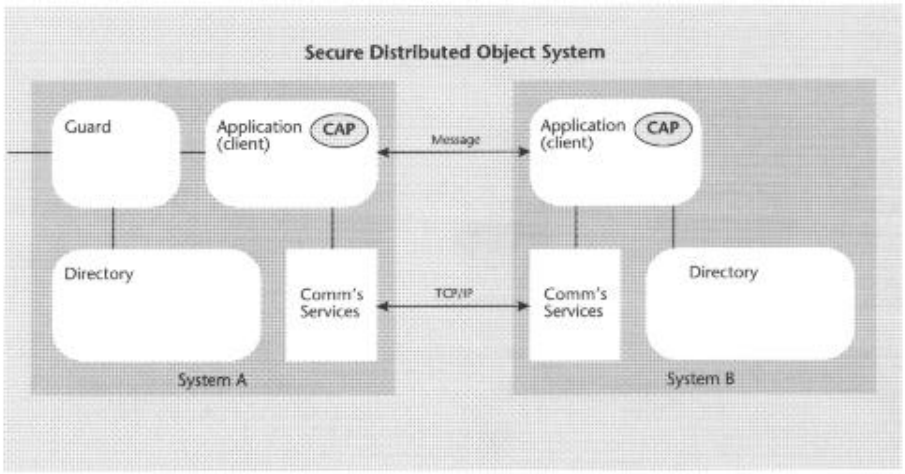
NCR Journal

Volume 4, Number 1

August 1990



This figure from *Progress in Secure Distributed Systems* by Jan Kruys shows a conceptual client/server system that makes use of object-oriented applications that implement their own access security.

In This Issue

The X.25 STREAMS Driver Prototype

L. P. Crosthwaite and M. F. Vigil
SE-San Diego

This is a description of our experiences relocating a portable networking product to a UNIX STREAMS environment. The capabilities of STREAMS make it possible to move existing X.25 networking software from an operating user space into its kernel. To test the feasibility of this assumption and to expose the relevant technical issues a prototype X.25 STREAMS driver was developed. The lessons learned from the prototyping effort are the subject of this article.

STREAMS is a major component of the Networking Support Utilities first provided by AT&T in UNIX® System V, Release 3.0. It provides the UNIX networking developer a new means of implementing Input/Output processing. STREAMS is a general, flexible facility which includes a set of tools for development of UNIX system communication services. Adherence to STREAMS architecture encourages modular software design and provides a standard mechanism and interface at the system call level and within the UNIX system kernel. STREAMS provides a powerful framework for development with a user interface consistent with the existing UNIX character I/O interface.

NCR has provided its customers with X.25 networking solutions for several years. The portable X.25 base is successfully marketed on the Tower®, ITX, and VRX product lines. With the announcement of AT&T's UNIX STREAMS architecture, another environment is identified which enhances the usability and performance of NCR's X.25 networking facilities.

Before we embarked on the Standard Development Process, a prototype X.25 STREAMS driver was developed. The purpose of this prototype was to provide the development staff with insight into the technical issues associated with the new STREAMS architecture. The lessons learned from the prototyping effort were then applied to the planning, design, and implementation of the final product.

Motivation

The primary motivation for porting X.25 to the STREAMS environment is the need to support a STREAMS-based OSI Networking Layer. With the current implementation of X.25 on the UNIX Tower, a considerable amount of process switching and data movement is required when sending or receiving messages. This becomes more apparent when you consider the new OSI Networking Services, which are implemented in STREAMS. Figure 1 depicts the components of the existing implementation.

With this configuration, the OSI application uses a standard STREAMS system call to pass a message to the OSI Network Services STREAMS driver. Over

