x-kernel Evaluation at the OSF RI

Franco Travostino (travos@osf.org)
Franklin Reynolds (fdr@osf.org)
OSF Research Institute
One Cambridge Center
Cambridge, MA 02142

September, 1993

Abstract

*The x-kernel, developed at the University of Arizona (UofA), is an object-oriented framework for the implementation of communication protocols. It was designed to **a**), facilitate the development of efficient protocols, and **b**), make the protocol development cycle itself efficient. The OSF Research Institute has monitored the UofA's work since 1991 and has acquired "hands-on" x-kernel experience by building and testing experimental MACH 3 kernels that incorporate x-kernel technology. Recently, the Research Institute has examined the x-kernel with an eye towards using it as the core architecture for our development of NORMA IPC, distributed real-time, and high availability protocols. This report presents our evaluation, enumerating the x-kernel's strengths and weaknesses with respect to our requirements, and briefly describes several tasks that the RI would undertake to integrate the x-kernel into our technologies.*

1 Overview of the x-kernel

With respect to performance, the most important component of any distributed system is its network software. This software manages the system's communication hardware and enables programs executing on separate nodes to communicate with one another.

Network software is necessarily complex. In addition to hiding the machine-specific aspects of the communication hardware from the rest of the system, network software is responsible for delivering messages to their proper destinations, delivering the messages in the correct order, encoding and decoding transmitted data, and responding reliably to transmission failures.

Historically, network software has been composed of multiple *protocol* layers, each of which provides specific functionality with respect to data transmission. A given protocol layer is complex in that it implements many operations. On conventional architectures, several protocol layers are statically configured into a protocol stack.

The x-kernel is an object-oriented framework for developing efficient protocols. Within the x-kernel, conventional protocol layers are decomposed into primitive building blocks known as *micro-protocols* and *virtual protocols*. Developers configure these objects into structures referred to as *protocol graphs***.** Micro-protocols perform operations on messages and virtual protocols route messages through the protocol graph.

Micro-protocols and virtual protocols are static objects that are configured into the protocol graph before the system is built. The protocol objects have capabilities for one another and developers configure given instances of the x-kernel to arrange the protocol objects into a protocol graph that

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

meets the requirements of the system's applications. Figure 1 depicts a simple x-kernel protocol graph. The rectangles represent micro-protocols, the diamonds represent virtual protocols.

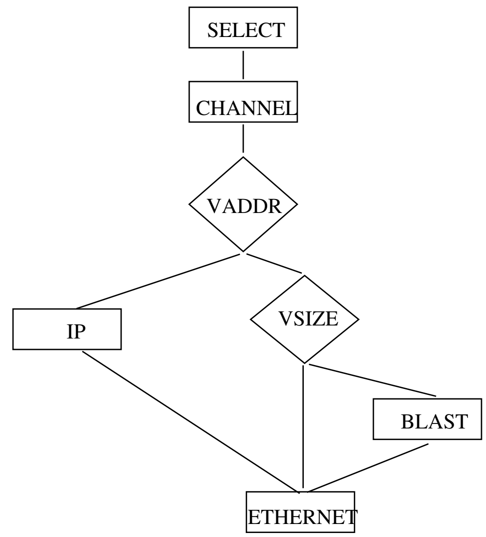


Figure 1. An x-kernel protocol graph

When an application opens a network connection, the x-kernel represents the connection with a set of dynamic objects referred to as *sessions*; each of a connection's session objects is mapped to a protocol object.

X-kernel messages flow through the protocol graph in the context of a single thread and so are transmitted very efficiently. When the networking hardware issues an interrupt to deliver a message, the kernel dispatches a kernel thread to *shepherd* the message through the protocol graph. The thread begins by invoking the bottom anchor protocol. The protocol dispatches the appropriate session object to process the message. When it has finished processing the message, the session object invokes the next protocol in the path, which dispatches the next appropriate session object, and so on.

For a more detailed description of the x-kernel's current design and features, please refer to [2].

x-kernel Features

The x-kernel includes the following interesting features:

- The protocol-specific code is well separated from the protocol-independent code.
- x-kernel protocol graphs can be extended across multiple domains without changing the protocol semantics.
- The system provides a powerful set of tools for developing and configuring new communication protocols
- The UofA claims that it outperforms most existing implementations of common communication protocols.

- It is free of software encumbrances.

