An Extensible Protocol Architecture for Application-Specific Networking

Marc E. Fiuczynski
Brian N. Bershad
{mef,bershad}@cs.washington.edu

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Plexus is a networking architecture that allows applications to achieve high performance with customized protocols. *Application-specific* protocols are written in a typesafe language and installed dynamically into the operating system kernel. Because these protocols execute within the kernel, they can access the network interface and other operating system services with low overhead. Protocols implemented with *Plexus* outperform equivalent protocols implemented on conventional monolithic systems. *Plexus* runs in the context of the *SPIN* extensible operating system.

1 Introduction

This paper describes the design and implementation of *Plexus*, a protocol architecture that allows arbitrary applications to define application-specific protocols. *Plexus* allows protocol processing to be tailored using application-level knowledge, thus providing the framework for supporting new protocols [CSZ92], and implementing optimizations to existing protocols such as integrated layer processing and application level framing and buffering strategies[CT90].

A key aspect of *Plexus* is that application protocol code executes within a kernel-level protocol graph that can be dynamically changed as applications come and go. Once in the kernel, *protocol extension code* can access physical devices and operating system services, such as virtual memory and scheduling, with low overhead, enabling protocols to execute with high efficiency. For the same reason, protocol processing consumes fewer CPU cycles when compared to conventional monolithic implementations.

This research was sponsored by the Advanced Research Projects Agency and by an equipment grant from Digital Equipment Corporation. Fiuczynski was partially supported by a National Science Foundation GEE Fellowship. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship.

Although application-specific protocol code executes in the kernel, *Plexus* ensures that it does not compromise overall system safety. We rely on two strategies to ensure safety with exceptionally low overhead. First, application protocol code running in the kernel is written in a typesafe programming language which guarantees that the code respects the interface boundaries against which it was compiled. Second, using link-time control services, *Plexus* restricts direct access to lower level interfaces, ensuring that applications do not snoop or spoof network packets. The strategies are functionally identical to, although less costly than, those found in conventional operating systems where application-specific protocols execute in user-space [TNML93, MB93]; the first prevents arbitrary kernel memory from being accessed, while the second prevents arbitrary kernel functions from being called.

We have built *Plexus* in the context of the *SPIN* extensible operating system [BSP+95]. The system runs on DEC ALPHA workstations, and supports a range of protocols, including IP, ARP, ICMP, UDP, TCP and HTTP. Between a pair of ALPHAs running *Plexus*, we have measured an application-to-application round trip packet latency with UDP of less than 600 μsecs on an Ethernet, 350 μsecs on a Fore ATM network, and 300 μsecs on a DEC T3 network. In contrast, the same protocols implemented in *DIGITAL UNIX* using the same device drivers are substantially slower despite the fact that the DEC implementation has been highly optimized [CFF+93].

1.1 Motivation

A protocol that performs well for one class of applications can be a bottleneck for others. For example, applications that perform large bulk data transfers over wide area networks are best served by a protocol implementation that provides large local buffers. On the other hand, a connection-oriented protocol that is used for many small transactions is best served by an implementation that minimizes connection lifetime. An application might also benefit from a protocol that is specific to the application itself, rather than just an implementation of an existing protocol. For example, applications where data integrity is optional such as audio and some flavors of video might use an implementation of UDP for which the checksum has been disabled. This application-specific approach violates the strict definition of the protocol, but, when agreed upon by the communicating applications, is a legitimate way to improve performance.

The protocol architecture in conventional operating systems does not easily accommodate application-specific protocols. First, application-specific code often runs substantially slower than native kernel protocol code [BFM+94, RH91, Bir93, vRHB94]. Second, the failure of a protocol module can cause the entire operating system to fail [SMP92, Tho95]. In response to this, some systems only allow the "superuser" to define new protocols [HPAO89, Sun, Wel95], greatly limiting the set of applications for which a new protocol can be defined. In other systems, the protocols must be defined when the system is built [OP92]. Finally, the overall structure of protocols in most commercial systems does not encourage small localized changes.

Plexus offers application developers a protocol architecture having five properties key to the construction of application-specific solutions:

- *Performance.* The system allows an application-specific solution to perform better than the general solution provided by the operating system vendor. Protocol extensions run in the kernel's address space. This places the protocol close to the network device, eliminates the need to copy data to user space, simplifies process scheduling, and enables resource management decisions using application-level knowledge.
- *Safety.* The use of an application-specific protocol does not compromise the safety of other applications or the operating system. Extensions are isolated from the system and one another through the use of a typesafe language and inexpensive access control mechanisms.
- *Openness.* An application, regardless of its privilege level, may define application-specific protocols, with only marginal performance advantages available to users of higher privilege. This property comes from the inexpensive access control and indirection mechanisms used by *Plexus*.
- *Runtime adaptation.* Applications may add extensions to the kernel at any point during the system's execution without requiring superuser privileges or a system reboot. *Plexus* allows extensions to be safely loaded and unloaded into a running system, so that they can come and go with their corresponding applications.
- *Incremental adaptation.* The level of effort required to make a change to an existing protocol graph is roughly proportional to the size of the change. The *Plexus* protocol architecture has a modular structure with well-defined interfaces between components. It is defined as a protocol graph with fine-grained nodes implementing portions of protocol functionality.

