Implementing XNS Protocols for 4.2bsd

James O'Toole <james@maryland>
Chris Torek <chris@maryland>
Mark Weiser <mark@maryland>

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

We have implemented the Xerox Networking Systems protocol suite in 4.2BSD UNIX[†] in order to communicate with Xerox workstations which do not implement IP/TCP. In this paper, we discuss the problems we encountered while making the necessary changes to the UNIX kernel. We found a multitude of IP/TCP dependencies in the lowest level UNIX device driver and network interface code, which were eliminated by replacing inline code with protocol specific subroutine calls. Another problem was that 4.2BSD expects all protocols to be connectionless or byte stream, and the XNS stream level is neither. We generalized the socket code to satisfy the additional requirements of the XNS stream protocol. The result of all these changes is not just the ability to handle XNS but a more general network system without IP/TCP dependencies, ready for the next generation of protocols.

Portions of this work were supported by grants from the National Science Foundation, the Air Force Office of Scientific Research, Xerox Corporation, and Digital Equipment Corporation.

[†] UNIX is a trademark of AT&T Bell Laboratories

Implementing XNS Protocols for 4.2bsd

Introduction

We have implemented the Internet Datagram Protocol (IDP) and the Sequenced Packet Protocol (SPP) of the Xerox Networking Systems (XNS) protocol suite. In June of 1984, Xerox corporation awarded the Computer Science Department thirty Xerox Development Environment (XDE) workstations and associated equipment for research purposes. This equipment was not due to arrive until mid-November; in the meantime, we obtained as much documentation as possible, and read. This quickly became boring, so we started looking around for something to *do!* We implemented the XNS protocols under 4.2BSD UNIX† on our department research minicomputers because we needed them to communicate with the Xerox workstations.

The Berkeley UNIX kernel supports two different forms of interprocess communication (IPC): pipes and sockets. Pipes are a standard UNIXism, but sockets are a new form of UNIX IPC. As distributed, 4.2BSD supports three different domains of communication within the socket abstraction: UNIX internal, DOD IP/TCP, and Xerox PUP. The UNIX internal protocols do not provide for communication between processes on separate UNIX systems. The Defense Department protocols IP [1] and TCP [2] are used widely, and are the primary means of intermachine communication in use in the department today. The Xerox PARC Universal Protocol (PUP) is a predecessor to the newer XNS [3] protocols. However, the Xerox workstations do not support any of these protocols, so we were faced with two choices:

1. Implement IP/TCP on the XDE.
2. Implement XNS under 4.2BSD.

We chose to implement XNS under 4.2BSD for a number of reasons:

1. We are very familiar with the 4.2BSD UNIX kernel.
2. The Xerox workstations had not yet arrived when we were ready to begin work.
3. We wanted the UNIX machines to communicate with the workstations as soon as they arrived.

This paper describes some of the changes we chose to make to the standard 4.2BSD socket code in order to permit the addition of the XNS protocols, as well as the implementation details of the XNS protocols themselves.

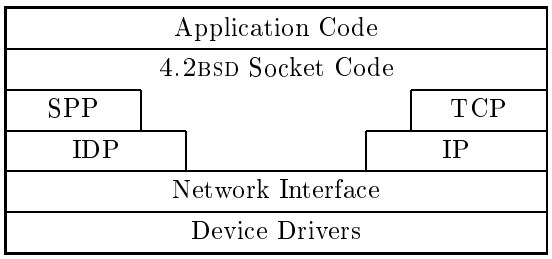
† UNIX is a trademark of AT&T Bell Laboratories

[1] See RFC 791 for the complete Internet Protocol specification

[2] See RFC 793 for the complete Transmission Control Protocol specification

[3] See *Internet Transport Protocols* (ISIS 018112), the XNS protocol specification

1. Preparing for Multiple Protocol Families



4.2BSD **Network Kernel Organization**
Figure 1.

