiceMatches` class, 224
`ServiceRegistrar`, 225–229
  function of, 225
  methods of, 225–227
  objects and, 23
  sequence numbers and, 228
`ServiceRegistration`
  manipulating service items with, 229
  methods of, 229–230
services. *See also* lookup services
  availability of, 62
  communicating between, 66
  communication problems and, 102
  compared with servers, 41
  definition of, 65, 234, 302
  examples of, 71
  interfaces of, 71
  in Jini system, 62, 68
  maintaining, 101
  multicast announcement protocol and, 95
  object nature of, 71
  registering, 226
  `ServiceID` and, 228
  sharing access to, 65
services, writing, 29–55
  `ChatStream` service example, 37–55
  `FortuneStream` service example, 30–36
  `JoinManager` utility and, 30
  lookup citizenship and, 29–30
`ServiceTemplate`, 223–224, 230
`ServiceType` class, 245
`setupFortunes`, 35
sharing resources, 62
skeletons, 303
smart proxies, 75
`snapshot`, 265–266
soft mounts, 321
software, 277–278
standard packages, 16–17
`Status` class, 248–249, 250
`StatusBean` class, 249, 250
`StatusType` class, 249, 250
store-and-forward agents, 171–173
  definition of, 303

implementation of, 173
interface for, 173
issuing leases with, 173
notifications and, 171–173
reliability of, 171
store operations, 127
StreamReader, 352–359
stubs, 303

T

take, 261, 265
in transactions, 270
takeIfExists, 265
TCP, 84, 98
templates
definition of, 304
item matching with, 223–224
search templates, 20
ServiceTemplate, 223–224, 230
using for exact matches with entries, 127, 131
time grants, 147–148
time-to-live (TTL) field, 106
to String, 133–134
toStringMethod, 40
transaction clients, 304
transaction managers, 71, 304
transaction participants, 304. ***See also*** **participants**
transaction specification, 185–214, 269
default transaction semantics and, 207–214
dependency on other specifications, 190
introduction to, 185–190
two-phase commit protocol and, 191–206
transaction states, 196
TransactionConstants interface, 196
TransactionFactory class, 187, 209
TransactionManager interface, 186
starting a transaction and, 192–193
two-phase commit and, 191
TransactionParticipant interface
joining a transaction and, 195–196
two-phase commit and, 191
transactions
ACID properties of, 188–189
ancestors in, 212
committing or aborting in, 187–188
completing, 185–186, 197–204
crash recovery and, 204–205
definition of, 304
event registration and, 169
inferior transactions, 297
joining, 195–196
managers for, 186
minimizing protocols for, 185
nested transaction and, 193–194
participants in, 187
pure transactions and, 300
requirements of, 189–190
semantics of, 187, 302
starting, 192–193
two-phase commit in, 67, 186
uses of, 185
transactions, JavaSpaces, 269–271
ACID properties and, 270–271
notify operation in, 270
read operation in, 269–270
take operation in, 270
write operation in, 269
transaction semantics, 207–214
CannotNestException class, 212
NestableServerTransaction class, 209–212
NestableTransactionManager interface, 207–209
orphans and, 213
sequential execution and, 212
serialized forms of transaction classes, 214
ServerTransaction class, 209–212
Transaction interface, 207–209
TransactionFactory class, 209
two-phase locking in, 212–213
visibility and, 213
VOTING stage and, 213
translators, note to, 385
transport, 305
transport layer, 305
two-phase commit protocol, 191–206
completing a transaction, client's view, 197–198
completing a transaction, manager's view, 202–204
completing a transaction, participant's view, 199–201
crash recovery and, 204–205
defining with primary types, 191
durability of, 205
importing types for, 191

Jini transaction interfaces and, 67
joining a transaction, 195–196
starting a nested transaction, 193–194
starting a transaction, 192–193
transaction states and, 196
Twain, Mark
quotation, 3
two-phase locking, 212–213