The rest of this paper describes *Plexus* and is organized as follows. In Section 2 we discuss *SPIN*, the operating system in which we have implemented *Plexus*. In Section 3 we describe the design and implementation of *Plexus*. In Section 4 we present some microbenchmarks that demonstrate the system's performance. In Section 5 we describe the use and performance of the system in building several application-specific protocols. In Section 6 we survey related work. Finally, in Section 7 we conclude.

2 Overview of the SPIN operating system

SPIN is an operating system that can be dynamically and safely specialized to meet the performance and functionality requirements of applications [BSP+95]. Applications define system extensions in Modula-3 [Nel91], which is an Algol-like typesafe programming language. Extensions are dynamically linked into the kernel virtual address space, where they can access other operating system services with low latency. Low latency is important because it enables *fine-grained* interaction between the application and the operating system.

The *SPIN* kernel is written in Modula-3 with the exception of the device drivers which are written in C and borrowed from the *DIGITAL UNIX* source tree. The kernel itself implements threads, virtual memory, and device management. The virtual memory service is used to implement address spaces, which allows applications to execute in their own address space and be written in any language.

In addition to conventional kernel services, *SPIN* also provides *extension* services that allow application-specific services to be integrated into a running system. The extension services address two integration problems: how to safely install code into the kernel's address space, and how to safely attach that new code to existing kernel services.

The "install" problem is solved by *SPIN*'s dynamic linker [SFPB96], which accepts extensions implemented as partially resolved object files that have been signed by our Modula-3 compiler. The linker resolves any unresolved symbols in the extension against the *logical protection domain* against which the extension is being linked. A logical protection domain defines a set of visible interfaces which provide access to procedures and variables. For example, there is one logical protection domain that includes all interfaces within the kernel (few extensions have access to this domain). There is also a kernel domain that contains the interface for allocating packet buffers (most extensions have access to this domain). If an extension references a symbol that is not contained within the logical protection domain against which it is being linked, the link will fail and the extension will be rejected. Logical protection domains are first-class kernel resources; they are referenced by typesafe pointers (capabilities), and can be created, copied, and passed around. In this way, different extensions can be given access to different services.

The "attach" problem is addressed by *SPIN*'s dynamic event dispatcher, which communicates *events* to *event handlers*. An *event* is raised by a kernel service or extension code to announce a change in system state or to request a service. For example, the Ethernet device drivers raise the event `Ethernet.PacketRecv` to indicate the arrival of a new ethernet packet.

Events are defined and raised using the syntax of procedure declaration and call. That is, the event `Ethernet.PacketRecv` is declared as some procedure `PacketRecv` within an interface `Ethernet` as:

```
INTERFACE Ethernet;
IMPORT Mbuf;
PROCEDURE PacketRecv(READONLY m:  Mbuf.T);
...
END Ethernet;
```

The event is raised by "calling" the procedure as in `Ethernet.PacketRecv()`.

Applications interested in the occurrence of an event register an *event handler* with the *SPIN* dispatcher. A handler is a procedure that is executed in response to a specific event. Any extension that can name a particular event (that is, which is linked against the logical protection domain in which the event is defined) may raise the event. More than one handler may be installed on an event, and the overhead of invoking each handler is roughly one procedure call.

An event handler can be associated with a *guard*, which defines an arbitrary predicate that is evaluated by the dispatcher prior to the handler being invoked. If the predicate is true when the event is raised, then the handler is invoked, otherwise the handler is ignored. Guards permit extensions and the base system to separate the specification of what should happen from when it should happen. Extensions define the operations that occur in response to events, whereas guards ensure that those operations happen only at the proper time. *Plexus* relies on guards to implement *packet filters* [MRA87] that correctly route packets through the protocol graph to particular implementations of a protocol.

3 The *Plexus* Architecture

Plexus allows applications to define new protocols or to change the implementation of existing protocols. The system is structured as a protocol graph of event raisers and event handlers. Nodes in the graph represent protocols for which protocol events are raised to announce or request the passage of a packet through the node. The protocol processing functions are implemented by event handlers that are "attached" to the protocol event name.

Two fundamental events defined by all protocols are `PacketSend` and `PacketRecv`. These events roughly correspond to the `output` and `input` functions found in BSD style protocol stacks, and serve as the binding points for extending a protocol with customized implementations. Access to these events is controlled by a protocol-specific manager, which ensures that applications neither spoof nor snoop packets. Packets sent by an application are pushed down the graph through each protocol's `PacketSend` event until they reach the actual device. Packets received from the network are pushed up through the protocol graph according to each protocol's `PacketRecv` event.

Figure 1 shows the graph structure for a TCP and UDP stack. The structure of the graph is a decision tree, with the network device and application extensions forming end-points in the graph. Edges indicate the flow of packets through the protocol stack. Each level of the graph defines a guard and event handler procedure. The guard procedure acts as a packet filter, limiting packets whose headers are not matched by the guard's predicate on either input (to prevent snooping) or output (to prevent spoofing). The protocol processing functions are implemented by the event handlers that are attached to the protocol event name. Packets sent by the application are pushed down the graph until they reach the actual device. Packets received from the network are pushed up through the protocol graph by the events raised in the preceding protocol layer.