1.1 The OSF RI and the x-kernel

Since the spring of 1992, we have periodically evaluated the x-kernel technology. We have interacted with the UofA and have made various proposals with respect to the x-kernel's evolution. We are also interested in other advanced research topics that are being investigated at the UofA, particularly their work in the area of fast communication software. These topics include efficient techniques for moving data across domain boundaries [3], and the support of next-generation network controllers [4].

2 x-kernel 3.2

The University of Arizona released the latest revision of x-kernel 3.2 in February of 1993 and has made the technology available via anonymous ftp. The latest revision includes the following components:

- The protocol-independent x-kernel framework.

- A set of 25 OS-independent and domain-independent protocols. This set includes the INET protocol family, SPRITE RPC, Blast (a protocol that manages massive transfers, with selective frame retransmission), peer reboot detection protocols, and several virtual protocols (VSIZE, VNET, VMUX). The UofA recently completed a group of protocols that implement MACH 3 remote IPC; a protocol graph based on these protocols will replace CMU's NetMsgServer.

- A MACH3 installation kit, which can be used to configure the x-kernel as a user-level task or as a MACH 3 kernel subsystem (CMU MK74, DecStation 5000). An installation kit for the SunOS is also available.

In the course of our investigation, we built and executed the x-kernel in both the user-level and kernel subsystem configurations, most recently on a norma 14 kernel. We ran tests on several protocol stacks, including the traditional INET stacks and the more complicated xRPC protocol stack, a stack which integrates the SPRITE RPC, Blast, and Boot ID protocols.

With respect to OSF's possible use of x-kernel technology, our investigation turned up the following major weaknesses:

- The x-kernel does not attempt to control resource allocation

- The x-kernel does not support concurrent execution; only one thread at a time is allowed to shepard a message through the protocol graph.

These deficiencies were promptly reported to the UofA and have been discussed at length, via the e-mail forum *xkernel@cs.arizona.edu,* with the members of the x-kernel community (the x-kernel community includes the UofA, MIT, Uppsala University, Chorus, CMU, ISIS, TIS, and the OSF RI). The community as a whole now understands the deficiencies and possible solutions, and work is continuing. For example, Mats Bjorkman of Uppsala University has recently been able to execute a prototype x-kernel (the so-called *xx-kernel*) that implements some level of concurrency.

As mentioned above, the current version of the x-kernel includes micro-protocols that can be used to implement MACH3 NORMA IPC. Currently, the OSF RI is modifying these micro-protocols so that they adhere to the changes we have made to the MACH3 IPC facilities, the most notable of which is the DARPA-sponsored interface unification project that resulted in the conversion of typed IPC to untyped IPC.

3 Network-related Projects at the OSF RI

One of the key components of the MACH operating system is its IPC subsystem. The OSF RI research agenda includes several network-related projects that are based on extending MACH 3 IPC semantics over communications media. This extension, referred to as NORMA IPC, is the basis for implementing MACH 3 on multicomputer platforms and single image clusters.

3.1 NORMA IPC

In earlier versions of MACH, the IPC subsystem was extended over the network via user level tasks such as the NetMsgServer. Multicomputer environments require IPC performance that exceeds the capabilities of a user-level task. Consequently, NORMA IPC is implemented within the kernel. Currently, NORMA IPC is fully operational and available in releases from CMU and OSF. OSF/1 AD, an operating system for multicomputers, is based on NORMA IPC, and is being shipped on Intel Paragon Supercomputers.

The RI's work with NORMA IPC has revealed several problems with its current design [7], including the following

- Partial failure semantics for IPC are not well defined. The current implementation does not detect or behave well in the presence of failures.
- Nodes within a domain must be booted in a prescribed order.
- Communication between different NORMA domains is not supported. Neither are disconnected operation, standard network transport protocols, heterogeneous domains, real-time, multiple network interfaces, multiple transport protocols, network bridging, routing, or network topology changes.

In order for MACH IPC to serve as a reasonable technology for the construction of distributed services, remote IPC must provide flexible, robust, efficient service in a networking environment. This means that the semantics of IPC must be modified to provide better support for distributed computing and a powerful and flexible framework for networking services must be implemented.

Consequently, we will produce a second generation NORMA IPC that provides an efficient, modular, and easy to configure version of the subsystem. This project is referred to as "the NORMA IPC rewrite."