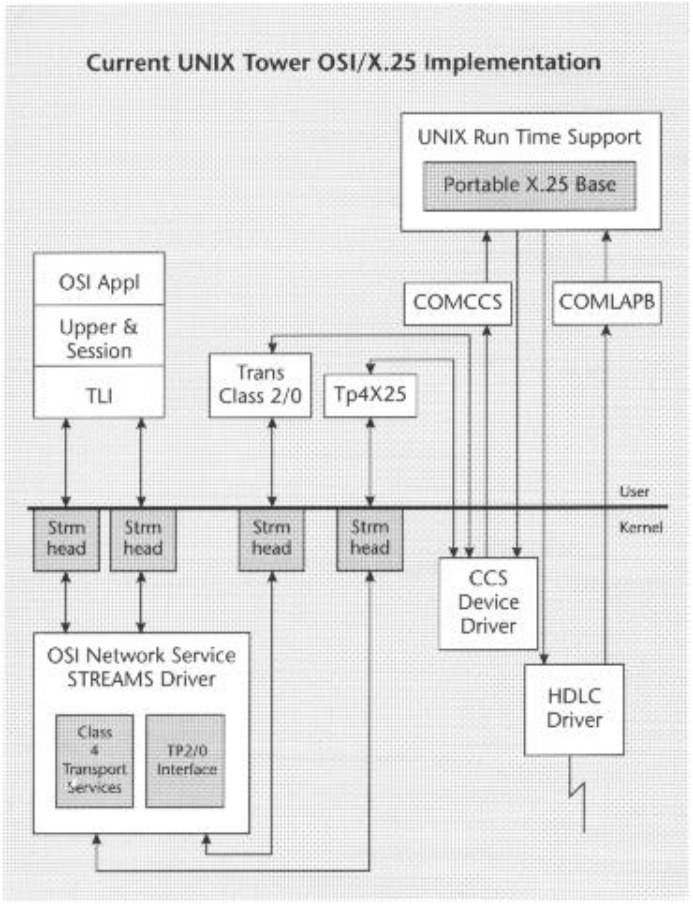


Figure 1: With the current implementation of X.25 on the UNIX Tower, a considerable amount of process switching and data movement is required when sending or receiving messages. This becomes more apparent when you consider the new OSI Networking Services, which are implemented in STREAMS.

X.25, an OSI application may use Class 4 Transport Services or the TP 2/0 Interface. Depending on which service is chosen, the appropriate STREAM to the OSI driver is used. The message is copied from user space to STREAMS message blocks at the Streamhead.

After the appropriate processing is performed, the OSI Network Services STREAMS driver passes the message via STREAMS facilities to one of two processes executing in user space (i.e., TRANSCLASS 2/0 or TP4X25). The STREAMS message blocks are copied back to user space at the Streamhead.

The TRANSCLASS 2/0 and TP4X25 processes request services from X.25 by issuing a standard I/O request to the CCS pseudo-device driver. The message is copied from user space into kernel space during this system call. The CCS driver passes the message, which is still destined for X.25, to a separate process, COMCCS. Again, the message is copied from kernel space to buffers in user space. COMCCS then transfers the message to the X.25 process via a queue, which is managed by the UNIX Run Time Support operating system dependent code. The message is sent by X.25 to the HDLC driver for transmission on the link using a standard I/O request, and the message is again copied from user space to kernel space before being transferred across the physical link to the network.

A message from the link bound for an OSI application follows nearly the same path in reverse.

With the X.25 functionality integrated into the STREAMS architecture, a more efficient configuration is possible. The ability to provide X.25 processing in the kernel eliminates the need for much of the process switching and message copying currently required. Figure 2 depicts a pure STREAMS implementation.

The interface between the OSI Networking Services and the X.25 software is transformed into STREAMS message blocks and those facilities provided by STREAMS to manipulate them. The interface between the X.25 software and the HDLC driver is also converted to STREAMS message facilities. Message copying occurs once at the Streamhead.

Assumptions about STREAMS

Realizing that the facilities of STREAMS would simplify the current OSI/X.25 implementation and possibly improve the performance of X.25 in general, we began researching the features of STREAMS (see Figure 3). After completion of this preliminary study, we made several assumptions based on our research.

STREAMS provides a standard interface and an overall strategy for implementing communications products on UNIX. The user interface provided by STREAMS incorporates the advantages of the existing character I/O subsystem. A consistent interface to all I/O devices is presented to the user ("open," "close," "read," "write," "ioctl"). Additional interfaces ("getmsg," "putmsg," "poll") are provided.

STREAMS provides a standard programming environment for the UNIX driver developer. This eliminates the need to develop common utility routines and should result in a savings in driver code space.