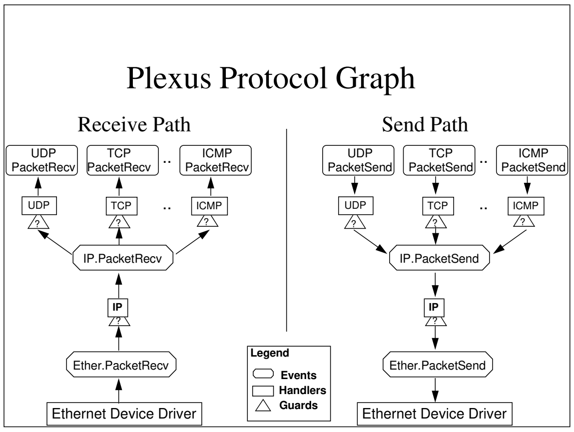

Figure 1: *This figure shows the basic structure of protocol stacks under Plexus. Each guard is responsible for demultiplexing through one layer of the stack. Each handler is responsible for pushing the packet up to the next layer. A guard (indicated by a "?") returns true if fields in the header satisfy its predicate.*

3.1 Controlling access to protocols

Protection reflects the canonical "mechanism vs. policy" issue for any system. As mentioned, *SPIN* and *Plexus* rely on a typesafe language and controlled linking as the protection mechanism. In this subsection, we discuss the protection policy.

There are two dimensions to protocol protection in *Plexus*: spoofing (sending a packet with an illegitimate source field) and snooping (illegitimately receiving a packet). Both are prevented through the use of *protocol managers* which ensure that a packet is never delivered to, nor accepted from, an illegitimate protocol graph node. It is the responsibility of the protocol manager to define the notion of "legitimacy" with respect to a protocol.

The manager provides a protocol specific interface for accessing and extending the protocol functions. It installs event handlers and guards on the behalf of untrusted applications. Consequently, application-specific extensions do not ever directly install their handlers on protocol events. However, once the handler has been installed, the dispatcher will route control directly to the handler (without going through the intermediate protocol manager). This model of protection is similar to the one used by the Mach user-level protocol library, where the UNIX server installs packet filters on behalf of the library's requests [MB93].

To send a packet, an untrusted, higher-level protocol must obtain the right to "raise" the `PacketSend` event that transfers the packet to a lower level protocol. Protocol managers prevent spoofing at this level by defining a `PacketSend` event specific to a legitimate sending endpoint. The handler associated with that event can either verify that the contents of the outgoing packet's source field match the endpoint, or more simply overwrite the source field. The former is useful for debugging protocols, while the latter provides the best performance.

Multiple protocol implementations

Plexus supports multiple implementations of the same protocol for different endpoints in the same way that it supports multiple protocols – different handlers are fired in response to different guard predicates evaluating true. For example, suppose that there are two TCP implementations, *TCP-standard* and *TCP-special*. Both handle the event `IP.PacketRecv`, but the first uses a guard which processes all TCP packets but those destined for the second, while the second handles only those packets destined for the particular set of ports for which it is responsible.

3.2 Safely and efficiently handling packets

Guards and handlers inspect a packet's payload as it traverses through the protocol graph. Since guards and handlers are written in Modula-3, and Modula-3 is a strongly typed language, the payload must have a type. More importantly, nodes in the graph must be able to cast packets from less specific types (for example, an array of bytes in a device buffer) to more specific types (for example, an Ethernet header followed by an IP header followed by a TCP header). Casting such as this is commonplace in networking software, but Modula-3 as defined lacks sufficient support to cast the primary form of an incoming packet – an array of bytes – safely into a Modula-3 type. The safe alternative, copying, imposes unacceptable overhead. Although unsafe casts are possible using Modula-3's `LOOPHOLE` operator, there is no way to ensure that the operator is used in only safe ways.

To allow for safe casting operations, we have defined a new operator for Modula-3 that converts an array of bytes to a restricted Modula-3 type [HFG+96]. The signature of the new operator is `VIEW(a,T):T`. The result of `VIEW(a,T)` is the expression a's bit pattern interpreted as a value of type T, which must be a scalar type or an aggregate of scalar types.

There are two major benefits that we gain as a result of the `VIEW` operator. First, it enables Modula-3 programs to safely view external data structures as instances of Modula-3 types without copying. Second, the array of bytes can be efficiently accessed with a more expressive Modula-3 type.

The code fragment in Figure 2 illustrates the use of `VIEW` in accessing packets from within a guard. It also shows how a guard is installed on an event using the Ethernet protocol manager.

3.3 Rapidly processing packets in response to interrupts

Protocols which require little processing for each incoming packet exhibit the best performance when they can run at interrupt level. For example, active messages [vECGS92] require a protocol that does little more than reference memory and reply with an acknowledgement. If such protocols run in a separate thread, outside of an interrupt context, then they will have unnecessarily large latency.