3.2 Additional Projects

In addition to the NORMA IPC rewrite, the RI has the following network-related projects on its agenda:

- Distributed RealTime IPC - The real-time project will add real-time behaviors to Mach IPC. In addition to modifying local and remote IPC semantics, the project will investigate reliable real-time network transports and other real-time protocols (such as node-alive, timely notification protocols and low latency, low drift distributed time protocols).
- Clusters - Message passing parallel computers are similar to a collection of computers connected by a high speed, local area network. It should be possible to extend OSF/1 AD to work

on a collection of closely cooperating computers, perhaps workstations, connected via a local area network. Our investigation of cluster configurations will include work in the area of fault tolerance.

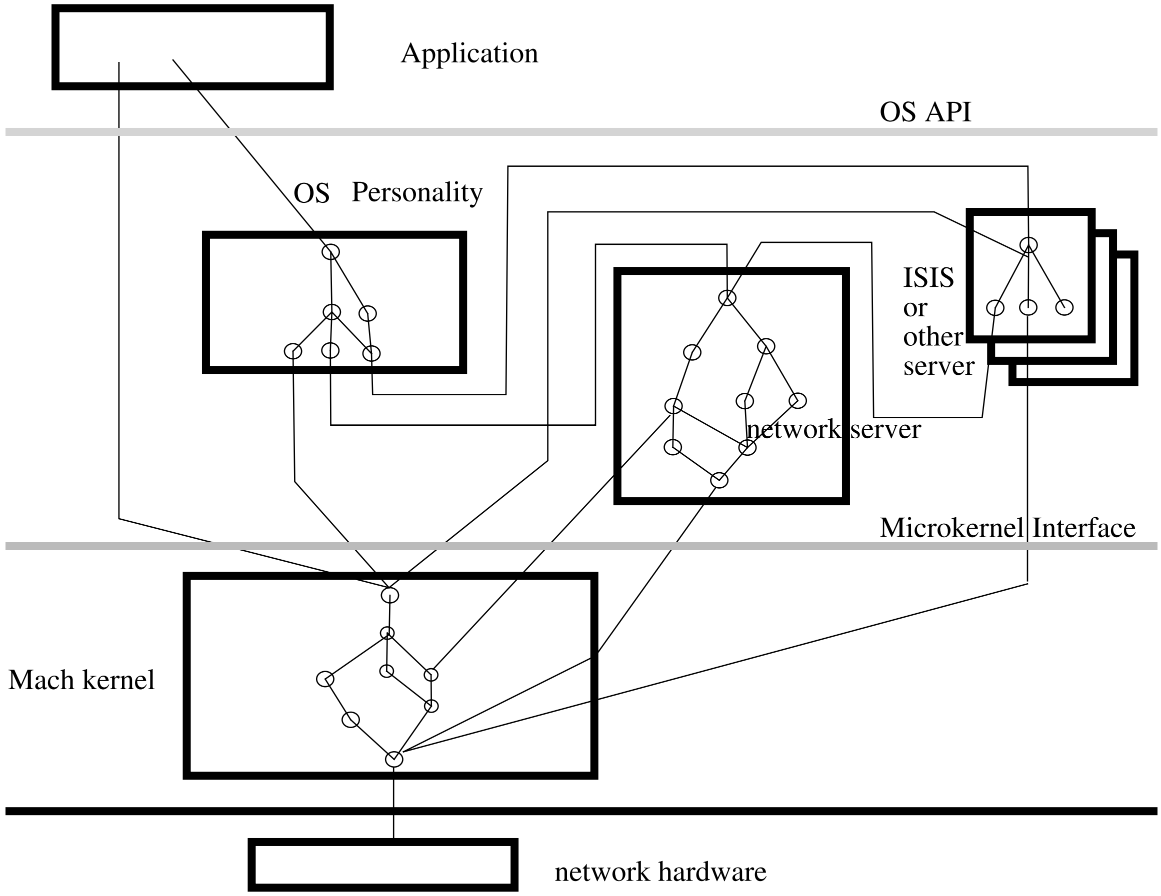
- Scalable I/O Initiative - ARPA has initiated a high speed I/O project as part of their supercomputer program. Mach, OSF/1 AD and the x-kernel play significant roles and will be integrated as part of this initiative. The goal of this project is to provide very high performance networking in a Mach system using the x-kernel.

- ISIS - ISIS is an ARPA-sponsored high availability technology. It is used commercially and by several ARPA sponsored research programs including Hiper-D (a program OSF is very interested in). ISIS is being integrated with Mach and the x-kernel by the ISIS group at Cornell.

- The research community has recently expressed some interest in being able to bind network protocols to MPP applications; thus, the protocol subsystem could be further fragmented in several instances, one per address space [8].