The 4.2BSD socket code is designed around the IP/TCP protocols. Figure 1 shows how the 4.2BSD network kernel is organized, including the XNS protocol implementation we have added. The data structures used for network interfaces contain fields called `if_net` and `if_host`, which are used for speedy access to network and host numbers. However, other (non-IP) addresses have very different formats, so these data structures are too inflexible. Similarly, even general socket code makes assumptions about the internal format of a `struct sockaddr`. Frankly, the problem was that all the code was using `bcmp()` [4] to deal with network addresses. We also modified the code which manipulates `struct sockbuf`s to support protocols which differ from those already included in 4.2BSD. All of these changes are internal to the 4.2BSD socket abstraction, and therefore transparent to user code.

1.1. Remove Fields from Network Interface Structure

In order to remove these stupid fields (i.e., `if_net`, `if_host`) we had to provide the same address lookup and network number matching functionality as before. So, we rewrote several routines in `if.c` to use address family based comparison routines and use *only* the `if_addr` field. All the code that "knew" about the `if_net` or `if_host` fields had to be changed to use the new versions of these routines.

1.2. Address Family Based Address Comparison

Too much socket code made assumptions about the internal format of a `struct sockaddr` in each address family, so we changed this code to use two new routines in the `afswitch` table, based on the address family of the address in question. The new routines in `afswitch` are:

1. `af_addrmatch()`
2. `af_rtinit()`

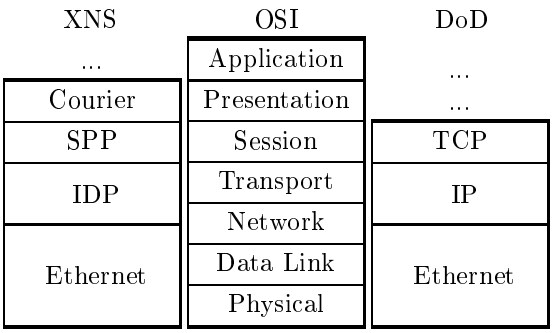
The `af_addrmatch` routine is a predicate which decides (for a particular address family) whether two `struct sockaddr`s represent the same host. The `af_rtinit` routine is called at interface initialization time to set up a routing table entry in the proper way (for a particular address family). Also, the `af_netmatch` was put to good use by all the code which had been using the `if_net` field of the network interface structure.

1.3. Support for Multiple Addresses Per Interface

Since we wanted to use the same Ethernet interface for both our IP/TCP and XNS communications, we needed to support multiple addresses per interface. Then we could get to the *real* work: actually implementing the datagram and packet stream XNS protocols.

To support multiple addresses per interface, we had to hide even more knowledge of `struct ifnet` from most code. That meant we had to modify all the IP/TCP code to use the routines in `if.c` for all access to

[4] `bcmp()`≡assembly code for block memory compare



Relationship among XNS, OSI, and IP/TCP
Figure 2.

interface addresses. We further modified all the code in `if.c` to search a list of addresses for each interface, and changed the `if_addr` field in `struct ifnet` into a small array of addresses.

1.4. Connectionless, Atomic, and Rights-Based Protocols

The code dealing with socket buffers (send and receive queues) assumed that any protocol was either a byte stream (e.g., TCP, pipes) or a connectionless protocol with optional rights (e.g., UDP). There are protocols which are atomic and also connection based. Such protocols would not pass source addresses with each message. 4.2BSD already includes flags called `PR_ATOMIC`, `PR_ADDR`, and `PR_RIGHTS`; we modified the socket buffer code and the receive system call to use these flags independently. With our changes, a protocol may be atomic without being unreliable, or vice versa. The only remaining restriction is that protocols with source addresses and/or rights must be atomic.

2. Actual XNS Protocol Implementation

Figure 2 shows how the IP/TCP and XNS protocol families fit into the the seven level OSI model of the International Standards Organization. There are differences between IP and IDP addressing that cause some problems when trying to splice IDP support into the 4.2BSD socket level interface. The semantic and functional differences between TCP and SPP are so great that providing SPP to the user level through the `SOCK_STREAM` type interface is difficult.

