SwitchWare: Towards a 21st Century Network Infrastructure

J. M. Smith†, D. J. Farber†, C. A. Gunter†, S. M. Nettles†,
Mark E. Segal‡, W. D. Sincoskie‡, D. C. Feldmeier and D. Scott Alexander†.*

ABSTRACT

SwitchWare is a set of software technologies which will enable rapid development and deployment of new network services. By making the basic network service selectable on a per user (or even per packet) basis, the need for formal standardization is eliminated. Additionally, by making the basic network service programmable, the deployment times, today constrained by capital funding limitations, are tremendously reduced (to the order of software distribution times). Finally, by constructing an advanced, robust programming environment, even the service development time can be reduced.

A *SwitchWare* switch consists of input and output ports controlled by a software-programmable element; programs are contained in sequences of messages sent to the *SwitchWare* switch's input ports, which interpret the messages as programs called ‘‘*Switchlets*’’. This accelerates the pace of network evolution, as evolving user needs can be immediately reflected in the network infrastructure. Immediate reconfigurability also enhances the adaptability of the network infrastructure to unexpected situations.

A network built from *SwitchWare* switches is an *active network*.

1. Introduction

The pace of network evolution (not switch evolution, *network* evolution) proceeds far too slowly. To a large degree this is a function of standardization. Standardization is a necessary step in network design to ensure interoperability, as a network's utility increases with the number of interconnected nodes. Since today's Internet architecture mandates the implementation of IP in all routers and hosts, and requires a 5-8 year standards→ development → deployment process (*e.g.*, IETF → Cisco → Internet Service Providers), it is inflexible and evolves slowly.

The Internet Protocol (IP) forces interoperability by defining a standard packet format and addressing scheme which is overlaid on networks comprising the internetwork. Since it must operate on the least capable of networks, it is designed to offer a minimal set of functions; additional services are added by overlays on IP. Three undesirable consequences of this design are:

1. It must run everywhere (*e.g.*, at hosts and switches). There are two subconsequences: changing IP means changing everything, and everyone must share the same service model.
2. Use of overlays (*e.g.*, the reliable stream overlay of TCP, or multimedia multicast with MBONE [12]) is forced for people who don't accept the communal service model, *i.e.*, they

---

† CIS Department, University of Pennsylvania. This research was supported by DARPA under contracts DABT 63-95-C-0073 and N66001-96-C-852. Additional support was provided by Intel, Hewlett-Packard and AT&T.

‡ Bellcore

* MUSIC Semiconductors, Inc. (formerly with Bellcore)

want or need a different service.

3. IP has no semantics for passing data-link layer information to the end-points.

Overlays are problems for two additional reasons. First, overlays may be inefficient because the underlying network does not take the functionality of the overlay into account, *e.g.*, packet loss versus cell-loss [40] for IP/ATM. Second, partitioning of resources is harder because we must split the partitioning of resources within an overlay from the partitioning of resources among overlays.

A second alternative, stemming from our overall goal of accelerating network evolution, is to create a virtual network infrastructure, consisting of *SwitchWare* switches. This alternative, if realized, has profound consequences for the engineering of future networks. These are:

1. *Programmable services*, to accelerate network evolution.
2. *Extensibility*, so that *logical* overlays can be implemented within the switches rather than as true overlays at the endpoints. Programmability alone is not extensibility; for example, extensibility is missing in control software for telephone switches [30]. It seems most useful to provide user-extensibility, so that new applications not imagined by the designers can be easily added, and we can avoid the risks of a ‘‘narrow-gauge’’ infrastructure.
3. *Security*, as this is both an increasing concern as networks become more widely applied, and increasingly difficult as they become more complex. For us, *robustness* is an aspect of security.
4. *Partitioning*, to control resource allocation and scheduling under a programmable *policy*.
5. *Portability*, so that software switching performance can keep pace with component technology curves, such as processor performance, and carry software switched applications along the same upslope.

We propose a *SwitchWare* switch, which provides a programmable element, essentially a computer, to perform switching functions and address this list of goals. Extending the role of computing in the network is the key to accelerating the evolution of network infrastructure; a compelling example is the rate of evolution of the World-Wide Web with its simple HTML language and Common Gateway Interface scripts.

The approach suggested in this paper is an extension of that used to revolutionize telephony in the early 1990’s. Advanced Intelligent Networking [39], developed in part by Bellcore, separates the implementation of telephony services from basic switching by moving the service control to an adjunct processor from the switch. Since each call can now have a different service, the need for standardization of new services has been eliminated. Deployment times are greatly reduced, since a new service is essentially data entered into the database of the adjunct processor. Development times are even reduced by enabling service providers and users to define and develop new services, and by a graphical programming interface developed by Bellcore. The telephony industry has seen new production quality service creation times drop from over two years to as little as two weeks as a result of AIN. The *SwitchWare* switch will extend the approach used by AIN to greatly increase the level of programmability in the switch, by reducing the need for a call model which constrains AIN. We will also apply the technique to internetwork routers and ATM switches, which have not been attempted by AIN.

2. Switching and the Pace of Network Evolution

The pace of network evolution proceeds far too slowly, relative to the technological changes in the underlying transmission systems, where laboratory results have reached Terabit/second bandwidths, and relative to the applications deployed at the edges of the network, such as the World-Wide Web and its supporting technologies such as the Java [26] Programming language. The element interconnecting the links and end-nodes is a switch; logically (although atypically) it is possible to view routers, bridges, etc. as specialized switches.

Programmability of switching elements led to major progress in the evolution of our national network infrastructure. An excellent case study of telecommunications switching infrastructure [28] is the Western Electric 3B20D processor [47] and the associated Duplex Multiple Environment Real Time (DMERT) [19, 17] operating system. This system was employed in the Bell System's 5ESS switch systems which remain in widespread use. DMERT is based on the earlier MERT operating system [24], and provides both a real-time and timesharing environment. The 3B20D offered user-programmable microcode so that high-performance applications could in fact create a custom or emulated machine architecture within the context of the 3B20D processing unit; this was used to support code and devices from earlier switch fabrics such as the 1A attached processing unit. Up to four concurrent instruction sets were supported; an instruction set could be selected with a single native microcode instruction.

This system reflected the importance of software in implementing the national telecommunications architecture, as it was designed from the start to be an effective execution platform for software. The programming model allowed programs to be loaded at run-time, but of course was not accessible to arbitrary users of the phone system.

What has changed in our modern environment is the need for a variety of programmed, customized *services*, and the model of updating central office switches using a van full of magnetic tapes is no longer appropriate.