Only privileged device driver code – the bottom of the *Plexus* protocol graph – runs directly in response to network device interrupts. That code, though, may delegate some of its processing to extensions. The critical requirement of an extension so delegated is that it (a) return quickly, and (b) not block. The first requirement reduces the likelihood that an interrupt is lost during the handling of a previous one. The second simplifies the delivery of interrupts, enabling them to be processed in the context of "special" lightweight kernel threads.

We have introduced a small amount of compile-time support [HFG+96] to enable a delegating protocol manager, such as the Ethernet layer, to ascertain whether a potential event handler is a "good" candidate for running within an interrupt context. We

```
MODULE ActiveMessages;
IMPORT Mbuf, Ethernet;
PROCEDURE Guard(READONLY m:Mbuf.T):BOOLEAN =
 BEGIN
 (* View Ethernet header using Modula-3 type. *)
 (* WITH creates an alias to VIEW'ed header.  *)
  WITH etherHdr = VIEW(m.m_data,Ethernet.T) DO
   RETURN etherHdr.type = ActiveMessageProtoNum;
  END;
 END Guard;

PROCEDURE Handler(READONLY m:Mbuf.T)=
 BEGIN
  ...  (* Do active message handling. *)
 END Handler;

BEGIN
 (* Install on Ethernet event *)
 Ethernet.InstallHandler(Ethernet.PacketRecv,
                         Guard, Handler);
END ActiveMessages.
```

Figure 2: *The active message guard/handler pair is installed on the* `Ethernet.PacketRecv` *event. The guard is invoked for each incoming packet, and acts as a packet filter discriminating on the Ethernet type field. It uses the* `VIEW` *operator to safely cast a Modula-3 type on the payload array. The code relies on Modula-3's* `WITH` *statement, which creates an alias of the* `VIEW`*'ed array, to avoid unnecessary copying.*

define a good candidate as one that *can* be asynchronously terminated without damaging important state, but leave it to the candidate handler to distinguish itself as such. A procedure for which the implementation can tolerate premature termination without violating any data structure invariants is explicitly labeled as `EPHEMERAL`. An obvious restriction on ephemeral procedures is that they only call other ephemeral procedures. Our compiler enforces this restriction. Figure 3 illustrates some legal and illegal ephemeral handlers.

A protocol manager can verify that a potential event handler being installed on its `PacketRecv` event is in fact ephemeral by querying the type of the handler procedure. If the procedure is not ephemeral, the manager can reject the handler. Otherwise, the manager can direct the dispatcher to install the handler on the protocol's `PacketRecv` event, and optionally assign a time limit which, if exceeded during handling, will cause the handler to be prematurely terminated. For example, we have extended the protocol graph in Figure 1 to support active messages [vECGS92] over Ethernet. To minimize latency, the active message handlers execute in the network interrupt handler. On the receive path, the extension defines a guard that distinguishes active messages from other incoming Ethernet packets, and provides an ephemeral event handler to process the active message packet. When an active message packet arrives from the Ethernet, the guard will dispatch on the Ethernet type field, and the *SPIN* dispatcher will invoke the corresponding event handler. If the active

```
EPHEMERAL
PROCEDURE Enqueue(q:Queue, READONLY m:Mbuf.T)=
  BEGIN
    NonBlockingQueue.Enqueue(q, m);
  END Enqueue;

EPHEMERAL
PROCEDURE GoodHandler(READONLY m: Mbuf.T) =
  BEGIN
    (* Enqeue incoming packet *)
    Enqueue(ipQueue, m);
  END GoodHandler;

(* This procedure is not declared as  *)
(*    EPHEMERAL                       *)
PROCEDURE NotEphemeral(READONLY m: Mbuf.T) =
  BEGIN ... END NotEphemeral;

EPHEMERAL
PROCEDURE IllegalHandler(READONLY m: Mbuf.T) =
  BEGIN
    (* This procedure won't compile *)
    (* because NotEphemeral is not ephemeral. *)
    NotEphemeral(m);
  END IllegalHandler;
```

Figure 3: *Ephemeral and non-ephemeral handlers. A manager for a protocol that runs directly in response to a device interrupt would allow the installation of GoodHandler on its PacketRecv event, but should not allow the installation of NotEphemeral. The compiler will generate an error when compiling IllegalHandler since it calls a procedure that is not ephemeral.*

message handler exceeds its time allotment, it will be terminated.

3.4 Sharing packet buffers

A packet can be shared across multiple levels of the protocol graph on both the send and receive paths. Although multiple extensions can view the network data, they cannot modify it without making a copy of it first. We achieve this effect by passing packets through the protocol graph as *read-only* buffers.[1] Figure 4 provides an example of read-only buffers using Modula-3's `READONLY` type specifier. The `BadPacketRecv` procedure overwrites the contents of the packet. However, the code fragment will be rejected by the compiler as the left-hand side of the statement is a read-only variable. An extension such as `GoodPacketRecv` must allocate a new buffer and copy the read-only contents in order to modify it (i.e., an explicit copy-on-write).