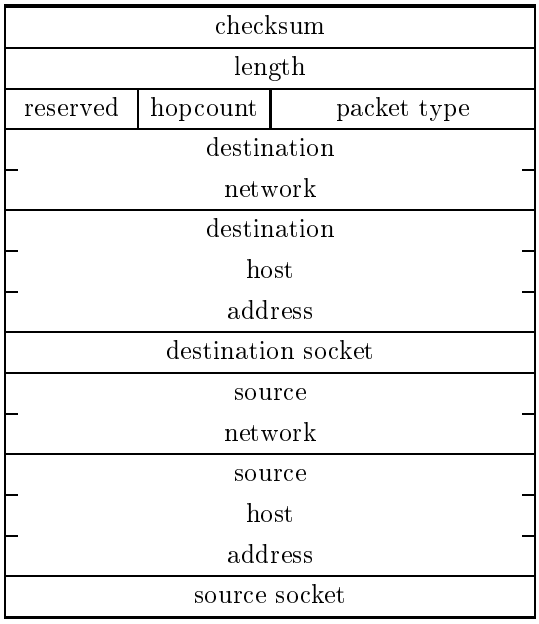
2.1. Internet Datagram Protocol

Both the Internet Protocol (Defense Department) and the Internet Datagram Protocol (Xerox) are designed to unreliably transport reasonable size pieces of data (*packets*) from one machine to another. Therefore, as one would expect, they are very similar. Both wrap headers around the data to be transported. Both headers provide for source, destination, rudimentary error checking, and some control over who (at the destination machine) receives the data. However, there are a number of big differences between IP and IDP semantics and functionality. Figure 3 shows how these fields are arranged in an IDP header.

IP includes all sorts of doodads for doing prioritized service and security, fragmentation for networks that can't handle large packet sizes, and a multitude of other options (i.e., source routing, recording route taken, packet timestamp, stream identifier). None of these embellishments exist in IDP. In the XNS protocol family, these features are provided by higher level protocols, if at all.

One of the above embellishments that is frequently considered necessary is fragmentation.[5] The IDP strategy for fragmentation is to require the gateways on any network which can't handle the maximum packet size (576 bytes) to implement their own private fragmentation method appropriate to that network. The fragmented packet must, however, leave that network whole once again. To our knowledge, this has never been done, because hardly any networks with such tiny maximum packet sizes use IDP.

[5] Many machines don't implement any of the other IP options.

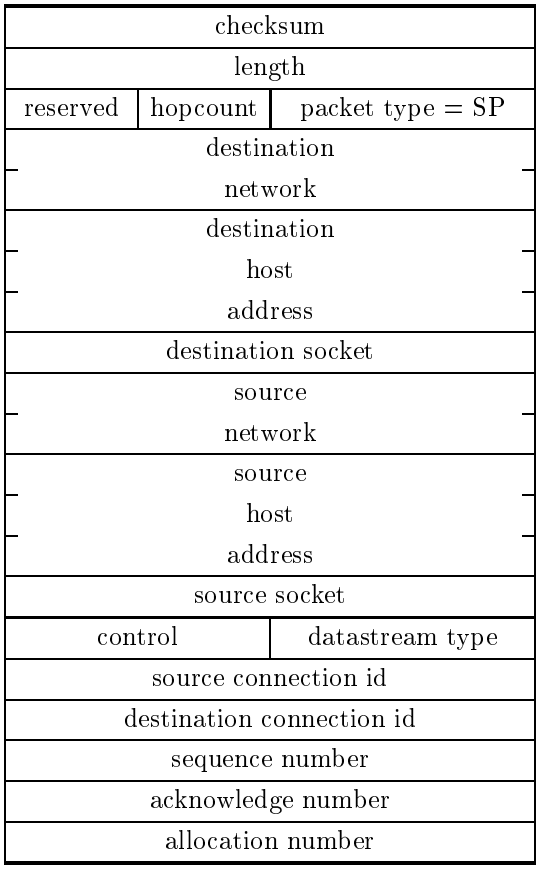


Internet Datagram Protocol Packet Header
Figure 3.

The most important difference between IP and IDP is how the sockets are multiplexed across the upper level protocols. In IP, each packet belongs to a particular *protocol*, and the individual upper level protocols (ICMP, UDP, TCP) must implement some kind of socket number in order to separate the traffic in that protocol. In IDP, the source and destination addresses include a *socket* number which accomplishes this separation. Another IDP header field specifies the *packet type*, and this field is interpreted by the process which receives the packet, *not* by the IDP layer.