2.1. A software approach: the Advanced Intelligent Network (AIN)

As we remarked earlier, the approach suggested in this paper is an extension of that used to revolutionize telephony in the early 1990's, Advanced Intelligent Networking [39], which was developed in part by Bellcore. The use of an independent control processor in the switching fabric gave service designers access to databases and other processors to provide call processing features. The response to a telephone call can then be represented as a state machine, which takes actions as information is input during a call. Examples of services that can be provided with this model would include routing of a call to the nearest shop in a chain of Pizza delivery services. The call processing would reference a Geographic Information System, and could be enhanced with vendor provided data such as availability of drivers.

The deep, and fundamental restriction on the applicability of this approach is its use of the call model, which is far too restrictive for the network infrastructure we have now, which is evolving from circuits to packets, and if the *SwitchWare* approach is taken, beyond to typed data objects.

2.2. Why not the Internet model?

As we argued in the Introduction, this slow evolutionary pace is a function of standardization. The Internet Protocol (IP) forces interoperability by defining a standard packet format and addressing scheme which is overlaid on networks comprising the internetwork. Since it must operate on the least capable of networks, it is designed to offer a minimal set of functions; additional services are added by overlays on IP.

The difficulty with this model is that it is extremely difficult to *interpose* new protocol functionality. This can be illustrated with the example of Domain Name Service (DNS). The pressures on DNS are tremendous and likely to increase. Many applications are dependent on it, and the World Wide Web's use of location-dependent naming places further pressure on DNS performance. The future will bring personal networks of perhaps hundreds of processors and intelligent sensors - such a network's elements will need names for management and function location. DNS will not scale to such an extent without caching, and yet the appropriate caching functionality cannot be built without interposed protocols for DNS cache management (including security features to prevent spoofing) and WWW proxies. These features require software embedded in the information network.

An excellent example of interposed functionality can be drawn from electronic mail systems, which can interpose tools like the ‘‘`vacation`’’ program to alter mail handling when people are on travel. Such systems have been extended with programming to provide priorities based on addressees and message sizes, which are transparent to the sender.

2.3. *SwitchWare* Programming

For any workable communication, there must be *some* agreement; standardization is essentially an agreement about what the agreement is. The IP protocol has been successful in standardizing packet formats, but because its standardization process operates at a political tempo rather than a technological tempo, the pace of evolution has been held back. We believe that a PostScript-like [43] concept, which raises the level of abstraction of the standard, to *SwitchWare* services rather than IP services, is the method for staying on the technology curve. Raising the level of abstraction also gives a much greater toehold for network management, specifically for automated self-diagnosis and repair. This is true because *(1)* behavioral assertions are simpler to state, *(2)* monitoring software is easier to write, and *(3)* the chain of assertions that lead to diagnosis and repair is less complex.

For most rapid evolution, networks must be user-customizable, and for users to drive deployment of new services, the network must be on-the-fly programmable. That is, it must be programmable by the packets that flow through it. While not all packets need contain code, packet sequences can contain modules of programming, as in the mobile agents prototyped by Knabe [22]. These code objects are used to provide customized services to the level of an individual user, or if predictions of hundreds of processors or intelligent sensors per person are true, perhaps composites of hundreds of such services.

3. *SwitchWare* Applications

We are implementing prototype services in a *SwitchWare* system to show feasibility. These services have the properties of being useful to a subset, but not necessarily all users of the active network. Services which are useful to all or most users of the network, like simple unreliable datagram forwarding, or unreliable multicast are susceptible to being included in a traditional bearer service such as IP. Services which are highly speculative, too forward looking, or simply not well understood are good candidates for being implemented in an active network. Several example services which match these characteristics are described below.

3.1. Self-paying information transport

The idea of Self-paying information transport (we'll resist using the acronym) is to have an object which is to be transported through the network include some form of electronic payment information as part of the object. A simple analogy would be to the stamp on a letter today. A transportable object (such as a packet or a virtual circuit) would contain, as part of the control information (*i.e.*, the packet header or VC setup messages) some sort of electronic payment information. This could be either e-cash, e-check, or an electronic credit card number. The payment information would then be examined by the *SwitchWare*, and if sufficient payment was offered, the object would be serviced by the *SwitchWare*. Note the service might be to provide computing by executing the object in the *SwitchWare*, or to provide communications by switching the object, to provide storage for state information the object may wish to leave in the *SwitchWare* switch, or some combination of these. The payment information may then be altered (some e-cash subtracted) as the object traverses the network.

This type of service is speculative enough that it would not be possible to consider standardizing it in a bearer service today. However, it is not hard to envision either commercial or military scenarios where it might be useful. In commercial situations, it provides the possibility of creating a dynamic market in network bandwidth, which may be more economically efficient than todays fairly static tariff structure where prices only change at fixed times of the day. A provider with an underutilized network might lower his prices, thus attracting objects into his network. A provider who was overloaded could raise prices until the demand subsided to match available capacity.

Since payment is really a complex form of priority, it's possible that in a military application, the payment may instead be interpreted as an authorization and priority. Requests that carried insufficient priority in times of high demand would be either offered a lower grade of service, delayed, or possibly even dropped. Far more dynamic schemes might be constructed as required. This scheme could be used, for example, to control QoS-based scheduling inside the *SwitchWare* run-time system.

3.2. Network management

Many network management tasks consist of collecting and collating data, such as event counts. To provide the most useful network management data, such as exception indications, intelligence must be used to filter out uninteresting (unexceptional) events. An easy way to write a network management system, assuming that appropriate authentication and protection can be devised, is to write a network management program using modules constructed from sequences of ‘‘program’’ packets.

Fault management is a very important and difficult task, particularly so for large networks and for correlated failures. Correlated failures may be caused by both environmental factors, such as earthquakes or explosions, or by malicious intruders. We believe that active networking can be used to significantly improve fault detection and management capabilities in the network.

Existing network monitoring for fault detection consists of gathering a known set of measurements. The fault management system filters and correlates these measurements. A problem with this approach is that it's difficult to integrate network elements that operate with different fault management systems. Network elements are designed to operate with one specific fault management system. Also, differing design philosophies may prevent the integration of several fault management systems. These incompatibility issues also make it difficult to evolve the fault management system, because it is difficult to add a network element that does not conform with all existing elements.

Active networks can provide the desired flexibility, because the fault management system can be changed as necessary without the need to worry about backward compatibility. Existing systems can be reconfigured as necessary simply by changing the code used for fault management. Active networking also may allow for hierarchical fault management. As faults are being isolated and identified, the fault management system can be refocused to examine in more detail those network elements that may be operating incorrectly. Different versions of fault detection code can be loaded into selected network elements for each level of the hierarchical fault management process.

3.3. Active Network Striping for Software Scalable Bandwidth

One of the major challenges to the vision of Active Network technology and virtual infrastructures is providing compelling examples of the usefulness of the on-they-fly programmable infrastructure. *SwitchWare* provides the opportunity for *software scalable bandwidth* to be derived from the virtual infrastructure. Variations on the same technique can address delay jitter (by resynching typed packets with *SwitchWare*) and reliability.

Two interconnected *SwitchWare* switches and attached host computers are shown in **Figure 1**.