STREAMS provides a high-performance buffer-management mechanism. The message-handling mechanisms provided by the STREAMS architecture eliminate the necessity to move data during message processing. The facilities provided by STREAMS to manage the message buffers are flexible yet efficient.

STREAMS eliminates one of the disadvantages of the existing I/O subsystem by providing dynamic configuration. The user program can dynamically configure additional services not supplied by the device driver. This is accomplished by "pushing" modules onto a protocol stack in a "pipelike" fashion to provide various protocol layers.

The modular nature of STREAMS allows various functions to be provided as separate modules independent of the device drivers. This leads to a large amount of portable code that can be shared across diverse platforms and networking configurations. This modular architecture can be used to reflect the layering characteristics of contemporary network architectures, and complements the existing portable communications products produced by NCR.

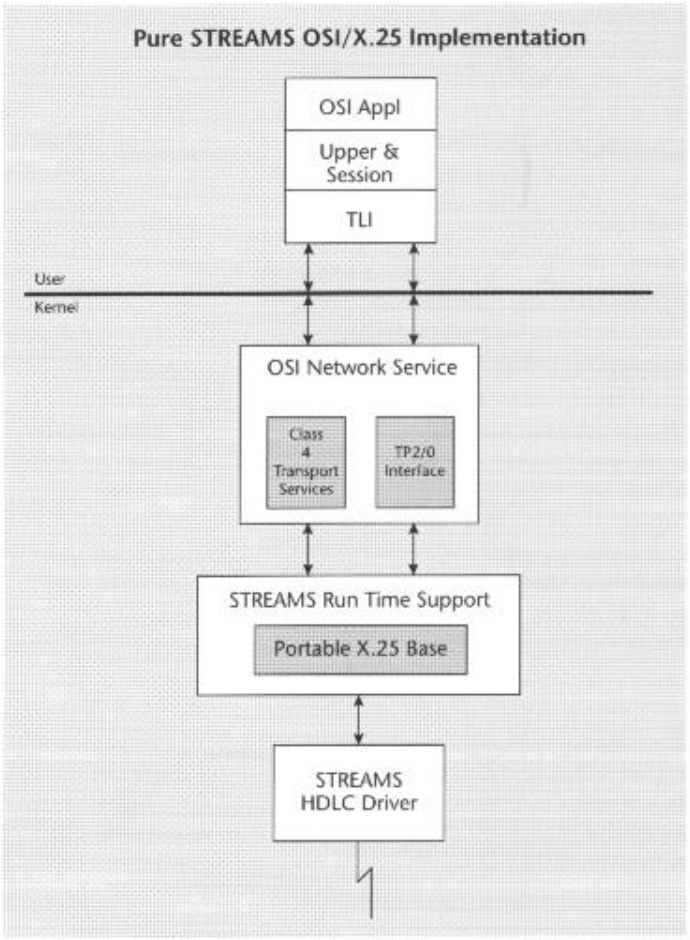


Figure 2: The ability to provide X.25 processing in the kernel eliminates the need for much of the process switching and message copying currently required. This figure depicts a pure STREAMS OSI/X.25 implementation.

Prototype Development

With these assumptions in mind, we began our port of the X.25 base software to STREAMS. Our goal was an X.25 DTE prototype, which could demonstrate message exchange. Our philosophy was to validate the architecture as quickly as possible. We saw the prototype as disposable, and applied cursory design to its implementation.

An existing X.25 test application was used to ensure that it was possible to issue a call and perform data transfers to a remote DTE using a DYNAPAC Multi-Switch. Only the device name used with the "open" command required modification. Minimal changes to the test application were required to run in the STREAMS environment.