[1] *Plexus* uses *mbufs* (the Berkeley memory buffer implementation) to pass packets through the protocol graph. A primary advantage of mbufs is that they are directly used by most UNIX device drivers.

```
TYPE Mbuf.T = RECORD
  m_hdr  : mh_hdrT;
  m_data : ARRAY [1..MLEN] OF Bytes;
END;

PROCEDURE GoodPacketRecv(READONLY m: Mbuf.T) =
  VAR p: Mbuf.T;
  BEGIN
    p := m;
    FOR i := FIRST(p.m_data) TO LAST(p.m_data) DO
      (* overwrite packet data *)
      (* will be allowed by compiler. *)
      p.m_data[i] := 0;
    END;
  END GoodPacketRecv;


PROCEDURE BadPacketRecv(READONLY m: Mbuf.T) =
  BEGIN
    FOR i := FIRST(m.m_data) TO LAST(m.m_data) DO
      (* overwrite packet data *)
      (* will be rejected by compiler *)
      m.m_data[i] := 0;
    END;
  END BadPacketRecv;
```

Figure 4: *A pair of packet receive handlers. One is legal because it does not modify its argument. The other is not legal and will be rejected by the compiler.*

3.5 Summary

Plexus defines a protocol graph in which applications can introduce new nodes (handlers) and edges (guards) at runtime. Safety is ensured by restricting an extension's access to underlying interfaces, and by filtering packets before they arrive at an extension's handler.

Although we have implemented *Plexus* in the context of *SPIN*, we believe that the general structure of our protocol system, in particular, the graph architecture, the interfaces, and the protocol protection model, could be implemented in more conventional systems such as UNIX provided they support kernel extensions in a safe fashion. Fundamentally, our protocol architecture requires that the kernel export two facilities: dynamic linking/unlinking and in-kernel firewalls. Dynamic linking/unlinking is necessary to install and remove new protocol extensions into and from the kernel. An in-kernel firewall makes it possible to run user code within the kernel without compromising system integrity. Typesafe languages are not the only mechanism for implementing firewalls. Alternative strategies include interpreted languages [GM, Ous94] and software based fault isolation techniques [WLAG93].

4 Performance

In this section we describe basic latency and throughput measurements for *Plexus* on a range of networking devices. Specifically, we examine the system's performance for UDP and TCP, and compare it to the same protocols running on *DIGITAL UNIX*, a commercial operating system.

All measurements were done using a version of the *SPIN* kernel from November 1995. We used the DEC SRC Modula-3 compiler (release 3.5.2) to compile the bulk of the kernel and its extensions. We used the DEC C compiler (from *DIGITAL UNIX* 3.2) to compile the system's device drivers, as well as version 3.2 of the *DIGITAL UNIX* operating system. Both *SPIN* and *DIGITAL UNIX* run on members of the DEC Alpha workstation family. For the measurements in this paper, we used DEC model 3000/400 workstations, which have an Alpha 21064 processor running at 133Mhz. Each workstation was equipped with 64MB of RAM, a 10Mb/sec Ethernet, a 155Mb/sec Fore TCA-100 ATM interface on the TurboChannel I/O bus, an experimental 45Mb/sec Digital T3 network adapter, and an SFB framebuffer. All Ethernet performance measurements were made between two machines on a private Ethernet segment. Our ATM cards are connected to a ForeRunner switch. The T3 measurements were made by connecting two workstations back-to-back. Our ATM network interface cards use programmed I/O, limiting maximum bandwidth to the rate with which the CPU can read the data from the network adapter. With our hardware configuration, we have been unable to achieve greater than 53Mb/sec when transferring data reliably between two device drivers. The T3 adapter uses DMA, and is able to deliver 45Mb/sec with minimal CPU involvement.

4.1 Protocol latency

The round-trip latency of a protocol reflects the overhead induced by the protocol as it transfers bytes from sender to receiver and back to the sender. Figure 5 shows the round-trip latency for small (8 byte) UDP/IP messages between a pair of application-specific functions on *SPIN*/*Plexus* and *DIGITAL UNIX* on Ethernet, the Fore ATM interface, and the DEC T3 interfaces. Both *SPIN* and *DIGITAL UNIX* use the same device drivers.

In *DIGITAL UNIX*, the application code executes at user-level, and each packet sent involves a trap and a copy-in as the data moves across the user/kernel boundary. In the worst case, the receive side must schedule the user process, copy the packet to user-space, and context-switch. In *SPIN*, the application code executes as an extension in the kernel, where it has low latency access to both the device and data. Each incoming packet causes a series of events to be generated for each layer in the protocol stacks (Ethernet/ATM, IP, UDP). The figure shows two bars for

Plexus for each device, one labeled `interrupt` and one labeled `thread`. The `interrupt` case reflects the round trip latency when the application-specific handler runs as an `EPHEMERAL` procedure that executes at interrupt level. The `thread` case reflects the overhead when the protocol and application handlers run in separate threads, with each event raise creating a new thread.

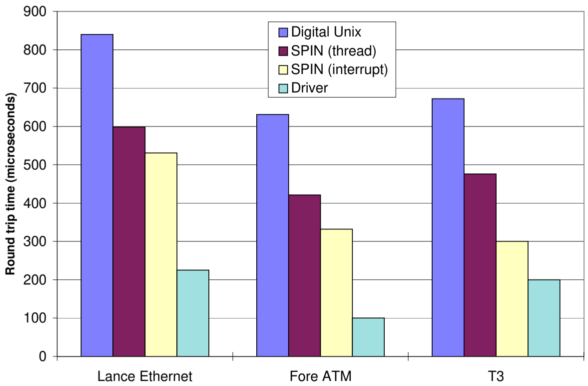

Figure 5: *UDP Round trip network send/receive time for small (8 byte) packets when using different networking hardware with* Plexus *and* DIGITAL UNIX.