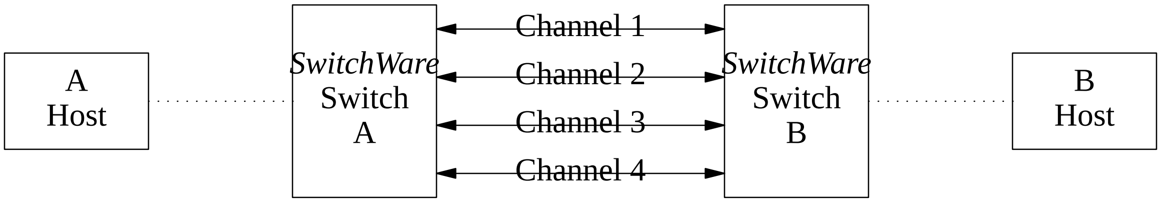

Figure 1: Interconnected *SwitchWare* Switches

While adding striping *hardware* to all switches in a network is unlikely to be cost-effective [48, 49], the *SwitchWare* infrastructure can be programmed to provide striped services. A software-implemented solution would stripe most effectively by using multiple interfaces to send multiple concurrent packets. Thus, simple pseudocode of a *Switchlet* for sender striping (asynchronous `Send()`), would be:

```
When Arrives(Packet,InPort)
{
      Send((SequenceNumber,Packet),OutPort);
      OutPort := (OutPort + 1) Mod Channels;
}
```

and the receiver would execute:

```
When (Arrives((SequenceNumber,Packet),InPort))
{
      If (InOrder(SequenceNumber,Expected))
      {
            Send(Packet,OutPort);
            Expected := Expected +1;

            While(CheckQueue(QueueName,Expected))
            {
                  DeQueue((Expected,Packet));
                  Send(Packet,OutPort);
                  Expected := Expected + 1;
            }
      }
      else
            Queue((SequenceNumber,Packet),QueueName);
}
```

The key observation to make about packet striping is that it offers the possibility of multiplying the throughput available between processors in proportion to the number of stripes.

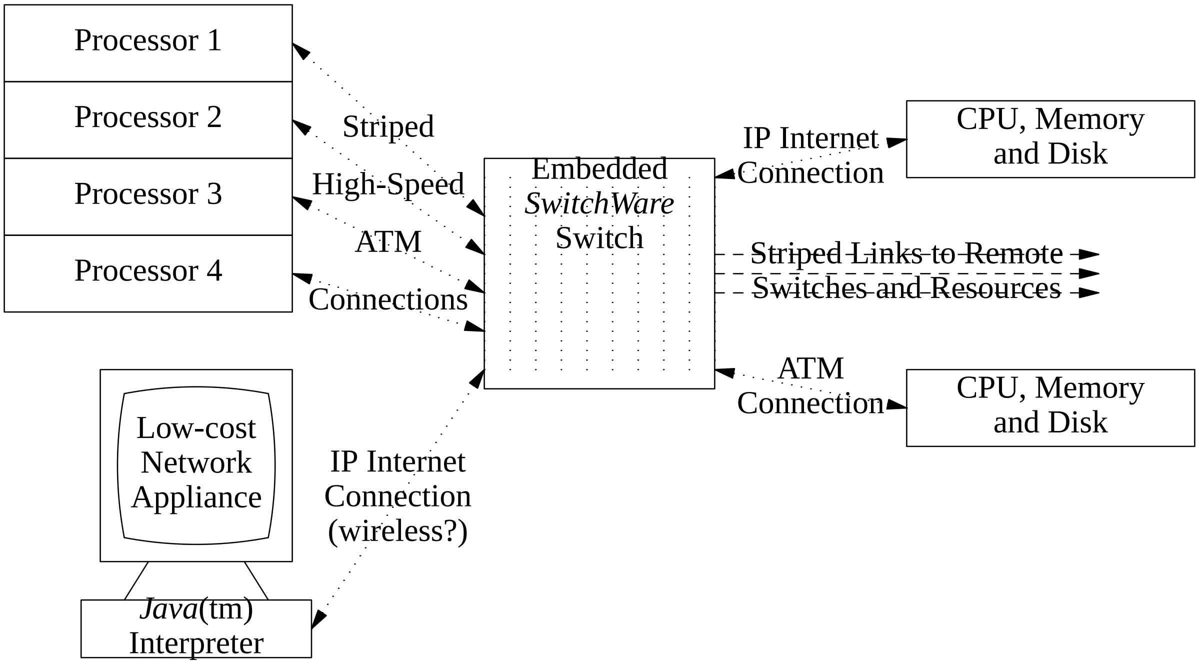

Figure 2: An embedded *SwitchWare* switch

This multiplication can be accomplished with no change in the hardware; rather, assuming that the interfaces are attached to processors able to support their memory bandwidth demands, the focus is on algorithms for deciding which interface(s) to use, and when to stripe, versus simply using a single connection. **Figure 2** illustrates striping in an embedded switch.

3.4. Self-Checking Active Networks (SCANs)

DARPA-sponsored network security research in the 1970s led to the Overseer [36] idea, where network packets were checked for correctness with respect to a protocol graph. Active Networks make this feasible. Active Network technology can be used to construct survivable network infrastructures based on the idea of ‘‘self-checking’’ systems. The basic approach is to use high-level specifications to generate Active Network programs that can be used to self-check the operation of the network.

As suggested by early work of Cerf [8], Farber, and Postel [37], protocols are represented as graphs that show the protocol’s flow of control and message emissions. Distributed systems use such protocols to operate over the network, giving rise to a ‘‘protocol graph,’’ a common and powerful formal representation of protocols. Traversing paths through the protocol graph results in valid sequences of messages. As an example, consider the modules `A`, `B`, and `C` constituting a protocol, shown below in **Figure 3**,

```
A:
 if( conditionA ) then B else C;
B:
 if( conditionB ) then C else STOP;
C: B
```

Figure 3: Simple Protocol with Modules A, B and C

and a derived graph in **Figure 4**:

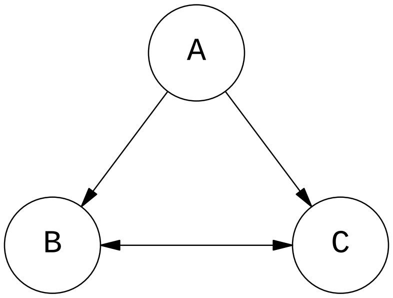

Figure 4: Graphical Representation of Control flow for Modules A, B and C

If each arc traversal generates a message, then we can test sequences of messages to see if they represent valid state sequences. For example, if we see {A→C, C→B}, we know the system is following a valid path through the graph. On the other hand, if we see {A→B, C→B}, then some failure has occurred, whether it be the loss of a packet resulting in a missing transition, or an invalid transition.

