OS Support for General-Purpose Routers

Larry L. Peterson, Scott C. Karlin, and Kai Li
Department of Computer Science
Princeton University

Abstract

This paper argues that there is a need for routers to move from being closed, special-purpose network devices to being open, general-purpose computing/communication systems. The central challenge in making this shift is to simultaneously support increasing complex forwarding logic and high performance, while using commercial hardware components and open operating systems. This paper introduces the hardware and software architecture for such a general-purpose router. The architecture includes two key innovations. First, it better integrates the router's switching capacity and compute cycles. We expect this to result in significantly better scaling properties, and an order of magnitude improvement in performance for packets that require only minimum processing cycles. Second, the architecture supports a hierarchy of forwarding paths, ranging from fast/fixed paths implemented entirely in hardware to slow/programmable paths implemented entirely in software, but also including intermediate paths that exploit the improved integration of cycles and switching.

1 Introduction

Much of the success of the current Internet can be traced to the elegance of its architecture: *routers* in the middle of the network simply forward packets to implement a best-effort delivery service, while *hosts* at the edges of the network address the more complex end-to-end issues (e.g., reliability, ordered delivery, security), as well as implement application programs. One consequence of this design is that only those fields needed by routers to forward packets are placed in the IP header; all other information is located in higher-level headers (e.g., TCP, HTTP), and is ignored by the routers. This allows routers to remain simple—they only forward packets based on the destination address contained in those packets.

Today, however, one does not have to look very hard to find examples of where this architecture breaks down.

- Many routers sit at the boundary between different regions of the Internet; for example, between a site and an ISP, between a tethered network and a wireless network, between an ISP and a dialup line, and so on. It is often the case that different assumptions hold on either side of such a router, and as a consequence, additional functionality is loaded into the router to transition packets from one region to the other. For example, a firewall is a boundary router that filters packets that flow from an untrusted region to a trusted region; a NAT translates from one address space to another; a QoS broker translates between QoS reservations; and a transcoder thins a data stream going from a high-speed link to a low-speed link [1].

- A router might function as the front end to a scalable server. Such a router is asked to make complex forwarding decisions based on application-specific knowledge of the packet stream. In the case of a scalable storage server, for example, the router might stripe outgoing write-streams across a set of servers, order incoming read-streams from a set of servers, and cache meta objects [4]. In the case of a scalable web server, the router might forward web requests to the most appropriate server based on both load and server cache affinity [9]. In the case of a scalable display server, the router might partition a stream of graphic directives or video macroblocks into packets that go to different frame buffers.

- A router might implement an entire LAN. Such a router would offer the performance advantages of a switched network, but also be able to serve as an internal firewall that protects hosts within a site from each other. This would be necessary to protect hosts from mobile code that has been imported into the site, or to isolate portions of a company's computing infrastructure. Such a router might enforce access control, much like a traditional firewall, but it would also authenticate users and hosts. Moreover, since all traffic flows through such router, it has the opportunity to log usage and implement intrusion detection.

One can argue with our calling nodes that implement these functions “routers”; perhaps they are better called *application gateways*, *proxies*, or even *topology-aware servers*. Whatever they are called, however, we recognize a general trend: the logic that decides how to process packets has grown more and more complex over time. It is our contention that the potential for this trend to continue is almost unlimited. Taking this trend to its logical conclusion, one might argue that packets should be allowed to carry code that defines how they are to be processed by the routers they visit; this is the vision of *active networks* [3].

We believe that active networks are a special case of extensible routers: the former support the dynamic loading of untrusted mobile code, while dynamically loading trusted native libraries or having a system administrator reconfigure and reboot the kernel are equally viable alternatives to extending the latter. The point is that being able to support the dynamic loading of untrusted architecture-neutral code (e.g., Java) may prove to be a useful capability—and an extensible router should be general enough to accommodate such an extension—but extensibility does not directly depend on nor require it. In any case, the issue is not so much what types of extensions are allowed, but rather how efficiently the router is able to *apply* the extensions to packets that flow through it.