The latency measurements show that cross-machine communication has lower latency when the target and source processes are able to send directly from the kernel. As mentioned, both systems use the same network device driver. Consequently, the difference between the performance of the two systems reflected in Figure 5 is due strictly to differences in operating system and protocol structure. The figure also shows the minimal round trip time using our hardware as measured between the device drivers. In tests using a faster device driver for *SPIN*, we measured a round-trip UDP latency of 337 μsecs on Ethernet and 241 μsecs on ATM. (We did not write a faster device driver for T3).

4.2 Throughput

Throughput is much less sensitive to operating system and application overheads than latency. Both *Plexus* and *DIGITAL UNIX* use the same TCP/IP implementation and device drivers. The *Plexus* TCP/IP implementation is one of the few cases in *SPIN* where we allow code not written in Modula-3 to be downloaded into the kernel. This code comes from a commercial vendor and we assume it to be safe, as it is conformant to interfaces and contains no illegal loads or stores. Consequently, the measured throughput for *Plexus* and *DIGITAL UNIX* are nearly identical. Specifically, for Ethernet, we saw 8.9 Mb/sec, and for the Fore ATM card, we saw 27.9 Mb/sec with *DIGITAL UNIX* and 33 Mb/sec with *Plexus*.[2]

5 Application-specific protocols

In this section we describe two application-specific protocols that we have built using the *Plexus* architecture.

- *A network video protocol* that shows how an application-specific protocol can deliver greater throughput with lower CPU utilization than a general protocol.
- *A packet forwarding protocol* that quickly reroutes TCP and UDP packets on a port-specific basis.

In the first case, we implemented multicast semantics for the UDP protocol. In the second case, we modified the TCP and UDP receive paths to simply reflect packets to another host. In both cases, we compare the performance of these application-specific protocols against standard protocols.

5.1 Network video

We have used our networking architecture to implement a networked video system consisting of a server that multicasts video clips to a set of clients. The server consists of one extension that reads video frame-by-frame off of the disk using *SPIN*'s file system interface. Because the video server extension is co-located with the kernel, it does not have to copy the data across the user/kernel boundary to send the disk block data back out over the network. The advantage of this structure is that video frames leave the server quickly as they travel from disk to network.

The server sends each frame as a UDP packet over the network to a number of clients. A video stream is composed of 30 frames per second. Each client receives one stream from the server. We experimented with 1–30 streams to determine when the server would fail to meet its deadline of sending each client a video stream.

Figure 6 shows the processor utilization on the server as a function of the number of client streams for our video system running over the T3 network. At 15 streams, both *SPIN* and *DIGITAL UNIX* saturate the network, but *SPIN* consumes only half as much of the processor. Compared to *DIGITAL UNIX*, *SPIN* can support more clients on a faster network, or as many clients on a slower processor. We do not present data for the Fore interface, because for both systems, the majority of the server's CPU resources are consumed by the programmed I/O that copies data to the network one word at a time (recall that T3 uses DMA).

[2] A bug in *SPIN*'s DMA support prevented us from measuring TCP throughput in this one case.

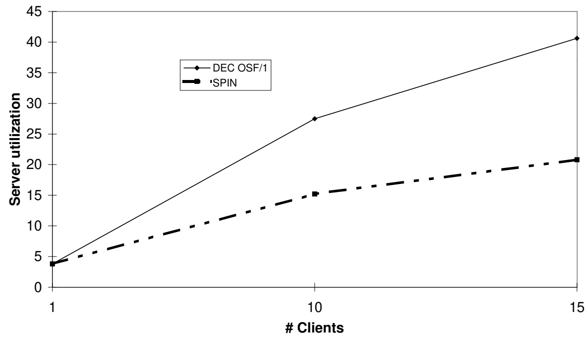

Figure 6: *Utilization of the server's CPU as a function of the number of client video streams.*

The client

On the client, an extension awaits incoming video packets with the protocol graph shown in Figure 1. The client extension checksums and decompresses the image and displays it directly to the screen's framebuffer. The current implementation makes two passes over the data, one pass for the checksum and another to decompress the image.

The client viewer is a good candidate for the integrated layer processing optimizations suggested by Clark [CT90]. We use the same video display code for both the *SPIN* application extension and the *DIGITAL UNIX* application. To achieve this, the video client implemented on *DIGITAL UNIX* maps the framebuffer directly into its address space and uses the same viewer code as the *SPIN* implementation. We expected that the overhead incurred for the data and control transfer to be significantly higher for *DIGITAL UNIX* compared to *SPIN*. However, the CPU utilization between the two operating systems was similar.