An important property of the path traversal<->message sequence model is that the number of paths in a real protocol is vastly fewer than what is possible for an arbitrary set of modules. Whereas complete static analysis of the protocol will require examining all possible paths between these modules (a number that grows exponentially in the lengths of the paths considered), checking that the protocol's behavior is correct only involves the examination of a number of paths that is proportional to the number of messages sent in the protocol. Furthermore, the requisite checking can be performed in a distributed fashion leading to even greater efficiency. Different portions of the network observe different sequences of messages. Thus it is natural to design a highly distributed monitor. Such a design has the added advantage of being efficient and scalable. We can derive checking criteria directly from the protocol graph, and deduce a message flow for each point in the network. These models can be converted to Active Network programs that behave as a set of sophisticated packet filters under the direction of the original protocol.

This checking of state traversals can be applied broadly in an Active Net, hence the name Self-Checking Active Network (SCAN). SCANs can be considered a distributed analogue of watchdogs [25] and some of the key ideas were sketched out (without the availability of Active Networks) by Pickens and Farber in their Overseer [36] scheme. If an invalid sequence is detected, the infrastructure performs an exceptional action, for example, raising an alarm, or blocking further messages. For network security, unusual and unexpected message sequences can be used to detect large classes of intrusions. SCANs provide the ability to protect the network against many active attacks. For example, Keromytis and Smith [21] have developed a method by which arbitrary cryptographic protocols can be made fail-stop. Rather than release information, fail-stop cryptographic protocols terminate when an active attack is detected. The method uses cryptographic hashes to validate sequences of messages by reflecting message dependencies in the hash values. The paper demonstrates the technique on a version of Netscape's Secure Socket Layer (SSL) and suggests some applications to IPSEC. Keromytis' method is easily deployable on SCANs, as it uses a similar model for validity (valid sequences) and checking is easily performed at multiple points in the network, all of which can ''fail-stop'' when attacks are detected. This approach simplifies the task of cryptographic protocol design, as it allows the designer to focus on passive attacks and malicious insiders.

SCANs can perform autonomous local activities which lead to a desired global behavior. For example, consider a denial-of-service attack based on repeated attempts to open a TCP connection without subsequent action [1]. A SCAN could specify TCP's expected behavior [46] using a state graph in which a `SYN` must result in further actions, *e.g.*, a `SYN-ACK` reply followed by an `ACK`. When intermediate SCAN nodes see an invalid state sequence (`SYN, SYN,...`) they can discard subsequent packets. This results in logically ''pushing'' the attack back to its entry point into the SCAN, and avoids wasting the resources of many intermediate nodes.

3.5. Other applications

Another application of the multiple channel approach is for reliability. Consider the two interconnected *SwitchWare* switches shown in **Figure 1**. If three channels worth of capacity are required, we can implement the striping algorithm on the three channels, and utilize the fourth as an error/loss correction channel, as in RAID systems [20]. So, for example, we could (using *SwitchWare*'s capacity for processing), for each three packets sent on the three stripes, compute a fourth packet consisting of the Exclusive-OR of the three packets comprising the stripe. Then, if any 3 of the four packets arrive in time, the data can be recovered and forwarded.

Such modules can carry out many tasks. For example, consider the sensor fusion required to detect an automobile on the other side of a bend; a CCD camera, IR camera (at night) or other sensor could be feeding a broadcast network. An application injected into the network by your automobile could run a motion detection algorithm on the real-time video feed and signal a monitor in the automobile with an approach speed indication or warning tone. An actuator for a rear window defroster in a car-area network might fuse information from a smart thermometer with light diffusion measures to automatically turn on; directional remote motion detection could dim the high beams, etc. Another example is personal multicast topologies; it is easy to write a small program which moves itself from *SwitchWare* switch to switch [41], replicating itself selectively to output ports to create a per-packet multicast.

Still more applications include:

- Speech coding conversion for interoperation of national telecommunications infrastructures; this would be accomplished with *SwitchWare* libraries or DSPs if higher performance is needed.
- Self-adaptation of packets to network dynamics such as failure and congestion, as they could carry algorithmic code specifying appropriate responses to failures.
- Subnet-specific compression, as bandwidth and latency characteristics dictate how much effort should be spent compressing.
- Data type-specific routing and stream synchronization. As an example video frames might choose a higher bandwidth link with a greater loss rate, while motion control streams for interactive telerobotics [4] would select a path with low bandwidth but high reliability and low delay jitter.

4. Security and *SwitchWare*

Security of information means that the right information gets to the right people at the right place at the right time, meaning that security failures occur when these conditions are not met, *i.e.*, wrong people, wrong place, wrong information, wrong time. Security failures can include unauthorized viewing of information, denial of service [32], and insertion of false information. These sorts of failures [9] will become more common unless security is *designed* into a system.

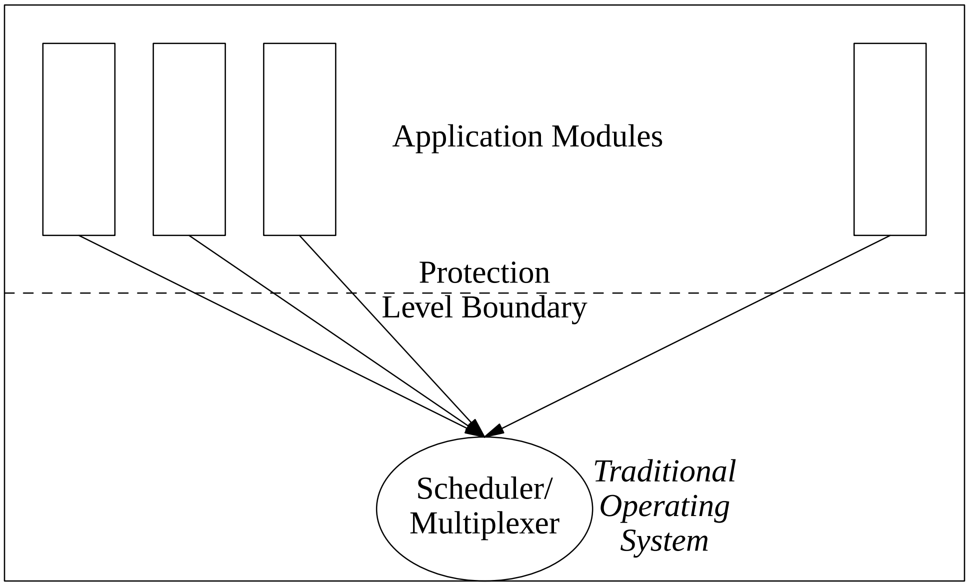

Figure 5: A multi-application's elements share a processor

While cryptography provides potential end-to-end privacy, it has no effect on denial-of-service attacks, which can prevent correct and timely delivery of important information; such attacks must be precluded. Consider for example the difficulty of preventing traffic analysis when packet switching is used. Typically, messages or packets must have headers in the clear, even if the data portion is protected by a cryptographic privacy transformation. It is easy to imagine a sequence of packets where the first packet contains a program capable of obtaining a key from a trusted authority, used by the *SwitchWare* to decode the headers of subsequent packets in the train.