- The RI is interested in the networking requirements of OS personalities that are implemented as user-level servers. We are looking for a network protocol framework that will allow us to provide network services in a uniform and flexible manner to multiple OS personalities.

Although the current version of the x-kernel lacks several features that are required for our networking development, it is nonetheless a strong candidate on which we can base our work.



It provides a programming model and a collection of tools that simplifies the modular design, implementation and maintenance of protocols without unduly sacrificing performance. Our next generation network architecture will consist of several components and must provide very high performance yet be flexible enough to permit a wide variety of experiments.

Our proposed network architecture divides the components at functional boundaries. The OS personality server implements personality-specific abstractions, for example, sockets. The kernel implements local and remote IPC semantics. The network server implements personality-independent protocols, such as TCP, SNMP, node-alive, etc. Other network services such as ISIS, distributed time, or decentralized name services are implemented in other servers.

The x-kernel is used to implement protocol graphs within each component and to connect to components to one another. Protocol objects can be transparently invoked across server boundaries, though there is a performance penalty. Co-locating servers [13] dramatically reduces the cost of crossing the server boundary. The ability to spread protocol graphs across multiple servers and the ability to co-locate servers provides the system designer with considerable flexibility. Protocols or portions of protocols can be moved to or duplicated in different servers depending on the performance requirements.

4 The x-kernel's Strengths

Although the current version of the x-kernel lacks several features that are required for our networking development, it includes several features that make it a strong candidate on which to base our future communication projects. We first enumerate those aspects of the x-kernel that we find especially appealing. In the next section we describe those aspects of the x-kernel that will need to be addressed to meet our requirements.

4.1 Protocol-Independent Framework

The x-kernel provides a protocol-independent framework that promotes the development and dynamic configuration of heterogeneous network services. The framework will give us a consistent development environment in which to pursue our diverse network-related projects and will allow us to seamlessly integrate each project into the system upon its completion without adversely affecting the integration of other projects.

4.2 Communication System Development

The x-kernel provides a powerful set of tools that facilitate the development and configuration of communication systems. Currently, the x-kernel system includes several protocols that can be used ***a***) in assembling new protocol graphs, and ***b***) as a basis for creating new protocols. The x-kernel also provides a rich toolkit that developers can use to implement new protocols. These tools allow programmers to reuse existing code and write encapsulated and easily maintainable code. The x-kernel will allow the RI to pursue parallel lines of development with respect to our network-based projects.

4.3 Cross-Domain Potential

The x-kernel can be built in both user space and kernel space without significant differences. The protocol-independent framework may have a few compile-time switches that accommodate the differences in how utilities are invoked from user space versus how they are invoked in kernel space. The technology proposed by the UofA to get efficient cross-domain transfers (fbufs [3]) might provide an excellent solution. We plan to investigate this technology further**.**

4.4 Flexible Configurability

Given the x-kernel's emphasis on modularity, it has a high potential with respect to flexible configurability. Today, developers statically define the topology of an x-kernel protocol graph via a meta language. It is very easy to change a protocol graph at build time (to accommodate, for instance, a change in the underlying network hardware). The x-kernel also allows developers to configure protocol graphs that support multiple heterogeneous network interfaces. For example, a developers could create a unique flow of protocol processing by configuring a protocol graph that merges an unreliable stack over FDDI with a reliable stack over a mesh**.**

The modularity inherent in the x-kernel allows us to cleanly separate machine-dependent from machine-independent code, much as is done with the MACH kernel. We feel that this feature, which has had a profound impact on MACH's success to date, will have a similar effect on our forthcoming network-related projects. This type of modularity is in keeping with the OSF goal of producing platform-independent software.

In the long term, we would like to enhance the x-kernel's configurability by providing support for the dynamic configuration of protocol graphs. In the context of readably available software, we believe that system administrators should be able to change the system's networking functionality (e.g., by simply interposing a new layer) without affecting the uptime of the machine.

4.5 The x-kernel Community

The x-kernel is free of any encumbrances. Consequently, the x-kernel community is growing. As a member of this community, the RI expects to have access to a large number of protocols from both the research community and industry. Our interactions with the UofA have been cordial and we believe that the RI can play a significant role in the x-kernel's evolution. Our involvement with the x-kernel will expose us to the latest topics of academic research, topics such as Integrated Layer Processing (ILP) and language-based approaches to protocol development.

5 x-kernel Shortcomings

This section describes those aspects of the x-kernel that will need to be addressed in order for the technology to meet our requirements.