The similarity in performance is due to the high overhead in writing video data to the framebuffer. Thus, the performance of the video client is limited by the write bandwidth of the framebuffer hardware rather than overhead incurred by the operating system. Writing to the framebuffer memory is a factor of 10 times slower than writing to standard RAM. Displaying the video data is a factor of up to 50 times slower than all of the low-level OS operations combined. Clearly, the benefits of application specific networking protocols are most notable when the protocol processing is not dominated by application processing. In particular, customized protocols are most appropriate for applications that require low latency or high throughput with lower CPU utilization. In the video client, the benefits of a customized video protocol are offset by the fact that the application spends a significant amount of time (more than 90%) writing data to the framebuffer, rather than processing video packets from the network. We expect that with better video hardware, such as the DEC J300 device [PP93], the dominant performance bottleneck will be the protocol processing rather than the application processing.

Protocol forwarding

Plexus can be used to provide protocol functionality not generally available in conventional systems. For example, we have built a forwarding protocol that can be used to load balance service requests across multiple servers. To do this, an application installs a node into the *Plexus* protocol graph that redirects all data and control packets destined for a particular port number to a secondary host. We have implemented a similar service using *DIGITAL UNIX* with a user-level process that splices together an incoming and outgoing socket. The *DIGITAL UNIX* forwarder is not able to forward protocol control packets because it executes above the transport layer. As a result it cannot maintain a protocol's end-to-end semantics. In the case of TCP, end-to-end connection establishment and termination semantics are violated. A user-level intermediary also interferes with the protocol's algorithms for window size negotiation, slow start, failure detection, and congestion control, possibly degrading the overall performance of connections between the hosts. Moreover, on the user-level forwarder, each packet makes two trips through the protocol stack where it is twice copied across the user/kernel boundary. Figure 7 shows the impact that this additional work has on TCP forwarding performance.

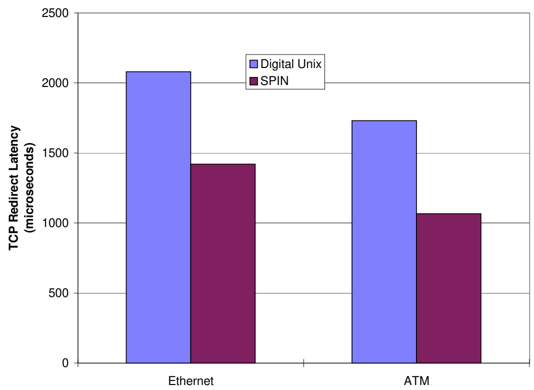

Figure 7: *TCP redirection latency using* Plexus *and* DIGITAL UNIX. *The* DIGITAL UNIX *implementation runs at user-level and is unable to respect end-to-end TCP semantics.*

6 Related Work

Many operating systems provide an interface that enable code to be installed into the kernel at runtime. Examples include dynamically linked device drivers, system calls, and networking protocols [Sun, IBM93, Wel95]. However, the extensions in these systems can see all kernel symbol names, giving them free-reign over the internals of the system. For example, a dynamically linked extension has complete access to the virtual memory subsystem, thus can manipulate data belonging to all currently running applications. In these systems, application-specific extensibility is not safely possible, as any system extension can bring down the entire system or violate system integrity. Fall and Pasquale describe a *splicing* mechanism that enables applications to route data between two devices directly within the kernel [FP93a, FP93b]. Splicing is only a partial solution, however, if applications must manipulate the data, for example, to decrypt, before passing it on to the next stage in the splice.

Several projects have defined protocol structures allowing applications to use their own protocols in a safe manner within their address space [TNML93, MB93]. Our architecture resembles their organization by separating protocol send and receive code from end-point management, but allows applications to link code into the kernel virtual address space where they have low latency access to operating system services, buffer management, and the network interface.

The *x*-kernel [OP92] provides a framework for implementing network protocols. In the *x*-kernel, many small protocols, called micro- and virtual- protocols, implement simple processing functions tied together by a complex graph to implement a standard protocol. Similarly, our architecture decomposes the stack into a protocol graph using *SPIN* events and guards. Our protocol event handlers resemble the *x*-kernel's *micro-protocols*. Similarly, our guards are functionally equivalent to *virtual protocols*. The main difference between the two architectures is that we permit users to safely and dynamically extend the protocol graph in the kernel by allowing application-specific protocol code to be placed within the stack.

The Fox project [BHL93] uses a high level language, ML, to implement protocol stacks. Through the use of strongly typed interfaces they can compose arbitrary protocol and experiment with customized protocols. However, their system runs only in user space, making it difficult to achieve good performance with a customized protocol. Moreover, current ML compilers focus on optimizing higher-order functions and polymorphism instead of loops and data representations, with the result being that typical systems code runs 3 to 5 times slower than comparable C code. In contrast, Modula-3 compilers can generate code with quality comparable to widely available C compilers [SSPB96].

Horus [vRHB94] identifies the need for fast and flexible protocols in the context of group communication. Horus provides a layered architecture for applications to compose customized protocols from a set of predefined basic protocols. All protocols inherit from a basic group which applications extend with group communication features, such as message ordering or flow control. Their work concentrates on providing application specific communication protocols in the context of group communication, whereas our work allows applications to dynamically extend the system's protocol stack.