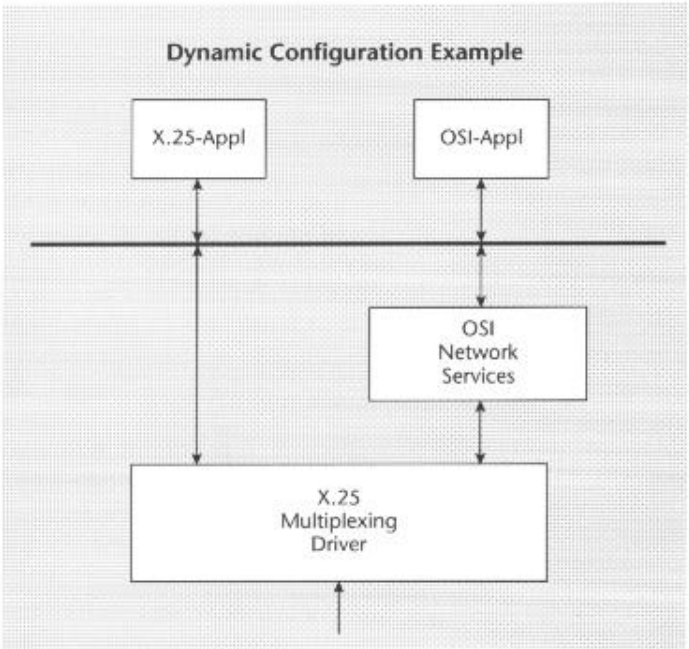


Figure 3: Dynamic Configuration Example.

In order to expedite the prototyping effort, a large amount of error and exception-handling code within the X.25 base was circumvented. We assumed the physical link to the X.25 network would always be up. Shutdown processing was considered unessential. Timing facilities of the X.25 base were disabled. The prototype did not need to handle large data flows, so the STREAMS internal flow control mechanism was not utilized.

The existing UNIX Run Time Support code, URTS (see Figure 1), was the basis for the prototype operating system-dependent code. Only those changes required to provide a STREAMS X.25 driver with "demo" capabilities were made. The resulting STREAMS Run Time Support code, SRTS, and the portable X.25 base were linked with the UNIX kernel.

This linking process was difficult and frustrating. It exposed the presence of

duplicate variable names shared by our X.25 software and the operating system. This, of course, was never an issue when the X.25 software resided in user space. Of greater concern was the fact that common UNIX service routines ("memcmp," "memcpy," "time," "scanf," "sprintf"), available in user space, were no longer available for the kernel-based implementation. Initially, stubs were added to resolve references. Eventually, new code was incorporated into the driver to perform this needed functionality.

At the beginning of the prototyping effort, our organization lacked experience in UNIX kernel development. We were communication developers accustomed to life in user space and were accustomed to receiving error messages and an occasional core dump when mistakes were encountered. We were faced with system crashes and the 68000 instruction set. The rules and tools associated with kernel development were not readily apparent. Trial and error with the Kernel Debugger (KDB) was our only recourse.

In this manner, we found that attempts to allocate memory for internal data structures resulted in system crashes. The important functionality of "malloc" and "free" would have to be implemented within the X.25 driver.

With previous development efforts, machine resources were available in multi-user mode. This luxury was lost with kernel development. Not only was it necessary to re-link the kernel and reboot (often several times per day), but programming errors resulted in system crashes.

A major goal of our prototype development was to identify the issues relevant to the final X.25 STREAMS driver. By the time our prototype was running, we had a very good idea of what would be required.

Implementation Details

The basic design of the X.25 STREAMS prototype is relatively simple. Since X.25 accepts and processes input and output for multiple applications, it is implemented as a STREAMS Multiplexing driver (Figure 4). By definition, a STREAMS multiplexer is a pseudo-driver with connected multiple STREAMS.