The paper argues that there is a need for routers to move from being closed, special-purpose network devices to being open, general-purpose computing/communication systems. The central challenge in making this shift is to simultaneously support increasing complex forwarding logic and high performance, while using commercial hardware components and commercial operating systems. We briefly sketch a hardware architecture that supports this goal, and then describes the OS support required to make routers extensible.

2 Scalable Hardware

Many router hardware architectures are possible. At the low-end, routers can be built using a standard workstation running a conventional operating system [10]. Because they are implemented in software, such routers can (and have) been programmed to support nearly every imaginable function, but they offer very limited performance: they can forward 10-100Kpps and support only a handful of 100Mbps ports. At the high-end, routers are designed around special-purpose hardware: they use a switched interconnect (e.g., a crossbar) rather than a shared bus, and they implement the performance-critical forwarding function in hardware (e.g., an ASIC) rather than software [2, 6]. As a consequence of this hardware-intensive design, high-end switches are able to forward packets at a very high rate (on the order of 1–10Mpps) and scale to fairly large sizes (on the order of 128 100Mbps ports), but they provide virtually no programmability.

In addition to the highly optimized forwarding path, high-end routers also have a network processor—usually connected to the one of the switch’s I/O ports—that handles exceptional cases like IP options and routing updates. One possibility is to implement the kinds of services outlined in the previous section on this network processor. Such an approach admits that a network node has two halves—a router half that forwards IP packets with hardware assistance, and a server half that performs computation. These two halves are not tightly integrated, but instead sit side-by-side, as illustrated in Figure 1.

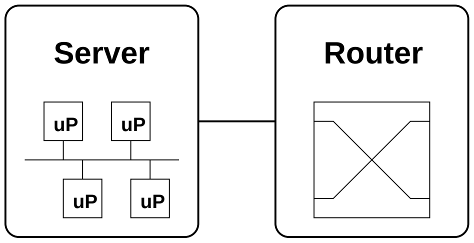

Figure 1: Extending a Conventional Router

The side-by-side design is sufficient if either (1) packets are passed to the server half in only rare cases, for example, to process IP options; or (2) the server half implements functions in the control plane (i.e., flow setup) but is not part of the data plane (i.e., packet forwarding). The side-by-side architecture is less suitable when processing is required on the data plane. This is because such an architecture does nothing to bring the cycles closer to the bandwidth: ASICs that implement the IP forwarding path cannot be extended in even the most trivial way, and the server is connected by a narrow, high-latency pipe to the router’s vast switching capacity.

We are designing an architecture that generalizes this side-by-side architecture to include a hierarchy of processors and switching elements. This architecture better integrates the cycles and the bandwidth, thereby providing a continuum of possibilities between the fast/fixed path and the slow/programmable path. The architecture has three main attributes:

- It is designed around a tightly-coupled cluster of high-end PC systems, where by tightly-coupled we mean that the PCs connect to a very high-speed crossbar switch via their internal system switch.

- Each PC system supports multiple line cards, each of which is implemented by a network interface (NI) that includes a microprocessor. Generally, we expect the line card processors to be two or three generations behind the main PC processors, or approximately four times slower.

- The PC systems and line cards include logic that allow packets to be moved from line card to line