Where the IP code would hand each packet to the upper level handler based on the protocol field of the packet, the IDP code must determine the proper handler for the packet based on the socket field, and the upper layers must interpret the protocol field. To be specific, this means the XNS protocol control block (PCB) data structures comprise one big global namespace. The IP/TCP PCBs, on the other hand, are maintained by the individual protocol code. Also, since the format of the sockets in the IP world is dependent upon the upper level protocol, the IP PCBs are necessarily a hodgepodge of various numbers needed by each protocol.

There are a few other minor differences. IDP packets are checksummed, so the data passed to the upper level is assumed to be correct. In IP, each upper level protocol must perform its own checksum. The internal format of XNS addresses includes both a network number and a host number; the network number is used for routing purposes, and the host number uniquely identifies the machine in question. IP addresses have a more complicated format, and do not uniquely identify a machine. This is one of the reasons `bcmp()` shouldn't be used to compare XNS addresses; the network and socket numbers are not relevant to determining whether two `struct sockaddr_xn`s refer to the same machine.



Sequenced Packet Protocol Packet Header
Figure 4.

2.2. Sequenced Packet Protocol

2.2.1. Comparison of TCP and SPP

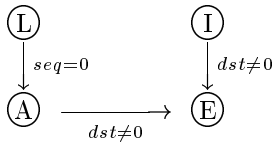
The differences between TCP and SPP are similar to those between IP and IDP. TCP has more options than SPP, while SPP leaves more decisions to the client protocol layer. SPP provides no special acknowledgement sequence to process the closing of a connection; the client protocol must determine when it is safe to shut down a socket. SPP provides more information to the client protocol than TCP. SPP provides end-of-message boundaries and a *datastream type* field, which are interpreted by the client. Figure 4 shows the format of a complete SPP header. The datastream type field is an example of what XNS calls a *bridge* field, that is, a field which is interpreted by the client protocol.[6] The bridge field in IDP is the *packet type*.

Because SPP leaves more of the work to the client layer, implementing it is easier than implementing TCP. However, because SPP provides the extra datastream type information and delimits the end of messages, it does not fit easily into the standard UNIX stream abstraction (`read()` and `write()`). The 4.2BSD socket abstraction also expects protocols to be either

1. Connectionless, usually for unreliable protocols, or
2. Byte Stream, connection based protocols.

The semantics of SPP require that the data packets be delivered to the client separately, and that end-of-message boundaries and the datastream type be communicated as well. Individual protocol requirements

[6] Courier is the primary client of the SPP protocol.



Homegrown Finite State Automaton for SPP
Figure 5.

such as these are what prompted us to make the changes to the socket level code described in section 1.4.

2.2.2. Formal Protocol Specification

The particular Xerox document we used to develop our implementation of IDP and SPP, *Internet Transport Protocols* (ISIS 018112), contains no formal specification of these protocols. It does provide the format of the headers used, and contains prose descriptions of these protocols in operation. These descriptions suffice to specify IDP, but seem inadequate in the case of SPP; the descriptions are too general, and appear to make optional certain behaviors which are in fact required. We found ourselves asking many questions about the details of SPP.

At any time, an end of an SPP connection (established or not) is in a particular state. When a packet arrives, some action must be taken (i.e., adjust window, acknowledge, drop packet, etc.). A complete specification of a protocol such as SPP would provide an algorithm for determining the proper action to be taken given the current state of the connection and the contents of the incoming packet. In other words, we needed a **Finite State Automaton** to answer all our questions. Therefore, we constructed our own FSA, illustrated in Figure 5.

Our SPP state machine is remarkably simple. As it turns out, there are only four states for any given connection: LISTEN, INIT_SENT, ALMOST, and ESTABLISHED.[7] These are represented in Figure 5 as the letters "L", "I", "A", and "E". These states correspond roughly to the TCP states LISTEN, SYN_SENT, SYN_RECEIVED, and ESTABLISHED.