5.1 Concurrency and Real Time Issues

OSF/1 AD is targeted for multicomputer platforms and some of these platforms are configured with nodes that are UMA multiprocessors. Therefore, we require a protocol framework and associated protocols that can run with the highest level of concurrency supported by the underlying hardware. Today, the x-kernel is MP-safe in that operations are funneled via a monitor: only a single thread at a time is allowed to shepard a message through the protocol graph. If we adopt the x-kernel technology, we will need to strengthen its concurrency mechanisms significantly. Section 8.1 describes this projected task in more detail.

On a related topic, the RI's MACH 3 real time project will produce a preemptable MACH 3 kernel; the x-kernel will need to accommodate preemption and multiple scheduling policies as well.

5.2 Resource Issues

As mentioned earlier, the current version of the x-kernel does not include any resource management mechanisms and policies. There are two types of resources future versions of the x-kernel must be able to manage: resources consumed by transient traffic (messages), and resources consumed by the x-kernel system itself.

The existing in-kernel version of the x-kernel assumes that memory allocation never fails and so makes no provision for memory shortages. Furthermore, when the system overallocates memory from the fixed-size buffer pool at the bottom anchor protocol, the x-kernel silently drops packets and does not attempt to garbage collect or look for more buffers. If we adopt the x-kernel technology, we will need to provide mechanisms to reserve, allocate, garbage collect, prune, and deallocate resources. Section 8.2 provides more information about our projected resource control work.

In addition to the resources used for transient traffic, we need to account for the resources used by the x-kernel's static objects (protocols) and dynamic objects (sessions). Currently, both object types are instances of a unique, primitive x-kernel object (XObj). On 32 bit architectures, every instance of this object consumes 152 bytes. Given that configurations of our systems commonly allocate 10,000 registered ports and hypothesizing a 1:1 association between MACH 3 ports and x-kernel sessions, using XObjs to represent session objects would increase our allocation of wired memory by 1.5 MB. Previous versions of the x-kernel provided separate objects for protocols and sessions; we may consider using a separate, and much smaller object for x-kernel sessions.

5.3 Code Conversion

Currently, the x-kernel source code is written in the K&R C language. The kernel baseline at the OSF RI is already converted to ANSI C and includes ANSI C prototypes; therefore, the x-kernel needs to be upgraded to ANSI C. In the medium term, we plan to rewrite some kernel subsystems in C++. Given that the x-kernel's design is already based on the object-oriented paradigm, conversion of the x-kernel to C++ seems like a natural step.

6 Issues that Require Further Investigation

The x-kernel's model of protocol decomposition has raised concerns about its performance as compared to the performance of conventionally implemented protocol stacks. The UofA claims that micro-protocol overhead is minimal and that the use of virtual protocols allows many common network operations to skip sections of the protocol graph at runtime, thus negating the overhead introduced by the presence of the micro-protocols. We must verify these assertions with empirical data.

7 Integration Issues

The integration of the x-kernel technology with the MACH 3 kernel has to be seamless. By this we mean that the x-kernel should use the MACH kernel's facilities for locking structures, performing queueing operations, and allocating memory.

As part of the integration with MACH, we also propose to make x-kernel sessions first class MACH 3 objects (other first class objects include tasks and threads). Each session object that sits next to a virtual domain boundary will be associated with a MACH 3 port and will be invoked from other virtual domains via MACH 3 IPC messages (and through a MIG-based interface). In this manner we can use existing MACH 3 semantics to extend protocol graphs across domain boundaries.

8 Major x-kernel Tasks

If the OSF RI adopts the x-kernel technology, we will have to carry out three major tasks in order to make the technology work for us. These tasks are as follows:

- Add SMP support to the x-kernel
- Add resource control mechanisms to the x-kernel
- Add support for multiple classes of traffic to the x-kernel.

The following subsections will describe these tasks in detail.

8.1 SMP Support

There already exist communication protocol frameworks that have been modified to support concurrency in symmetric multiprocessing environments. For example, the OSF/1 operating system implements a version of STREAMS that supports both implicit and explicit locking mechanisms [9]. Under the implicit locking scheme, protocol modules are not concerned with MP issues; synchronization is affected via message queues. When a protocol module finishes processing a message, it places the message on the next module's message queue and returns. Under the explicit locking scheme, protocol modules use fine grain locks. The OSF/1 implementation of STREAMS runs on different MP platforms, scales well, and performs well.