Active networks offer the network users a powerful tool for improving network performance and flexibility. However, the powerful capabilities of the system provide powerful tools to malicious intruders. Consequently, network security and authentication become correspondingly more important. Network elements must assure that any code they execute was produced by an authorized source. Also, any fault detection and management systems must be able to verify the validity of any network monitoring data that are received from network elements.

Although security and authentication mechanisms are being proposed in many networking forums, active networking may allow us to design a single integrated security mechanism for *all* network resources. This eliminates the need for multiple security/authentication systems that operate independently at each communication protocol layer. It would also allow us to address the traditional need for separation of the transport and management planes, which have been separate for reasons of security, performance and modularity. The difficulty is the resource management. Any switching system, no matter how simple or complex, represents a *multi-application*, consisting of a number of tasks, which may be concurrent or provided with the illusion of being so via time-division multiplexing of a shared element. **Figure 5** illustrates a multiapplication mapped onto a single shared processor.

The dashed line of **Figure 5** illustrates a crucial design decision; to control multiplexing of the machine resources, *e.g.*, to associate code activations with interrupts, to operate on network adapters and persistent state resident on secondary storage, an *operating system* is used to create a resource protection boundary between applications, which have access to a ‘‘virtual machine’’, and the multiplexing mechanism, which has access to all system resources and provides the virtual machine.

4.1. Access Control by O.S. or Compiler? — Multiplexing Implications

Unfortunately, this multiplexing architecture has severe performance limitations, in particular for the boundary crossing operation between the application and the operating system (the ‘‘kernel’’). Multiplexing performance is crucial in switching. A great deal of recent research has tried to alleviate these costs while preserving the protection semantics of the operating system [13, 14]. To obtain an order of magnitude estimate of the penalty for this boundary crossing, we compared system calls with an ideal scheduling method, *i.e.*, co-routine scheduling. The method used was the facilities `setjmp()` and `longjmp()` provided by the C library. They provide the ability to achieve a non-local goto, that is, one which crosses routine boundaries. `setjmp()` saves the current "state" of the program (*i.e.*, a minimal set of registers, floating registers, frame pointers, etc) into a `jmp_buf` structure (described in `/usr/include/setjmp.h`) and `longjmp()`, given a `jmp_buf`, restores the specified state, including the program counter.

On an SGI Challenge L, when the program is run on a 3,334,216 octet file (`/unix`), it requires 2.43 seconds of execution time; a version of the program which merely reads and writes takes 0.69 seconds. Counting two context switches per character, we get (2*3334216)/(2.43-0.69) or 3,832,432 contexts per second, or 260 nanoseconds per context switch. Measurements of a microbenchmark which reads 0 bytes from `/dev/null` repeatedly show that each `read()` requires 17 microseconds, with 14.7 microseconds consumed by the operating system. This suggests that we can context-switch between threads *at least 60 times faster* if we get the programming language model right, and further optimizations should be able to reduce costs to approximately a procedure call.

Allowing the user to program and extend the basic network fabric provides great flexibility and power, but as with any power tool it also creates a safety hazard. It is possible (likely) that programs down-loaded into a switch or router, could interfere with, corrupt, or subvert the traffic of other users. Thus a key question in the design of *SwitchWare* is how this power can be provided safely.

4.2. Systems Security and Programming Environments

Familiarity with the Internet Worm [42] or recent security problems [11] found in systems such as Netscape’s Web browser and the Java [26] highlight the importance of security in distributed computing. Although these problems manifested themselves as security breaches, many of them were a result of the lack of safety features in the programming language, notably C. Languages like SML and Java avoid these problems by supporting *pointer safety*. In pointer safe languages, pointers *cannot* point to invalid locations in memory, thus avoiding "core dumps" and array overruns. The key features needed for pointer safety are strong (though not necessarily static) type checking, array bounds checking and automatic storage management or *garbage collection*.

4.3. Formal Semantics of Programming Languages

‘‘Formal Methods’’ is a rubric used to refer to a collection of techniques which seek to apply ideas from formal mathematical logic to computational problems arising from hardware or software. Such techniques have been an active area of investigation for at least two decades. Although not a panacea, the techniques do have the potential of being quite useful, especially in areas of program specification, hardware verification, and language design.

In the area of language design, research in *programming language theory* has developed a collection of tools appropriate to the mathematical specification of programming languages. The value of such a specification is that it makes properties of the language, the programs written in it, and its compilers amenable to rigorous or (in limited cases) automated proof. Formal treatments have been provided for most widely-used languages. For instance: DoD commissioned the completion of such a semantics for a substantial portion of its Ada language while, more recently, C++ and other object-oriented languages have been the subject of focused attention.

It has been less common for a programming language to be *developed* in the context of considerations from programming language theory—language designs are usually more influenced by programming and compilation issues. However, accounting for theoretical considerations as part of a design has significant advantages if ensuring certain properties of (programs written in) the language is of paramount concern. In particular, this is the case when there is a strong need to guarantee various safety or security constraints. As a motivating example, the programming language Standard ML (SML) is descended from a Meta-Language (ML) used to guide a goal-directed theorem-proving system [16]. The standard [27] was completed in 1987 and is described via a set of mathematical rules. Since the soundness of the language as a theorem-proving vehicle was a paramount early concern, the semantics of the language was constructed with great rigor and attention to detail. Consequently, it is one of the most rigorously designed languages being used in significant programming projects. It has, for instance, been of interest to DARPA, which has funded research on its potential use in systems and network programming [18].

We would like to apply techniques similar to the ones used to design and specify SML to similar goals for the *SwitchWare* language. This will make it possible to apply a collection of techniques developed by the programming language theory community to the language. In particular, it will be possible to formulate and prove various safety and security properties based on the language definition. This will ensure that programs written in the language and evaluated with a correct interpreter will respect such properties. Proofs of this kind cannot be viewed as a ‘silver bullet’—they will be limited in scope and difficult if the *SwitchWare* language is large—but researchers have had success with the development of appropriate mathematical techniques and marshalling of automated tools to attack such problems for various languages. In particular, work at Penn under the supervision of Carl Gunter has had significant success with SML, which should form a solid starting-point for work on the *SwitchWare* language, which will be implemented as part of the experimental effort in this project.