7 Conclusion

This paper described an application specific networking architecture that allows an application's extension to be co-located with the kernel, thus enabling it to directly access system resources with low overhead. The extension model allows untrusted application extensions to handle system events efficiently inside the kernel. A demonstration of the protocol stack as it services HTTP requests can be found at *http://www-spin.cs.washington.edu*. The source for the protocol architecture can be found on the *SPIN* home page at that site.

Acknowledgements

We would like to thank the members of the *SPIN* group at the University of Washington for their help in building the operating system that hosts *Plexus*. David Becker did the initial port of TCP/IP to *SPIN* that became the basis for the *Plexus* implementation. Emin Gün Sirer and Przemysław Pardyak assisted with the measurements presented in this paper. Stefan Savage provided feedback on earlier versions of this paper. Our friends from Digital Equipment Corporation also deserve special thanks. David Boggs supplied us with our T3 interfaces. Finally, Bill Kalsow and others at DEC SRC have made available a high quality implementation of Modula-3, enabling us to concentrate on the high level interface issues that arise in the design of an extensible protocol system.

References

[BFM+94] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, , and H. Zhang. The Tenet Real-Time Protocol Suite: Design, Implementation, and Experiences. Technical Report 94-059, International Computer Science Institute, Berkeley, November 1994.

[BHL93] Edoardo Biagioni, Robert Harper, and Peter Lee. Standard ML Signatures for a Protocol Stack. Technical Report CMU-CS-FOX-93-01, Carnegie Mellon University, 1993.

[Bir93] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12), December 1993.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[CFF+93] Chran-Ham Chang, Richard Flower, John Forecast, Heather Gray, William R. Haw, K. K. Ramakrishnan, Ashok P. Nadkarni, Uttam N. Shikarpur, and Kathleen M. Wilde. High-performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP. *Digitial Technical Journal*, 5(1), 1993.

[CSZ92] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *SIGCOMM '92 Conference Proceedings*, pages 14–26, August 1992.

[CT90] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the ACM SIGCOMM '90 Symposium on Communications Architectures and Protocols*, pages 200–208, September 1990.

[FP93a] Kevin Fall and Joseph Pasquale. Exploiting In-kernel Data Paths to Improve I/O Throughput and CPU Availability. In *Proceedings of the 1993 Winter USENIX Conference*, pages 327–333, January 1993.

[FP93b] Kevin Fall and Joseph Pasquale. Improving Continuous-media Playback Performance With In-kernel Data Paths. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, pages 100–109, June 1993.

[GM] James Gosling and Henry McGilton. The Java Language Environment: A White Paper. http://java.sun.com.

[HFG+96] W. C. Hsieh, M. E. Fiuczynski, C. Garrett, S. Savage, D. Becker, and B. N. Bershad. Language Support for Extensible Systems. In *First Workshop on Compiler Support for Systems Software*, February 1996.

[HPAO89] Norman C. Hutchinson, Larry Peterson, Mark B. Abbott, and Sean O'Malley. RPC in *x*-kernel: Evaluating New Design Techniques. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 91–101, December 1989.

[IBM93] IBM Corporation. *AIX Version 3 for RISC System/6000 - Kernel Extensions and Device Support Programming Concepts*, October 1993. SC23-2207-00.

[MB93] Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.

[MRA87] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, 1987.

[Nel91] Greg Nelson, editor. *System Programming in Modula-3.* Prentice Hall, 1991.

[OP92] S. W. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.

[Ous94] John K. Ousterhout. *Tcl and the TK Toolkit.* Addison-Wesley Publishing Company, 1994.

[PP93] Lawrence G. Palmer and Rick S. Palmer. DECspin: A Networked Desktop Videoconferencing Application. *Digitial Technical Journal*, 5(2), 1993.

[RH91] Franklin Reynolds and Jeffrey Heller. Kernel Support for Network Protocol Servers. In *Proceedings of the Second Usenix Mach Workshop*, November 1991.

[SFPB96] E. G. Sirer, M. E. Fiuczynski, P. Pardyak, and B. N. Bershad. Safe Dynamic Linking in an Extensible Operating System. In *First Workshop on Compiler Support for Systems Software*, February 1996.

[SMP92] Andrew Schulman, David Maxey, and Matt Pietrek. *Undocumented Windows.* Addison-Wesley, 1992.

[SSPB96] E. G. Sirer, S. Savage, P. Pardyak, and B. N. Bershad. Writing an Operating System in Modula-3. In *First Workshop on Compiler Support for Systems Software*, February 1996.

[Sun] Sun Microsystems. *Solaris - SunOS 5.3 Writing Device Drivers.*

[Tho95] Tom Thompson. Copland: The Abstract Mac OS, July 1995.

[TNML93] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing Network Protocols at User Level. In *Proceedings of the ACM SIGCOMM '93 Symposium on Communications Architectures and Protocols*, September 1993.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[vRHB94] R. van Renesse, Takako M. Hickey, and Kenneth P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report 94-1442, Cornell University's Computer Science Department, August 1994.

[Wel95] Matt Welsh. Implementing Loadable Kernel Modules For Linux. *Dr. Dobbs Journal*, 20(5), 1995.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 203–216, December 1993.