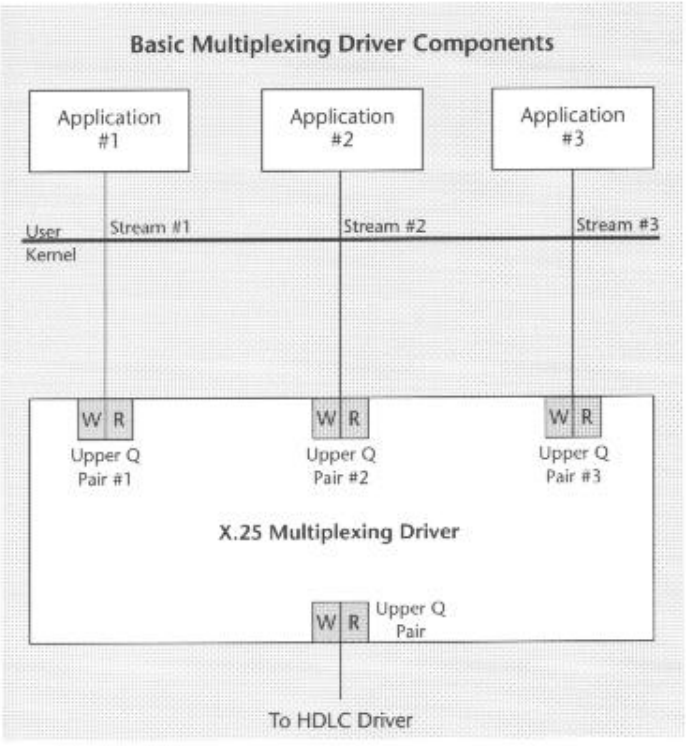


Figure 4: The basic design of the X.25 STREAMS prototype is relatively simple. Since X.25 accepts and processes input and output for multiple applications, it is implemented as a STREAMS Multiplexing drive.

Multiple STREAMS are connected above a driver by multiple open calls. Each distinct STREAM provides a pair of queues, which the applications use to communicate with the multiplexing X.25 STREAMS driver. The close system call dismantles the STREAM. STREAMS requires that all multiplexer drivers contain special developer-provided software to perform the multiplexing data routing and to handle flow control.

To summarize the X.25 open and close processing of the STREAMS Run Time Support (SRTS) code, a minor device number is assigned to each application by "x25open" when it issues an open to the X.25 Multiplexing driver. A global data structure ("x25-mux") is used to correlate the upper queue pair of the newly opened STREAM to the application and its minor device number. When the application issues a close system call, the "x25close" procedure clears the associated "x25-mux" entry.

The lower queue pair of the X.25 driver is used to transfer messages to and from the HDLC STREAMS driver. Since this driver was being developed in parallel and was unavailable, a loopback driver routed messages to user space where the existing HDLC driver was accessed. Figure 5 illustrates this.

To provide the connection between the loopback driver and the X.25 Multiplexing driver, a daemon process opens the multiplexing and the loopback drivers using two separate open system calls. The daemon then links the drivers together using the STREAMS IOCTL-Link facility. This process remains active to "hold" the X.25 Multiplexer-to-Loopback configuration together.

To complete the prototype configuration, a user process, PROCLAPB, issues an open to the loopback driver and the HDLC driver. This process issues the appropriate commands to the HDLC driver to connect the link. A link-up indicator is sent to the X.25 multiplexing driver through the loopback driver. The process then waits for a message either from the link or from the STREAM and passes it on in the appropriate direction.

Once the drivers are opened and linked, the X.25 communication

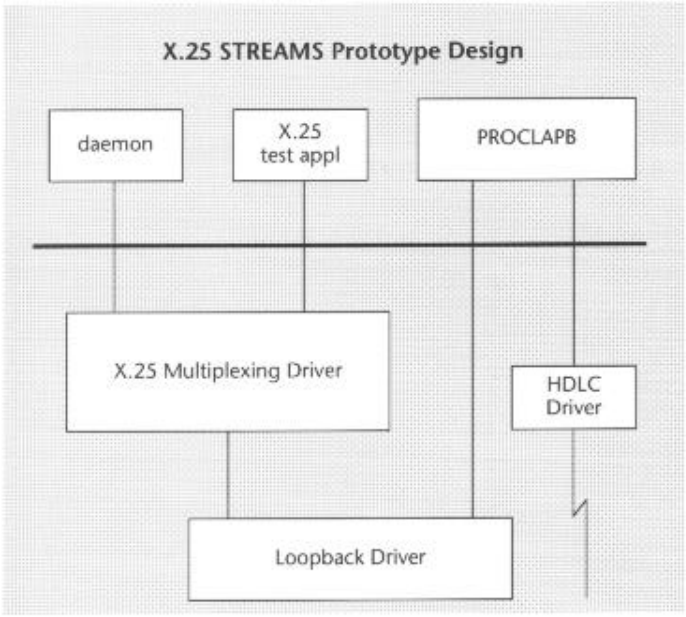


Figure 5: The lower queue pair of the X.25 driver is used to transfer messages to and from the HDLC STREAMS driver. Since this driver was being developed in parallel and was unavailable, a loopback driver routed messages to user space where the existing HDLC driver was accessed.

