

Network Table Lookup

Efficiency scheme

Fri, Jul 5, 2002

Introduction

Most modern network communication is based on Internet Protocols. The Linac/IRM front-end system code supports the usual IP, ARP, ICMP, and IGMP protocol standards, and it supports the User Datagram Protocol (UDP) transport layer. Inside the system code, as well as within each local or page application, a form of node number is used when referring to any network entity. In order to relate to a UDP socket, a “pseudo node number” (PNN) is fabricated that refers to an entry in the IPARP table, each element of which holds a 6-byte physical address, IP address, and a pointer to an allocated block of memory that contains up to 15 active UDP port numbers. This table thus serves the needs of an ARP cache, but also the needs of retaining the source UDP port numbers. To be more specific, the form of a 16-bit pseudo node number is as follows:

<i>Bits</i>	<i>Size</i>	<i>Meaning</i>
15–12	4	value 6 or 0xE indicates a pseudo node number
11–4	8	Index into IPARP table of 16-byte entries
3–0	4	Index to list of active port numbers in Port number block

Given a pseudo node number, then, it is easy to find the corresponding IP address and UDP port number that comprise what is normally referred to as a UDP socket. When a PNN is passed to lower level network routines, a simple IPARP table lookup is enough to find the information needed to build the UDP datagram for transmission.

On the reception side, things are not so simple. When a UDP datagram is received, one starts with the information contained within the network frame, which includes the source IP address and UDP port number. One of the early steps the system makes is constructing a relevant pseudo node number for referring to the node and UDP port number that sent this datagram. (In the case of a request, one may soon have to deliver a reply, for which a PNN is needed.) Traditionally, this has been accomplished by searching the IPARP table for a match on the IP address, then searching the list of UDP port numbers active from that node in the associated Port number block.

Two other forms of node numbers are supported by the system. One form is a “native” node number (NNN), which is assigned uniquely to every node that runs the system software. These numbers are in the range 0x0500–0x07FF. Today, by convention, most “operational” front-ends use numbers in the 06xx range, whereas nodes used primarily for testing use numbers in the 05xx range. Almost no nodes now use the 07xx range, which was used more extensively years ago for the DZero experiment front-ends during Run I.

Communication with native node numbers is also supported by the system code. This allows a higher level of support for communications that is not even dependent upon the IP protocols. The two particular protocols that are supported in this way are the Classic and Acnet protocols. Each protocol is supported by server code that listens to the two UDP ports assigned for these protocols, specifically, 6800 for Classic and 6801 for Acnet. Within the front-end code, a Classic message is transmitted by the system support from the Classic port, and an Acnet message is sent from the Acnet port.

Another type of node number is supported by all Acnet protocol nodes, including Acnet consoles and other Acnet client nodes. This type is referred to here as an Acnet node number (ANN). All of these node numbers use IP communications with the Acnet protocol. Every ANN, then, has a corresponding IP address, and every Acnet front-end needs to obtain a copy of the Acnet Trunk tables that are merely a list of IP addresses indexed by Acnet node numbers.

The reserved range of Acnet node numbers is 0x0900–0x10FF. At this time, the range in actual use is 0x0900–0x0BFF.

The system code supports communications with a PNN, NNN, or ANN. Switching between these various forms and the related IP addresses often involves table searching. Searching the IPARP table to form the proper PNN was described above. But when the target node is characterized by a NNN, the IP Node Address Table (IPNAT) must be searched. This table is really a kind of DNS cache that each node maintains. A new entry is placed in this table the first time that the front-end ever targets a given NNN. (This table survives resets and power outages, because it resides in nonvolatile memory.) The first time it is used, the DNS is queried to obtain the IP address, the DNS reply serving to populate the entry. Once populated, periodic (12-hour) queries to the DNS serve to keep the entry reasonably up-to-date. (These periodic queries are staggered so as not to place a high load on the DNS at one time.) All of this logic is managed by the local application LOOPDNSQ.