4.4. Authenticated Type-checked modules

When we apply mathematical methods to the context of a highly-available distributed switching fabric, which depends on type-checking, we must face the challenge of making the formal guarantees in the face of threats in the network [31]. Several authors have addressed the need for secure object storage [15] in such an environment, and new cryptographic technologies [10] for digital signatures are applicable to this environment; in particular a type-checked module can be stored in a machine-independent form, which is then either signed directly or supported by a secure hashing algorithm. New technologies are becoming available for message-hashing, such as MD5, which can be very helpful in distributed type-checking. A trusted authority is referenced as part of loading a new module into the system. This work can easily build on existing work for distributing loadable modules [22]. A rogue loadable module can be looked at as a particularly harmful form of virus, one introduced directly into the network infrastructure, so we can draw on the considerable work [38] focused on this topic.

5. Concurrency and garbage collection

Garbage collection is crucial because it avoids the possibility that storage will be returned to the memory allocator while it is still in use. Using garbage collection also avoids the possibility that unused storage will not be returned to the allocator, avoiding the problem of "memory leaks". Even slow leaks can cause long-lived servers to crash and they also cause systems to use resources unnecessarily.

Users of `emacs` know that garbage collectors typically stop the client program while reclaiming storage, creating a *garbage collection pause*. These pauses can be of arbitrary length and several second pauses are not uncommon. While annoying to the users of interactive programs, they can be catastrophic to real-time control programs. Consider, for example, a computer-augmented jet fighter occasionally losing control for a few seconds at Mach 2! And yet, such applications are also ones in which freedom from crashes related to pointer errors is highly desirable. The basic technique for eliminating pauses is to allow the collector to run concurrently with the client, as discussed next.

Threads are provided in Java, thus providing low-level support for parallelism; it seems likely that this will be one of the low-level mechanisms used by parallel applications. Unfortunately, the degree of concurrency offered by such an implementation is limited by the need to garbage collect the store sequentially. Nettles, *et al.*, have developed a new concurrent GC technique, *replicating collection* [34]. Based on ideas from distributed systems, replicating collection is a simple and and elegant solution to the difficult problem of making copying collection concurrent. It has been implemented in the runtime of SML/NJ on both DEC uniprocessors running Mach and on SGI multiprocessors using IRIX. The results of the implementation show that GC can make good advantage of parallel machines, thus eliminating the concurrency bottleneck caused by garbage collection. More importantly, the results show that replicating collection is very successful at eliminating the long pauses often associated with garbage collection. These pauses are a substantial reason for high-level languages not being used for performance critical applications [33]. These techniques are applicable to other garbage-collected languages like Java and should greatly improve the performance of garbage-collected languages, and allow significant speedups on multiprocessors.

6. Related research

Borenstein’s ATOMICMAIL [7] system used LISP functions embedded in electronic mail messages, to support overlay functions such as automatically generated mailing lists and software distribution via e-mail. Considerable value stemmed from combining message transport with programs applied to interpreting the messages, especially for widely heterogeneous user environments.

The SOFTNET [50] system was a packet radio network where packets of multithreaded M-FORTH code were interpreted by network elements consisting of two-processor nodes; one serviced network events, and the other ran user programs. The nodes were supported by a small operating system, which protected the network elements, *e.g.*, to prevent buggy programs from destroying the packet-switching fabric. The focus was proof-of-concept rather than a wholesale change in network infrastructure, models and run-time support.

Erlang [3] is a concurrent functional programming language for large industrial real-time systems, providing transparent cross-platform distribution, primitives for detecting run-time errors, real-time GC, and dynamic code replacement. Erlang has been deployed in switches built by Ericsson. It does not provide the strong static type checking we propose in our approach.

A previous Bellcore project, the Touring Machine, is a distributed multimedia communication system which supported 150 users in both point-to-point communications and broadcast meetings and lectures. The architecture has many similarities to an active network. Network elements, such as the end nodes, switches, and audio/video bridges, all have associated processors. All communication functions, such as connection setup/tear-down, were performed by sending blocks of executable LISP code to various processor platforms in the network. There was no formal model and no abstraction useful for security and interoperability validation developed.

The SPIN [5] Project is an effort to build extensible operating systems kernels, with the idea that type-safe Modula-3 code could be loaded into an operating system for reasons of performance or access to resources. This work reinforces our belief that type-safe modern programming languages are a fertile ground for systems programming in even the most performance-sensitive environment. While it is unlikely we can directly employ any of the code produced by the SPIN Project in our efforts, we expect that interactions with like-minded researchers will be valuable. The setting of a switch infrastructure has different challenges, including the need for resource partitioning algorithms, distributed loading of type-checked modules, security and a high degree of multiplexing/ parallel processing, that are less pressing for workstations.

The Scout Project [29] at the University of Arizona uses an algorithm, pathfinding, to optimize the paths through protocol executions in a realization. This is a valuable technology that could be employed in the building of *SwitchWare*, but does not directly address the algorithmic, security and management issues we face in the design of an on-the-fly upgradable network infrastructure. We believe that while Scout itself may be able to operate across many environments, it is providing a level of abstraction that is too low to gain the interoperability advantages of our extensions to SML/NJ.

The Exokernel [14] project at MIT has been focusing on an operating system restructuring, where much of the operating system functionality is carried out in libraries. There is still, for security, a need for a small kernel. We believe that the protection kernel approach has some fundamental performance limitations, especially as regards the high degree of multiplexing found in a network switch. We believe that as the Exokernel architects attempt to re-virtualize the O.S. functions, for example by providing multiplexing of an adapter with a processor embedded in the adapter, that they will run into problems either with the level of abstraction (and therefore interoperability) or with performance barriers that are unavoidable on today’s hardware. What it seems likely they will contribute is a great deal of knowledge on how to craft systems which provide dedicated application access to adaptors, a model which the *SwitchWare* run-time may employ.

The FOX Project at CMU [18] is likely to be an essential supplier of technology. To somewhat oversimplify, the research group at CMU has been focused on evangelizing SML to the systems community, and they have been doing this by focusing on interesting problems such as writing a TCP/IP in Standard ML [6]. We look at the CMU work as providing tools. Their implementation ideas for compilers [44] and run-time environments [23] can be viewed as aids and assists to providing a high-performance implementation of our *SwitchWare* language system; in essence, our SML/NJ extensions for *SwitchWare* ride the compiler technology curve as well. Our run-time research compliments their research, and our setting, a high-performance switch as part of an active network infrastructure, draws on the strengths of our group. Our focus in programming language semantics allows us to attack the theoretical problems in a restricted context, that of an Active Network Switch, that increases our chances of success.

Turner’s group [35] at Washington University propose an approach of interconnecting powerful general purpose processors with an ATM switching system. Thus, the hardware has the ability, in principle, to execute *SwitchWare*-like software. We believe that the security of this approach (both for access control and resource denial, such as bandwidth starvation) is dependent on the security of UNIX rather than provable statements about security enforced at compile time, as in our approach. If you can do it once in the compiler, why do it repeatedly at run time?