subsystem is initialized. The initialization process involves reading a parameter file from disk and building the appropriate data structures which describe the network to the X.25 base. Since file access is not possible from the kernel, a STREAMS service interface between the Multiplexing X.25 STREAMS driver and the daemon process is used. This service interface allows the STREAMS driver to make file requests to the daemon process. The daemon process performs the actual file I/O and then passes each record to the driver by putting a message on the STREAM.

Messages processed by the X.25 STREAMS Multiplexing driver are received from two sources. Messages may come from the X.25 application, which uses a read/write interface. Messages may also be received from the HDLC link via the putmsg/getmsg interface used by the PROCLAPB process. To facilitate this architecture, two distinct STREAMS "put" procedures and one central service procedure are defined.

Messages from applications are received by the "x25uwput" procedure within SRTS when the write system call is issued. To identify this as a message from an application, "x25uwput" appends a small message block to the front of it. This string of message blocks is placed on the queue for processing by the central STREAMS service procedure, "x25srv."

Message blocks received from PROCLAPB are initially processed by the "x251rput" procedure. The putmsg interface used by PROCLAPB guarantees that a small message block, indicating this is a message from the link, already precedes the actual message. Therefore, "x251rput" places this message directly on the queue for processing by "x25srv."

The X.25 Multiplexing driver service procedure, "x25srv," is automatically invoked when a message is placed on any of its queues. This service procedure removes and processes all messages on all of the queues it services before terminating.

When "x25srv" receives a message, the message-type indicator in the first message block is examined. If the message is from an application, it is copied into a buffer global to the X.25 driver. The portable X.25 base procedure, DPN_APPL_SEND, is then called to process the message. If a message is received from the link, it is copied into a global buffer and the portable X.25 base procedure, DPN_LINK_INPUT, is called.

When messages must be sent to a particular application, "x25srv" identifies the appropriate STREAMS queue through the global "x25_mux" data structure. A STREAMS message block is allocated and the outgoing message is copied into it. The STREAMS "putnext" procedure is called to place the message on the appropriate Streamhead.

When messages must be sent out on the link, "x25srv" allocates a message block and issues the "putnext" STREAMS procedure to place the message on the loopback driver's queue. The loopback driver then passes the message to PROCLAPB, which calls the HDLC driver to send it out on the link.

Conclusions

The development of the X.25 STREAMS prototype enabled us to verify our original assumptions about STREAMS. Nearly all proved to be true. Significant performance improvements were realized in the final product. STREAMS architecture does simplify communications solutions by providing a consistent interface standard. The ability to tailor protocols by pushing various modules onto the STREAM provides flexible networking solutions. The modular nature of this architecture is a consistent approach for NCR's communications products.

However, STREAMS did not address all of our networking driver needs as claimed. STREAMS provided a mechanism to allocate and deallocate message blocks; however, other types of memory management became our responsibility. UNIX V.4 has addressed these problems with more comprehensive memory management facilities. Required UNIX subroutines were also unavailable. This necessitated our developing additional subroutines to provide this functionality.

Prototyping was a valuable investment of our resources. It provided us with an opportunity to make mistakes and learn from them. The planning and design of the final product was easier because of this exposure.

References

1. AT&T UNIX System V, STREAMS Primer, 307-229, Issue 1, 1986.

2. AT&T UNIX System V, Release 3, *STREAMS Programmer's Guide*, 307227, Issue 1, 1986.
3. "STREAMS Technology," a paper by Gilbert J. McGrath, AT&T Information Systems.
4. NCR Tower 32, Networks — X.25, D1-1080-A02, January, 1988.

Pat Crosthwaite graduated with a B.A. degree in Mathematics from San Diego State University. He joined NCR in 1974.

Mary Frances Vigil graduated with a B.S. degree in Mathematics from Arizona State University. She joined NCR in 1981.

Most recently they have both been working in the Communications Network Software Department at SE-San Diego and have been involved with a variety of communications products.