In a typical connection setup, the following events should occur:

1. A server enters the LISTEN state and awaits a connection.
2. A client sends a system packet with sequence number zero to the server, and enters the INIT_SENT state.
3. The server receives the system packet, sends a system packet with sequence number zero in reply, and enters the ALMOST state. At this point, the server may send data, but must not act on any received data (in other words, received data must not be passed up to user code).
4. The client receives the reply and become ESTABLISHED, allowing it to send and receive data.
5. The server receives data (close requests are data, as far as SPP is concerned) from the client, telling it that the client is indeed connected. The server enters the ESTABLISHED state, and may now receive, as well as send, data.

Real networks are not so nice as to deliver packets in order and without loss. Data packets will be retransmitted if they are not acknowledged, but system packets cannot be acknowledged, since they do not consume sequence numbers. Thus data packets are reliably delivered, but system packets are not, and the five-step process above is not sufficient to implement SPP.

With some modifications, however, the procedure outlined above will work. We now describe exactly what action should be taken for an incoming packet in each state. First we make the following definitions:

[7] The Xerox documents use "ESTABLISHED" where we use "ALMOST", and "OPEN" where we use "ESTABLISHED". We prefer our own names.

Definition. An *initiating* packet is any packet with sequence number zero and destination connection ID zero (i.e., unspecified).

Definition. A *fully specified* packet is one with a nonzero destination connection ID. (This connection ID must match the local connection ID at the receiving machine.)

1. In the LISTEN state (entered when the client opens a passive connection), any initiating packet must be accepted, causing a transition to the ALMOST state, and a system packet with sequence number zero must be sent in reply. Any other packets must be dropped (ignored).
2. In the ALMOST state, any fully specified packet must be accepted, causing a transition to the ESTABLISHED state. Other packets must be dropped.
3. In the INIT_SENT state (entered when client opens an active connection, at which time an initiating packet is sent), any fully specified packet must be accepted, causing a transition to the ESTABLISHED state. All other packets must be dropped. If no fully specified packet is received within a reasonable time, another system packet with sequence number zero should be sent.

Acceptance of a packet implies that an acknowledgement must be sent (if requested), windows must be updated, and so forth, as outlined in the Xerox XNS documentation.

We claim that these procedures will successfully transfer data even if the the transport layer loses, reorders, or delays packets. Arguments showing the correctness of this kind of reliable transmission protocol abound; we will not repeat them here. However, a connection could remain in the ALMOST state indefinitely if a stray initiating packet were delivered. Aborting such a connection after an arbitrary timeout is unreasonable, because this is a legitimate state: a client might connect to a server but not send data immediately.

3. Conclusions About Integration of Multiple Protocols

Implementing another unreliable datagram protocol in the 4.2BSD kernel was not too hard. Implementing a reliable protocol was much harder, because the state information for each connection (e.g., FSA state, sequence numbers, data packets) must be maintained flawlessly. As we write this, the equipment being provided by Xerox Corporation is due to begin arriving next week. Therefore, we have not yet had the opportunity to test our implementations with the actual Xerox implementations. Although our implementation does enable our 4.2BSD machines to communicate, there may be some discrepancies with the standard which will only be resolved in the next few months. In our talk, we hope to have more to report about the success of our implementation.

In order to add multiple protocols to 4.2BSD UNIX cleanly, it is necessary to make small modifications to many portions of the socket code. Some of these modifications require corresponding changes in specialized user level programs such as *netstat*, *ifconfig*, and *routed*. Together, all these changes make our current 4.2BSD kernel very different internally from most other kernels, and any bug fixes or other kernel modifications we receive from other 4.2BSD sites must now be installed with care. We had to modify portions of all code in the `net`, `netinet`, and `vaxif` directories. We estimate that we modified approximately 700 lines of code in these directories, and added 2900 lines of code in the new `netxns` directory. We chose to ignore the PUP implementation distributed with 4.2BSD, because it is superceded by the XNS protocols, and was not in use in the Computer Science Department.

As we modified the 4.2BSD network code, we tried to make it as general as possible without doing much more work than necessary to implement the XNS protocols. Now that we have made these changes, we expect that it will be far easier to add new protocols in the future.

Table of Contents

List of Figures