7. Status of the Project

Although *SwitchWare* will depend critically on formal and mathematical techniques, we plan to pursue an experimental approach for the project as a whole. We have structured our work using the three language model shown below:

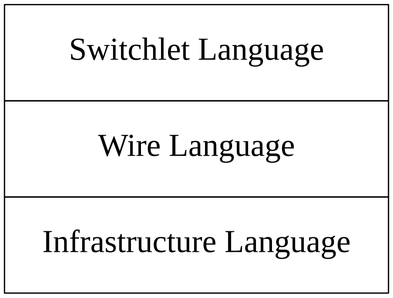

Figure 6: The 3-language model of a *SwitchWare* switch

To understand what we mean by these three language levels, the Switchlet language is the language with which users can access the programmable features of the *SwitchWare* switch, the wire language is the form in which the Switchlets are moved between switches, and the infrastructure language is the language in which the SwitchWare switch itself is programmed. One example of three languages might be a Java program written by a user, its byte code form, and the C language programs which comprise the byte code interpreter. This is fact might provide a straightforward early implementation of *SwitchWare* ideas, except for the high hurdles of security and formal semantics which we set for ourselves. Some early implementations of active network prototypes, *e.g.*, the "Active IP Option" of Tennenhouse and Wetherall [45], use interpreters such as TCL.

Our target for the three languages is a restricted version of ML as a Switchlet languages, with the restrictions implemented via strong type checking. The wire language will be an intermediate language in which the semantic properties of the Switchlet language will be preserved with type enforcement. The loadable language modules would be transported between *SwitchWare* switches, forming trains of active packets, comprising *Switchlets* in the wire language. To preserve the power of the semantic model, type-checked modules can be digitally checksummed with a Secure Hash Algorithm provided by a trusted authority. This takes advantage of the fact that it is easier to verify the proof than to do the proof.

We believe that ML can be used as an infrastructure language, albeit with some non-typesafe calls to low-level device drivers. The programming language implementation challenge in *SwitchWare* will be providing good performance for SML when it is used as a systems programming language. A recent implementation of SML, the TIL compiler at CMU by Morrisett and Tarditi [44], strongly suggests that SML implementations with performance similar to C are feasible.

7.1. A Preliminary Experiment

To gain some experience and to rank order the issues to be addressed in a full-scale implementation, we are building a small prototype switch, with initial software supporting 100 Mb/s bridging. The prototype is now bridging successfully, performs self-learning, and we are implementing a fairly complete subset of 802.1D spanning tree protocols for experimental purposes. All of the bridging functions are loadable, and are programmed in CAML, a dialect of ML. Thus CAML is the Switchlet language, CAML bytecodes are the wire language, and C (in the form of a LINUX kernel) is the infrastructure language. The system is running today on a 4 processor HP Netserver with 166 Mhz Pentium processors [2], and we expect to have IP routing features operational to report in a final paper. The idea with the multiprocessor is that the processors act simultaneously as port controllers and execution engines for the language.

8. Conclusions

SwitchWare is an attempt to overcome the flexibility limitations inherent in today's approaches to internetworking. The project attacks the design of network infrastructure using ideas from communications and computer science, and exploits the latest advances in programming language technology, which allow many run-time checks to be replaced with static checking.

SwitchWare, and *SwitchWare*-like approaches, enable an acceleration in network evolution. This dramatic speedup is the most important factor in bringing new services and capabilities to all of us in the future.

9. Acknowledgments

Many conversations and e-mail exchanges with David Tennenhouse have helped refine our approach. Bryan Lyles pointed us to the inspirational Softnet work at Linkoping. Insup Lee and Sampath Kannan are working with us on the Self-Checking Active Network application of *SwitchWare*. Marianne Shaw is helping in development of the Active Bridge mentioned in the paper.

10. References

[1] *CERT Advisory CA-96.21: TCP SYN Flooding and IP Spoofing Attacks*, Software Engineering Institute Computer Emergency Response Team, Pittsburgh, PA (Sep. 19, 1996). `ftp://info.cert.org/pub/cert_advisories/CA-96.21.tcp_syn_flooding`

[2] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith, "Active Bridging," in *Proceedings, ACM SIGCOMM Conference*, Cannes, FR (October 1997), pp. 101-111.

[3] J. Armstrong, M. Williams, and R. Virding, *Concurrent Programming in Erlang,* Prentice Hall (1993). ISBN 13-285792-8

[4] Ruzena Bajcsy, David J. Farber, Richard P. Paul, and Jonathan M. Smith, "Gigabit Telerobotics: Applying Advanced Information Infrastructure," in *1994 International Symposium on Robotics and Manufacturing*, Maui, HI (August 1994).

[5] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain, CO (December 1995), pp. 267-284.

[6] E. Biagioni, "A Structured TCP in Standard ML," in *Proceedings, 1994 SIGCOMM Conference*, London, UK (Aug. 31st - Sep. 2nd, 1994), pp. 36-45.

[7] Nathaniel S. Borenstein, ‘‘Computational Mail as Network Infrastructure for Computer-Supported Cooperative Work,’’ in *Proceedings, Computer Supported Cooperative Work Conference*, Toronto, CANADA (1992).

[8] Vinton G. Cerf, ‘‘Multiprocessors, Semaphores and a Graph Model of Computation,’’ Technical Report UCLA-ENG-7223, University of California, Los Angeles, Los Angeles, CA (April, 1972). Ph.D. Thesis

[9] System Security Study Committee - National Research Council, *Computers at Risk: Safe Computing in the Information Age,* National Academy Press (1991).

[10] G. Davida, Y. Desmedt, and B. Matt, ‘‘Defending Systems Against Viruses through Cryptographic Authentication,’’ in *Proceedings, IEEE Symposium on Security and Privacy* (1989), pp. 312-318.

[11] D. Dean and D. Wallach, ‘‘Security Flaws in the HotJava Web Browser,’’ Technical Report, Princeton University, Computer Science (November 3rd, 1995).

[12] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei, ‘‘An Architecture for Wide-Area Multicast Routing,’’ in *Proceedings, 1994 SIGCOMM Conference*, London, UK (Aug. 31st - Sep. 2nd, 1994), pp. 126-135.

[13] P. Druschel, L. L. Peterson, and B. S. Davie, ‘‘Experiences with a High-Speed Network Adaptor: A Software Perspective,’’ in *Proceedings, 1994 SIGCOMM Conference*, London, UK (Aug. 31st - Sep. 2nd, 1994), pp. 2-13.

[14] Dawson R. Engler, M. Frans Kaashoek, and James O’Toole, Jr., ‘‘Exokernel: An Operating System Architecture for Application-Level Resource Management,’’ in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain, CO (December 1995).

[15] Virgil D. Gligor and Bruce G. Lindsay, ‘‘Object Migration and Authentication,’’ *IEEE Transactions on Software Engineering* **SE-5**(6), pp. 607-611 (November 1979).

[16] M. J. C. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF,* Springer (1979).

