Network Working Group
Request for Comments: 951

Bill Croft (Stanford University)
John Gilmore (Sun Microsystems)
September 1985

BOOTSTRAP PROTOCOL (BOOTP)

1. Status of this Memo

This RFC suggests a proposed protocol for the ARPA-Internet community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited.

2. Overview

This RFC describes an IP/UDP bootstrap protocol (BOOTP) which allows a diskless client machine to discover its own IP address, the address of a server host, and the name of a file to be loaded into memory and executed. The bootstrap operation can be thought of as consisting of TWO PHASES. This RFC describes the first phase, which could be labeled 'address determination and bootfile selection'. After this address and filename information is obtained, control passes to the second phase of the bootstrap where a file transfer occurs. The file transfer will typically use the TFTP protocol [9], since it is intended that both phases reside in PROM on the client. However BOOTP could also work with other protocols such as SFTP [3] or FTP [6].

We suggest that the client's PROM software provide a way to do a complete bootstrap without 'user' interaction. This is the type of boot that would occur during an unattended power-up. A mechanism should be provided for the user to manually supply the necessary address and filename information to bypass the BOOTP protocol and enter the file transfer phase directly. If non-volatile storage is available, we suggest keeping default settings there and bypassing the BOOTP protocol unless these settings cause the file transfer phase to fail. If the cached information fails, the bootstrap should fall back to phase 1 and use BOOTP.

Here is a brief outline of the protocol:

1. A single packet exchange is performed. Timeouts are used to retransmit until a reply is received. The same packet field layout is used in both directions. Fixed length fields of maximum reasonable length are used to simplify structure definition and parsing.

2. An 'opcode' field exists with two values. The client broadcasts a 'bootrequest' packet. The server then answers with a 'bootreply' packet. The bootrequest contains the client's hardware address and its IP address, if known.

3. The request can optionally contain the name of the server the client wishes to respond. This is so the client can force the boot to occur from a specific host (e.g. if multiple versions of the same bootfile exist or if the server is in a far distant net/domain). The client does not have to deal with name / domain services; instead this function is pushed off to the BOOTP server.

4. The request can optionally contain the 'generic' filename to be booted. For example 'unix' or 'ethertip'. When the server sends the bootreply, it replaces this field with the fully qualified path name of the appropriate boot file. In determining this name, the server may consult his own database correlating the client's address and filename request, with a particular boot file customized for that client. If the bootrequest filename is a null string, then the server returns a filename field indicating the 'default' file to be loaded for that client.

5. In the case of clients who do not know their IP addresses, the server must also have a database relating hardware address to IP address. This client IP address is then placed into a field in the bootreply.

6. Certain network topologies (such as Stanford's) may be such that a given physical cable does not have a TFTP server directly attached to it (e.g. all the gateways and hosts on a certain cable may be diskless). With the cooperation of neighboring gateways, BOOTP can allow clients to boot off of servers several hops away, through these gateways. See the section 'Booting Through Gateways' below. This part of the protocol requires no special action on the part of the client. Implementation is optional and requires a small amount of additional code in gateways and servers.

3. Packet Format

All numbers shown are decimal, unless indicated otherwise. The BOOTP packet is enclosed in a standard IP [8] UDP [7] datagram. For simplicity it is assumed that the BOOTP packet is never fragmented. Any numeric fields shown are packed in 'standard network byte order', i.e. high order bits are sent first.

In the IP header of a bootrequest, the client fills in its own IP source address if known, otherwise zero. When the server address is unknown, the IP destination address will be the 'broadcast address' 255.255.255.255. This address means 'broadcast on the local cable, (I don't know my net number)' [4].

The UDP header contains source and destination port numbers. The BOOTP protocol uses two reserved port numbers, 'BOOTP client' (68) and 'BOOTP server' (67). The client sends requests using 'BOOTP server' as the destination port; this is usually a broadcast. The server sends replies using 'BOOTP client' as the destination port; depending on the kernel or driver facilities in the server, this may or may not be a broadcast (this is explained further in the section titled 'Chicken/Egg issues' below). The reason TWO reserved ports are used, is to avoid 'waking up' and scheduling the BOOTP server daemons, when a bootreply must be broadcast to a client. Since the server and other hosts won't be listening on the 'BOOTP client' port, any such incoming broadcasts will be filtered out at the kernel level. We could not simply allow the client to pick a 'random' port number for the UDP source port field; since the server reply may be broadcast, a randomly chosen port number could confuse other hosts that happened to be listening on that port.