While we would like the SMP support for the x-kernel to perform and scale as well as that of STREAMS, there are two behaviors specific to the x-kernel that will make our work particularly challenging:

1. A given message is shepherded through the protocol graph by a single thread. When the thread calls from one module (call it Module A) down to another module (call it Module B), and that call returns, the thread continues protocol processing in Module A under the

assumption that all actions defined in Module B and all modules below Module B, have been executed.

2. Threads pursue complex paths through the protocol graph. Consider the situation where the completion of a message-receive operation requires the sending of an acknowledgment message. This is a case where a send operation is nested within a receive operation. When the message is received, the x-kernel assigns the message a thread at the bottom anchor protocol and this thread shepherds the message up to the source of traffic via the intervening modules. Then, from the source of traffic, the thread carries out the acknowledgment operation by descending through the modules back to the anchor protocol. From the bottom, the thread begins to unwind its stack, going up the protocols first, and then back down them. By the time the thread reaches the anchor protocol for the last time, it will have traversed the protocol graph four times.

These two aspects of the x-kernel show that the design and implementation of SMP support must be careful to avoid deadlocking and significant lock contention between protocol modules while at the same time preserving the x-kernel's performance.

We have discussed with the UofA three complementary approaches to the SMP problem:

- Concurrency without SMP by statically partitioning the protocol graph among processors.
- Implicit locking model.
- Explicit locking model.

We have rejected the static partition model for the following reasons:

- Forcing a given protocol module to execute on a specific processor means that the performance of that module will be adversely affected when that processor is under heavy load.
- Protocol graphs tend to funnel traffic coming from different sources to a single point (the network driver). The static partition model does not scale well with large fan in and large numbers of processors.

Ideally, our locking model would provide both implicit *and* explicit mechanisms so that protocol developers could choose the mechanism that best suits their needs. Unfortunately, the implicit locking mechanism will not work well in the x-kernel. The x-kernel's flexibility allows developers to create many different paths through the protocol graph; in order to provide MP protection for the most general of protocols, the protocol-independent framework would have to take on the order of twenty locks each time a shepard thread invokes a new protocol object.

Consequently, we have decided to base our SMP support on the explicit locking model only. In fact, Mats Bjorkman, with his work on the xx-kernel, has already explored some of the issues inherent in explicit locking and has generated a set of rules that can be used to implement a deadlock-free x-kernel [6].

We are confident that we can implement a fully general SMP solution that is of production quality. The milestones for our effort are as follows:

- Identify areas within the protocol-independent code and protocol modules code that need protection and specify a locking hierarchy with respect to these areas.

- Document rules for protocol developers. In the course of establishing locks for the existing protocols, we will formulate locking rules that are general, necessary, and sufficient.

Our SMP work with the x-kernel will assume a UMA, fully symmetric hardware base that scales to 32 processors.

Integration with a Preemptible Kernel

The OSF RI development agenda includes the MACH 3 Preemption Project. When this project is completed, we will make the x-kernel's lock types consistent with the types established by that project. The integration of the x-kernel with a preemptive MACH 3 kernel will allow us to implement a priority-based message semantic that will support expedited messages by assigning each x-kernel shepard thread a scheduling priority based on the type of expedited message the thread is taking through the protocol graph.

8.2 Resource Control

As pointed out earlier, the lack of resource control mechanisms is a serious problem in the current version of the x-kernel. Our requirements mandate the implementation of resource control in the x-kernel. We are not the only party interested in solving this problem; resource control is a key component of projects underway at the UofA. Their interest in fast network subsystems requires some form of resource control.

Resource Control and Flow Control

The x-kernel consumes memory for a variety of reasons (messages, objects, thread stacks, driver queues); at this time we are most concerned with resources associated with messages because messages represent transient traffic and they introduce dependencies on the protocol graph's flow control facilities. In the context of these dependencies, there are two types of traffic that we are concerned with: privileged traffic and regular traffic. Currently, we refer to network traffic that is system-related as privileged; all other traffic is considered regular**.**

Our distinction between privileged traffic and regular traffic stems from the RI's experience with OSF/1 AD and its NORMA IPC subsystem. In the course of the AD work, the RI had to deal with an architectural deadlock between NORMA IPC and the Virtual Memory management system. For example, suppose node A is using node B for default paging activities. When node A runs low on memory resources, it begins to pageout data to node B. However, before node A can free a given page that has been paged out to node B, it must receive a message from node B indicating that the page has been successfully handled. If memory is so low that Node A is no longer accepting messages from the network, the node will be unable to free resources because it cannot receive the messages that would allow it to do so. We have solved this deadlock by modifying NORMA IPC to continue accepting paging-critical messages while memory is low, and we would like see a mechanism implemented within the x-kernel that would allow us to specify special paths for such messages through the protocol graph.