[17] M.E. Grzelakowski, J.H. Campbell, and M.R. Dubman, ‘‘DMERT Operating System,’’ *Bell System Technical Journal* **62**(1), pp. 303-323 (January 1983).

[18] `http://www.cs.cmu.edu/afs/cs/project/fox/mosaic/HomePage.html`, *Fox Project, CMU School of Computer Science*, 1995.

[19] J.R. Kane, R.E. Anderson, and P.S. McCabe, ‘‘Overview, Architecture, and Performance of DMERT,’’ *Bell System Technical Journal* **62**(1), pp. 291-302 (January 1983).

[20] Randy H. Katz, Garth A. Gibson, and David A. Patterson, ‘‘Disk System Architectures for High Performance Computing,’’ *Proceedings of the IEEE* **77**(12) (December 1989).

[21] A. Keromytis and J. Smith, ‘‘Creating Efficient Fail-Stop Cryptographic Protocols,’’ Technical Report MS-CIS-96-32, CIS Dept., University of Pennsylvania (December 2, 1996).

[22] Frederick Colville Knabe, ‘‘Language Support for Mobile Agents,’’ CMU-CS-95-223, CMU School of Computer Science (December 1995). Ph.D. Thesis

[23] Mark Leone and Peter Lee, ‘‘Optimizing ML with Run-Time Code Generation,’’ in *Proceedings, ACM SIGPLAN PLDI ’96* (May 1996).

[24] H. Lycklama and D.L. Bayer, ‘‘The MERT Operating System,’’ *Bell System Technical Journal* **57**(6, Part 2), pp. 2049-2086 (July/August 1978).

[25] Aamer Mahmood and E. J. McCluskey, ‘‘Concurrent Error Detection Using Watchdog Processors - A Survey,’’ *IEEE Transactions on Computers* **37**(2), pp. 160-174 (February, 1988).

[26] Sun Microsystems, ‘‘The Java Language: A White Paper,’’ `http://java.sun.com`

(1995).

[27] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML,* The MIT Press (1990).

[28] R. W. Mitze, H. L. Bosco, N. X. DeLessio, R. J. Frank, N. A. Martellotto, W. C. Schwartz, and R. W. Wolfe, ‘‘3B20D Processor and DMERT as a Base for Telecommunications Applications,’’ *Bell System Technical Journal* **62**(1), pp. 181-190 (January 1983).

[29] A. B. Montz and D. Mosberger, S. W. O’Malley, L. L. Peterson, T. A. Proebsting, J. H. Hartman, ‘‘Scout: A communications-oriented operating system,’’ Technical Report 94-20,, Department of Computer Science, University of Arizona (June 1994).

[30] MTSs, *Engineering and Operations in The Bell System,* AT&T Bell Laboratories, Murray Hill, NJ (1983). ISBN #0-932764-04-5

[31] R. Needham and M. Schroeder, ‘‘Using Encryption for Authentication in Large Networks,’’ *Communications of the ACM* **21**(12), pp. 993-999 (December, 1978).

[32] Roger M. Needham, ‘‘Denial of Service: An Example,’’ *Communications of the ACM* **37**(11), pp. 42-46 (November 1994).

[33] S. Nettles and J. O’Toole, ‘‘Real-Time Replication Garbage Collection,’’ in *SIGPLAN Symposium on Programming Language Design and Implementation,* ACM (June 1993).

[34] James O’Toole, Scott Nettles, and David Gifford, ‘‘Concurrent Compacting Garbage Collection of a Persistent Heap,’’ in *Proceedings, 14th ACM Symp. Operating Syst. Principles* (December, 1993), pp. 161-174.

[35] Guru Parulkar, Douglas C. Schmidt, and Jonathan S. Turner, ‘‘a^{I}t^{P}m: a Strategy for Integrating IP with ATM,’’ in *Proceedings, SIGCOMM 95*, Cambridge, MA (Aug. 28th to Sept. 1, 1995), pp. 49-58.

[36] J. R. Pickens and D. J. Farber, ‘‘The Overseer: A Powerful Communications Attribute for Debugging and Security in Thin-Wire Connected Control Structures,’’ in *Proceedings, International Computer Communications Conference* (August, 1976). also TR #75, UC Irvine, 1975

[37] J. Postel and D. J. Farber, ‘‘Graph Modeling of Computer Communications Protocols,’’ in *Proc. of the Fifth Texas Conference on Computing Systems*, University of Texas, Austin, TX (October, 1976), pp. 66-77.

[38] M. Pozzo and T. Gray, ‘‘A model for the containment of Computer Viruses,’’ in *Second Aerospace Computer Security Applications Conference* (December 1986), pp. 11-18.

[39] Bell Communications Research, Inc., ‘‘AIN Release 1 Service Logic Program Framework Generic Requirements,’’ FA-NWT-001132.

[40] A. Romanow and S. Floyd, ‘‘The Dynamics of TCP Traffic over ATM Networks,’’ in *Proceedings, 1994 SIGCOMM Conference,* London, UK (Aug. 31st - Sep. 2nd, 1994), pp. 79-88.

[41] John F. Shoch and Jon A. Hupp, ‘‘The Worm Programs - Early Experience with a Distributed Computation,’’ *Communications of the ACM* **25**(3) (March 1982).

[42] Eugene H. Spafford, ‘‘The Internet Worm: Crisis and Aftermath,’’ *Communications of the ACM* **32**(6), pp. 678-687 (June 1989).

[43] Adobe Systems, Inc., *PostScript Language Reference Manual,* Addison-Wesley, Reading, MA (1985).

[44] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Bob Harper, and Peter Lee, ‘‘TIL: A Type-Directed Optimizing Compiler for ML,’’ in *Proceedings, ACM SIGPLAN PLDI ’96* (May 1996).

[45] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, ‘‘A Survey of Active Network Research,’’ *IEEE Communications* **35**(1), pp. 80-86, (Earlier version MIT LCS TR #MIT/LCS/TM-557, 1996) (January, 1997).

[46] R. S. Tomlinson, ‘‘Selecting Sequence Numbers,’’ in *Proc. SIGCOMM/SIGOPS IPC Workshop*, Santa Monica, CA (March, 1975).

[47] W.N. Toy and L.E. Gallaher, ‘‘Overview and Architecture of the 3B20D Processor,’’ *Bell System Technical Journal* **62**(1), pp. 181-190 (January 1983).

[48] C. Brendan S. Traw, *Applying Architectural Parallelism in High Performance Network Subsystems*, CIS Dept., University of Pennsylvania (1995). Ph.D. Thesis

[49] C. Brendan S. Traw and Jonathan M. Smith, ‘‘Striping within the Network Subsystem,’’ *IEEE Network*, pp. 22-32 (July/August 1995).

[50] J. Zander and R. Forchheimer, ‘‘Softnet - An approach to Higher Level Packet Radio,’’ in *Proceedings, AMRAD Conference*, San Francisco (1983).