Another time that a search of the IPNAT is required is upon reception of a reply message to a server-style Classic data request. The original Classic request is described in terms of “idents,” which include an NNN and an index. (A simple example is an analog channel ident that includes a NNN and a channel number local to that node.) A server that multicasts such a request can anticipate replies from as many unique nodes as are represented in the idents of the request. In order to match the replying node with an NNN from which a reply is expected—so the data can be properly captured into the eventual composite reply buffer for transmission to the original requester—the NNN of the incoming datagram must be found, which requires a search of the IPNAT for a match on the IP address.

When a target node is specified as an ANN, the local copy of the Trunk table provides the needed IP address, in this case without searching. But when a reply is to be sent to a PNN using the Acnet protocol, the Trunk table must be searched to discover the appropriate ANN to use in the Acnet header portion of the message.

How bad can such searching be? The size of the IPNAT may be short, or not so short, depending on the history of that particular node’s network communications. The Trunk table size is fairly fixed, as it includes IP addresses for all existing Acnet nodes. The IPARP table size depends upon the number of nodes with which the node has communicated recently. (Entries are freed after inactivity of about an hour or so.) Its maximum size is about 250 entries, although a more typical size might be only 10–20% of that.

It was thought that modern processors are fast enough so that whatever table searches are needed should not be too costly. The latest version of the system code runs with a 233 MHz PowerPC processor, which is good, but its nonvolatile memory has an approximate access time of 1 μ s, which is not so good. The CPU has a RISC architecture, so one can think of a single access to nonvolatile memory costing some 200-odd CPU wait states. This elevated the concern about time spent searching nonvolatile network tables.

The purpose of this note is to describe a new scheme for eliminating node number and IP address searches.

Origin of 16-bit node numbers

The original motivation for using a 16-bit node number came from the Classic data request protocol design. When making a request for specific data of any kind, the specificity is defined in terms of an ident, which is most often 4 to 6 bytes in length, but in all cases includes a 16-bit node number in the first two bytes. (This design was used to support access

between front-ends. Any front-end can both issue requests and send replies in answer to requests received; both client and server software is included.) The protocol supports requests that span multiple nodes, in which the nodes involved are those whose (native) node numbers are included in the ident list used in the request. When a request is issued from one node, in which data from several nodes is sought, it is multicast, so that all eligible nodes receive the request at once. Each node inspects the request to discover whether any of the idents refers to itself. If any of the idents is local, then that node accepts the request and initializes itself to return the requested data (its own portion only) at the requested rate to the requesting node.

Back at the requesting node, the request message that was multicast was carefully analyzed ahead of time, so that the requesting node knows what data it should expect to receive in reply, and from which nodes. As those expected replies are received, it uses a "request map" to spread the reply data appropriately into the ultimate composite reply buffer. Most often, such requests are issued on behalf of a client node, such as a PC, Macintosh, or unix box, in which the requesting node described here is called a "server" node. Although this description focuses on the Classic protocol, the same server support also exists to support the Acnet protocols, in which case the requests are most often issued on behalf of an Acnet console. In either case, the original client node has an easy job: build the request including all the relevant idents in the appropriate protocol, send the request to a server node, and anticipate the consequent composite replies that are returned by the server node. The complexity of sorting out the separate replies arriving at the server node from the actual source nodes is eliminated, as all that is handled by the server node. But the potentially high traffic of these replies means that the server node must frequently translate from IP address and port number to PNN, and also to a NNN, in order to consult the request map.

Table lookup scheme

A new scheme has been designed to save time when looking up IP address and node number relationships. The first attempt at such support was done as a pilot study in the IRM code, although these MVME-162-based nodes do not have slow access to nonvolatile memory. It supported efficient lookups for all of Fermilab's Class B internet. It was relatively simple, but it required a total of 640K bytes to house the needed lookup tables. The second attempt is designed, at the cost of increased complexity, to fit within a target memory budget of 64K bytes. In addition, it not only supports the local Class B internet addresses, but it also supports IP addresses that are outside that range.