With respect to regular traffic, we may process messages via unreliable protocols, which do not guarantee message delivery, and reliable protocols, which typically use flow control facilities such as windowing, and can guarantee message delivery. Reliable protocols are usually configured above a number of unreliable protocols.

With respect to privileged traffic, the dropping of packets is likely to introduce significant penalties as described above: in some cases even unreliable protocols should not drop packets related to such traffic. Consequently, we would like the x-kernel to distinguish privileged traffic from regular traffic and guarantee delivery of privileged messages when memory resources are low.

Proposed Resource Control Mechanisms

A year ago, we proposed that the resource control facilities be centralized in the protocol-independent framework part of the x-kernel [10]. In that proposal, the protocol-independent code is responsible for detecting memory shortages (e.g. low water marks); upon detecting a shortage, the protocol-independent code passes the notification to the protocols, which then take protocol-specific operations to free memory. A given protocol might garbage collect, close sessions, or drop messages. Sessions associated with privileged traffic would continue to operate regardless of memory shortages.

Researchers at the UofA have investigated similar ideas, and have come up with an articulated resource control model. In this model, protocol modules allocate memory via *allocators* that are used to fetch memory from *pools*. There are a small number of independent pools (e.g., wired pool, pageable pool, a pool dedicated to paging traffic), and a large number of allocators, one per protocol object and one per session object. Each allocator allocates its memory from a single, assigned pool. Typically, memory is credited and debited from allocater to allocator as a given message is demultiplexed while travelling through a protocol graph.

Allocators may be used in two ways:

1. Protocols reserve memory by configuring their allocators to provide a minimal amount of bytes in a reliable manner. The Flow Control facilities within the protocol can rely upon this guarantee. This is referred to as the synchronous, or protocol-centric approach.

2. Protocols allocate memory as though there is an unlimited supply. When a given pool begins to reach a point where it cannot satisfy its allocators, the system calls back to the protocols to reclaim resources. When a protocol receives such a notification asynchronously, it performs the appropriate operations to release unneeded memory (garbage collection, connection pruning, flow control tuning, and so on). This is referred to as the asynchronous, or system-centric approach.

Which of these two approaches should the x-kernel implement, or should both approaches be implemented? This has been an extensively debated question. At this point in time, we are not confident that the first approach is necessary and sufficient to address our resource control problems because we at an early stage in the design of our protocols and their flow control facilities. Therefore, we propose that both approaches be made available. We will choose the appropriate approach when we have more experience and empirical data to reflect upon.

9 Paths: Distinguishing Regular Traffic from Privileged Traffic

As was mentioned earlier, we have made the distinction between regular and privileged network traffic. We are interested in providing a mechanism within the x-kernel that expedites the delivery of privileged messages regardless of whether or not memory is in short supply. We have proposed a *Paths* mechanism that would enable the x-kernel to support delivery of privileged messages.

The delivery of an x-kernel message involves the following set of actors:

- One or more messages.
- A shepard thread.
- One or more x-kernel objects (protocols, sessions).

We have associated each of these actors with a set of attributes, as follows:

messages	**allocator, pool, class of traffic**
shepard thread	**scheduling priority, scheduling policy, type of stack**
x-kernel objects	**allocator, pool, class of traffic, flow control, xThrottle**

It is also the case that x-kernel sessions objects regularly merge as we move downstream and fork as we move upstream. For example, an IPC session is identified by the <protocol type, remote IP, local IP> triple, with the effect that multiple TCP sessions associated with the same remote host (but different TCP ports) merge into a unique IP session. Although the action of merging is usually advantageous (fewer data structures are required), there are situations in which we cannot afford to lose the attributes of the sessions as the result of merging (or forking). For example, in a given situation, we may not want to lose a session's allocator.

These considerations led us to design a new type of x-kernel object, the path object. Path objects would be classified as a first class object and added to the existing protocol and session objects. The UofA has indicated that they like this idea as it would solve several problems related to the use of the x-kernel with ATMs.

With the path object:

1. We group together all session objects that must be invoked to pass a message from a top anchor session to the bottom anchor protocol. This group constitutes what we call the path object.

2. We collect and centralize into the path object all attributes that have hitherto been associated with the individual actors (the messages, the shepard thread, and the x-kernel objects).i

3. The x-kernel user can specify when it opens a path, the associated attributes, and level of associativity of the sessions that make up this path (by level of associativity, we mean the extent to which the path's sessions may merge or fork with one another).