The UDP length field is set to the length of the UDP plus BOOTP portions of the packet. The UDP checksum field can be set to zero by the client (or server) if desired, to avoid this extra overhead in a PROM implementation. In the 'Packet Processing' section below the phrase '[UDP checksum.]' is used whenever the checksum might be verified/computed.

FIELD	BYTES	DESCRIPTION
op	1	packet op code / message type. 1 = BOOTREQUEST, 2 = BOOTREPLY
htype	1	hardware address type, see ARP section in "Assigned Numbers" RFC. '1' = 10mb ethernet
hlen	1	hardware address length (eg '6' for 10mb ethernet).
hops	1	client sets to zero, optionally used by gateways in cross-gateway booting.
xid	4	transaction ID, a random number, used to match this boot request with the responses it generates.
secs	2	filled in by client, seconds elapsed since client started trying to boot.

--	2	unused
ciaddr	4	client IP address; filled in by client in bootrequest if known.
yiaddr	4	'your' (client) IP address; filled by server if client doesn't know its own address (ciaddr was 0).
siaddr	4	server IP address; returned in bootreply by server.
giaddr	4	gateway IP address, used in optional cross-gateway booting.
chaddr	16	client hardware address, filled in by client.
sname	64	optional server host name, null terminated string.
file	128	boot file name, null terminated string; 'generic' name or null in bootrequest, fully qualified directory-path name in bootreply.
vend	64	optional vendor-specific area, e.g. could be hardware type/serial on request, or 'capability' / remote file system handle on reply. This info may be set aside for use by a third phase bootstrap or kernel.

4. Chicken / Egg Issues

How can the server send an IP datagram to the client, if the client doesnt know its own IP address (yet)? Whenever a bootreply is being sent, the transmitting machine performs the following operations:

1. If the client knows its own IP address ('ciaddr' field is nonzero), then the IP can be sent 'as normal', since the client will respond to ARPs [5].

2. If the client does not yet know its IP address (ciaddr zero), then the client cannot respond to ARPs sent by the transmitter of the bootreply. There are two options:

   a. If the transmitter has the necessary kernel or driver hooks

to 'manually' construct an ARP address cache entry, then it can fill in an entry using the 'chaddr' and 'yiaddr' fields. Of course, this entry should have a timeout on it, just like any other entry made by the normal ARP code itself. The transmitter of the bootreply can then simply send the bootreply to the client's IP address. UNIX (4.2 BSD) has this capability.

b. If the transmitter lacks these kernel hooks, it can simply send the bootreply to the IP broadcast address on the appropriate interface. This is only one additional broadcast over the previous case.

5. Client Use of ARP

The client PROM must contain a simple implementation of ARP, e.g. the address cache could be just one entry in size. This will allow a second-phase-only boot (TFTP) to be performed when the client knows the IP addresses and bootfile name.

Any time the client is expecting to receive a TFTP or BOOTP reply, it should be prepared to answer an ARP request for its own IP to hardware address mapping (if known).

Since the bootreply will contain (in the hardware encapsulation) the hardware source address of the server/gateway, the client MAY be able to avoid sending an ARP request for the server/gateway IP address to be used in the following TFTP phase. However this should be treated only as a special case, since it is desirable to still allow a second-phase-only boot as described above.

6. Comparison to RARP

An earlier protocol, Reverse Address Resolution Protocol (RARP) [1] was proposed to allow a client to determine its IP address, given that it knew its hardware address. However RARP had the disadvantage that it was a hardware link level protocol (not IP/UDP based). This means that RARP could only be implemented on hosts containing special kernel or driver modifications to access these 'raw' packets. Since there are many network kernels existent now, with each source maintained by different organizations, a boot protocol that does not require kernel modifications is a decided advantage.

BOOTP provides this hardware to IP address lookup function, in addition to the other useful features described in the sections above.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.