U

UDP, 84
unicast discovery, 97–100
referencing remote djinns with, 98
request format of, 99
as request-response protocol, 98–99
response format of, 100
unicast TCP and, 98
unicast discovery protocol
definition of, 85
discovery process in, 85
`net.jini.core.lookup.ServiceRigistrar` and, 88
unified objects, 308–310
`UnknownEventException`, 165–166
`UnknownLeaseException`
serialized forms of, 148
use of, 145
`UnusableEntryException`
entry specification and, 129–130
JavaSpaces services and, 261
`URL` syntax, 108
user interfaces, 75
user specifications, 20–21
utilities specification, 111–125
multicast discovery utility and, 113–118
protocol utilities and, 119–125

V

variables, 92
visibility, transactions, 213
`VOTING` stage, 187, 213

W

weak references, 305
wide area networks (WANs), 89
workgroups, 65
`write`, 261, 264
in transactions, 269

Colophon

***Collaboration**, n.:*
A literary partnership based on the false assumption that the other people can spell.

THIS book is set in 11 point Times Roman, with variations of size, angle, and weight for headers, chapter quotes, and diagram labels. All code is set in `Lucida Sans Typewriter` at 83% of the surrounding text size. A few decorations are in Zapf Dingbats.

The text was written using FrameMaker on several Sun workstations and two Macintosh laptop computers.

Code examples in the introductory material and its associated appendix were written and compiled on the Solaris systems and then broken into fragments by a Perl script looking for specially formatted comments. Source fragments and generated output were inserted in the book by another Perl script.

NOTE TO TRANSLATORS

The fonts in this book have been chosen carefully. The font for code, when mixed with body text, has the same "x" height and roughly the same weight and "color." `Code` in text looks even—if you read quickly it can seem like body text, but it is nonetheless easy to tell that `code` text *is* different. Please use the fonts that we have used (we would be happy to help you locate any that you do not have) or choose other code and body fonts that are balanced in the same way.

ABOUT THE AUTHORS

KEN ARNOLD, of Sun Microsystems, Inc., is one of the original architects of the Jini technology and is the lead engineer of Sun's JavaSpaces technology. He is the co-author, with James Gosling, of *The Java Programming Language* and is a leading expert in object-oriented design, C, C++ and distributed computing.

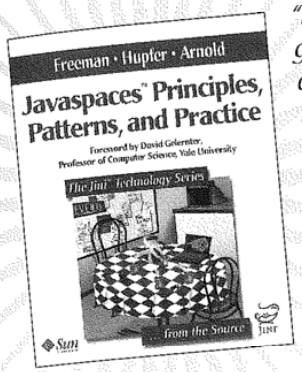
BRYAN O'SULLIVAN, while at Sun Microsystems, Inc., developed the Jini Discovery and Join Protocol. He supports his rock climbing habit by designing and building distributed systems.

ROBERT W. SCHEIFLER is a Senior Staff Engineer and one of the original architects of Jini technology with Sun Microsystems, where he has been responsible for the design and implementation of the lookup service and the associated discovery protocol and attribute schema. Before joining Sun, he spent nine years as Director and then President of the X Consortium, a non-profit organization devoted to the development and evolution of the X Window System. He was chief architect of the X Window System protocol, and created the Consortium originally while a principal research scientist at the MIT Laboratory for Computer Science.

JIM WALDO is a Distinguished Engineer with Sun Microsystems, where he has been the lead architect for the Jini project since its inception. Prior to the Jini project, Jim worked in Sun's Java Software group and in Sun Microsystems Laboratories, doing research in the areas of object-oriented programming and systems, distributed computing, and user environments. Jim is also on the faculty of Harvard University, where he teaches distributed computing in the department of computer science.

ANN WOLLRATH is a Senior Staff Engineer with Sun Microsystems where she is the architect of the Java Remote Method Invocation (RMI) system and one of the original architects of the Jini technology. Previously, during her tenure at Sun Microsystems Laboratories and at the MITRE Corporation, she researched reliable, large-scale distributed systems and parallel computation.