Within the new 64K byte structure, 50K bytes are used to house up to 100 subnet blocks, each of which contains 64 entries of 8-bytes each. The 64 entries support up to 4 related node numbers for 64 consecutive IP addresses. Since 6 bits is enough to cover the 64 consecutive addresses, the subnet size corresponds to 26 bits of the 32-bit IP address.

Subnet blocks are allocated as needed to contain the node numbers of interest. A subnet block is needed when a nonzero node number must be associated with an IP address. Once a subnet block is allocated, the same block can be used for as many as 63 other IP addresses that share the same upper 26 bits.

At initialization time, the contents of the nonvolatile memory tables IPNAT and TRUNK are reviewed to help initialize the subnet tables. The IPNAT keeps native node numbers that relate to IP addresses; the TRUNK table keeps Acnet node numbers that relate to IP addresses. At the time of this writing, node0509, which maintains a rather large IPNAT, uses about 64 subnet blocks as a result of this initialization process. After that, most new needs for subnet block allocation will occur as a result of communication with IP addresses unknown

to the IPNAT or TRUNK tables. When a new entry of a foreign node is entered into the IPARP table cache, a subnet block may be allocated. Without further communication from such a node, the subnet block will be freed after an hour or so. The new scheme logs such occurrences as a matter of diagnostic interest.

The mechanism for allocating subnet blocks involves a check of the IP address in question against the local IP address. If the upper 16 bits match, the IP address is considered to be part of the local Class B internet. For such a case, a 1024-member array of allocated subnet block numbers, each in the range 1–100, is kept, indexed by the next 10 bits of the IP address. If the given IP address has an allocated block, then the least-significant 6 bits of the IP address are used to index into the allocated block to find the relevant 4-word entry.

But suppose the upper 16 bits of the IP address did not match the local IP address. This is referred to as an external subnet, for which a table of 64 possible 26-bit external subnet masks is kept. This table must be searched for a match against the upper 26 bits of the given IP address. If a match is found, then a parallel table of external subnet block numbers is referenced in the same way that was done for the local subnets. A block number obtained in this way is then used in the same way, indexing via the low 6 bits of the IP address.

Whether the local or external case, suppose there is no allocated block for the subnet in question. In that case, one must be allocated, assuming it is necessary to store a nonzero node number therein. An array of 100 counts of active entries, indexed by subnet block number, is maintained to facilitate discovery of an unused subnet block. On finding a zero entry in this table, its index is an unused block number that can be allocated. The act of storing a nonzero node number in one of the entries of the new subnet block will increment the count so that the new block will appear unavailable for future allocation. Likewise, any time that a node number is removed from the table entry, and that removal results in that 8-byte entry becoming all zero, the count of active entries for that subnet block is decremented. Once such decrementing reaches zero, the subnet block is freed and is available for future allocation as needed. Note that, by definition, such a freed block is all zero.

Returning to the original purpose for the new scheme, that of minimizing the time to lookup a node number given an IP address, here is how that function typically proceeds. A check is made of the IP address against the upper 16 bits of the local node IP address. This will very likely result in a match, so that the IP address is local. Using the next 10 bits of the address as an index, lookup the block number that has been allocated for this subnet, which leads to the subnet block itself, where the low 6 bits of the IP address are used to index to the 8-byte entry that holds up to 4 related node numbers for the given IP address. In the case that no allocated block exists, the resulting node number sought is simply zero. There is no reason to allocate a subnet block unless it is desired to store a nonzero node number of some type in its 8-byte entry. Two error conditions can occur for a table lookup operation. The first is when an IP address is found for which no subnet block is allocated, and no subnet block is available for allocation. The second is when an external IP address is found not in the list of known external subnets, and the list is full. In either error case, the procedure should be to use the old method that required a search of the network table in question. It is not expected that this will happen at all frequently in practice.

A document describing the evolution of the scheme is called "Table Lookup Improvement." A document describing the communications between local applications and the system code *vis-a-vis* changes needed for IPNOD table entries is called "Table Lookup for IP Addresses." A document detailing the changes required to the system code to support the table lookup scheme, plus preliminary timing results, is called "Network Table Changes."