In practice, the developer will call the xOpen routine (xOpenEnable) with an additional argument which points to a path structure whose attributes have been initialized with the class of services required (alternately, a built-in default path may be used). Among attributes, the developer may specify whether an "absolute" or a "relative" notion of path is required.

The absolute path implies no associativity among sessions: the xOpen call is propagated downstream and new session objects are always created; they cannot be merged or forked. (The session objects are logically distinct, but the implementation will allow them to share some data.) When a message destined for the path's endpoint is received, it is routed to the absolute path at the level of the bottom anchor protocol. The resources that will be required by the message are debited from the allocator defined by the path before any protocol processing begins.

The relative path allows sessions to merge; every relative path is associated with a priority number. When two sessions merge, the resulting session acquires its attributes from the path with the higher priority. Whenever an allocator is changed as a result of a merge/fork operation, the flow of messages through the sessions requires that memory be debited/credited from allocator to allocator.

The actual list of attributes that are associated with the path abstraction are as follows:

The type (Absolute or Relative) and priority (Relative only).

1. A filter command list specific to the path.
2. The definition of an xThrottle policy.
3. The definition of a garbage collection policy.
4. The definition of memory resources (Allocator, Pool, etc.)
5. A linked list of XObjs (the session objects that are part of the path).
6. Scheduling attributes.
7. A lock.
8. Binding data.

We have added the lock and binding information attributes to take advantage of features that may be available in the future with respect to SMP. These include the following:

- The path lock may be used to implement an intermediate locking mechanism as we evolve the x-kernel's SMP support from the global monitor/funnel mechanism to an explicit locking mechanism.
- The path lock may be used to exploit a convey effect to minimize lock contention (paths may be "released" in some order).
- The binding data may be used to bind the shepherd thread to processors (or processor sets, if they are available), creating dynamic partitions of the protocol graph, with the granularity of paths (this type of partitioning is much more efficient than static protocol-based partitioning).

10 Conclusion

In our evaluation of the x-kernel technology, we have found it to have many features that we believe will greatly benefit the network-related work currently on the OSF RI's agenda. We also believe that the technology will facilitate the development of projects that are likely to find their way onto that agenda in the future. As described in this paper, the OSF RI will need to put in a significant amount of work to address areas of the x-kernel that currently do not meet our requirements, but we are confident that the advantages inherent in the x-kernel technology far outweigh the costs required to merge it into our technology base.

11 Bibliography

[1] N. C. Hutchinson and L. L. Peterson. "The x-kernel: an Architecture for Implementing Network Protocols", IEEE Trans. on Software Eng., vol. 17, no. 1, pp 64-76, Jan 1991.

[2] S. O'Malley and L. L. Peterson. "A Dynamic Network Architecture", ACM Transactions on Computer Systems, 10(2), May, 1992.

[3] P. Druschel and L. L. Peterson. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", Proc 14th Symp. on Operating Systems Principals, 1993.

[4] P.Druschel, M. B. Abbott, M. A. Pagels and L. L. Peterson. "Network Subsystem Design", IEEE Network 7(4):8-17, July 1993

[5] H. Orman, E. Menze, S. O'Malley, and L. L. Peterson. "A Fast and General Implementation of Mach IPC in a Network", Mach Usenix, 1993.

[6] M. Bjorkman and P. Gunningberg. "Locking Effects in Multiprocessor Implementations of Protocols", Proc. 14th Symp. on Operating Systems Principals, 1993.

[7] A. Langerman. "NORMA IPC Version Two: Requirements", OSF RI Internal Report.

[8] C. Maeda and B. N. Bershad. "Protocol Service Decomposition for High-Performance Networking", Proc. 14th Symp. on Operating Systems Principals, 1993.

[9] Open Software Foundation. "The Design of the OSF/1 Operating System", 1990.

[10] F. Travostino. "X-kernel Integrated in Mach3 Kernel: Evolutions of the Protocol Independent Framework", x-kernel Workshop, Tucson, AZ, 1992

[11] F. Reynolds, F. Travostino, R. MacDonald, D. Elliston, K. Loepere. "Untyped IPC", OSF RI Technical Report #19930513, to appear, 1993.

[12] K. Loepere. "MACH3 - draft proposed specification", OSF RI, 1993

[13] M. Condict, D. Mitchell, F. Reynolds. "Server Co-location Project Proposal", OSF RI, June 21 1993.