Introducing...The Jini™ Technology Series



"Ever since I first saw David Gelernter's Linda programming language almost twenty years ago, I felt that the basic ideas of Linda could be used to make an important advance in the ease of distributed and parallel programming. As part of the fruits of Sun's Jini project, we now have the JavaSpaces technology, a wonderfully simple platform for developing distributed applications that takes advantage of the power of the Java programming language. This important book and its many examples will help you learn about distributed and parallel programming. I highly recommend it to students, programmers, and the technically curious."
–Bill Joy, Chief Scientist and co-founder, Sun Microsystems, Inc.

JavaSpaces™ technology, a powerful Jini™ service from Sun Microsystems, facilitates building distributed applications for the Internet and Intranets. The JavaSpaces model involves persistent object exchange "areas" in which remote Java™ processes can coordinate their actions and exchange data. It provides a necessary ubiquitous, cross-platform framework for distributed computing, emerging as a key technology in this expanding field.

This book introduces the JavaSpaces architecture, provides a definitive and comprehensive description of the model, and demonstrates how to use it to develop distributed computing applications. The book presents an overview of the JavaSpaces design and walks you through the basics, demonstrating key features through examples. Every aspect of JavaSpaces programming is examined in depth: entries, distributed data structures, synchronization, communication, application patterns, leases, distributed events, and transactions. You will find information on the official JavaSpaces specification from Sun Microsystems. ***JavaSpaces Principles, Patterns, and Practice*** also includes two full-scale applications—one collaborative and the other parallel—that demonstrate how to put the JavaSpaces model to work.

JINI™

The Jini™ Technology Series

From the creators of the Jini™ technology at Sun Microsystems comes the official Series for reference material and programming guides. Written by those who design, implement, and document the technology, these books show you how to use, deploy, and create applications using the Jini architecture. The Series is a vital resource of unique insights for anyone utilizing the power of the Java™ programming language and the simplicity of Jini technology.

...from the Source™

http:/java.sun.com/docs/books/jini

Addison-Wesley

Addison-Wesley Computer and Engineering Publishing Group

How to Interact with Us

1. Visit our Web site

http://www.awl.com/cseng

When you think you've read enough, there's always more content for you at Addison-Wesley's web site. Our web site contains a directory of complete product information including:

- Chapters
- Exclusive author interviews
- Links to authors' pages
- Tables of contents
- Source code

You can also discover what tradeshows and conferences Addison-Wesley will be attending, read what others are saying about our titles, and find out where and when you can meet our authors and have them sign your book.

2. Subscribe to Our Email Mailing Lists

Subscribe to our electronic mailing lists and be the first to know when new books are publishing. Here's how it works: Sign up for our electronic mailing at **http://www.awl.com/cseng/mailinglists.html**. Just select the subject areas that interest you and you will receive notification via email when we publish a book in that area.

3. Contact Us via Email

cepubprof@awl.com
Ask general questions about our books.
Sign up for our electronic mailing lists.
Submit corrections for our web site.

bexpress@awl.com
Request an Addison-Wesley catalog.
Get answers to questions regarding your order or our products.

innovations@awl.com
Request a current Innovations Newsletter.

webmaster@awl.com
Send comments about our web site.

mikeh@awl.com
Submit a book proposal.
Send errata for an Addison-Wesley book.

cepubpublicity@awl.com
Request a review copy for a member of the media interested in reviewing new Addison-Wesley titles.

We encourage you to patronize the many fine retailers who stock Addison-Wesley titles. Visit our online directory to find stores near you or visit our online store: **http://store.awl.com/** or call **800-824-7799**.

Addison Wesley Longman
Computer and Engineering Publishing Group
One Jacob Way, Reading, Massachusetts 01867 USA
TEL 781-944-3700 • FAX 781-942-3076