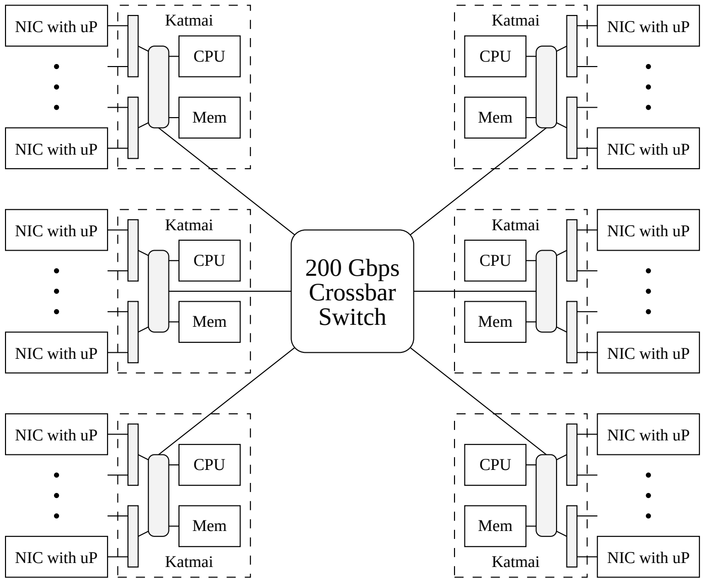

Figure 2: A Katmai-Based Implementation of the Scalable Hardware Architecture (240 × 100Mbps ports).

card—either within the same PC or across the crossbar switch—without the involvement of the main system processor(s).

This architecture has two advantages over the side-by-side architecture shown in Figure 1. First, it has lower latency in the case when one wants to apply a very trivial operation (a handful of instructions) to each packet. This is because such simple operations can be implemented on the line cards rather than the server's processors. This, in turn, means that: (1) the I/O bus needs to be crossed once instead of twice, and (2) a lower-overhead thread model can be used. Lower latency translates into more packets per second; we estimate an order of magnitude difference in this case.

Second, while it is tempting to claim that our architecture provides more bandwidth between the switching element and the processors than does the side-by-side architecture, this is not necessarily the case because it is possible to connect the server half to the router half with multiple I/O ports. The real argument in favor of our architecture is that its throughput has better scaling properties—both in the number of ports and in the number of compute cycles available per port. This is because scaling the side-by-side architecture requires buying a larger switch, while our architecture can be grown to several hundred ports simply by adding additional processors.

Figure 2 depicts an example implementation of our architecture. It consists of multiple 500MHz – 1GHz Katmai (next generation Pentium-II) systems connected by a 200Gbps crossbar switch. Each Katmai system can support four independent 32-bit standard PCI buses, each with 2 Gbps of bandwidth. Each bus is therefore able to support ten full-duplex 100Mbps ethernet line cards or one 1Gbps line card. The system uses bridge chips to connect these buses, such that they can work simultaneously. Thus, assuming 100Mbps ethernet line cards, a single Katmai system can support up to 40 ports. By cascading a collection of crossbar switchs, we expect to have enough switching capacity to support up twelve PC systems, or 480 × 100Mbps ports. The configuration shown in Figure 2 uses a single crossbar switch and supports up to 240 × 100Mbps ports.

In addition to its switching capacity, the six-Katmai configuration (when outfitted with a full complement of 100MHz line card processors) also has 27GHz of aggregate computing capacity. Assuming 1KB packets, this means that 240 separate 100Mbps flows through the router could apply approximately 9,200 cycles to each packet. More generally, we characterize the ratio between computing and communication capacity in terms of *cycles-per-word* (per-second) (CPW), with the hardware configuration shown in Figure 2 having a CPW of 36.

3 OS Support

Our software architecture is based on the Scout operating system [7]. Scout has two relevant attributes. First, it supports extensibility at multiple time scales. Different modules can be configured into the system either statically (requiring reconfiguration and rebooting) or dynamically (either as conventional dynamic libraries or by JIT-compiling Java bytecodes) [5].

Second, Scout supports an explicit *path* abstraction. Paths carry data from input device to output device—from input port to output port—possibly computing on the data along the way. Figure 3 illustrates two different configurations of Scout paths. The example on the left shows a path on a typical end-system: it delivers packets from a network device (ETH) to a frame buffer (VGA), and applies the MPEG decompression algorithm (MPEG) to the data that traverses it. The example on the right shows a path that might run on a firewall: it moves packets between network devices via a proxy (PROXY) that enforces some security policy. In both cases, the path delivers data from an input queue to an output queue (possibly in both directions) by executing a sequence of modules; the path labels in the Figure identify the modules.

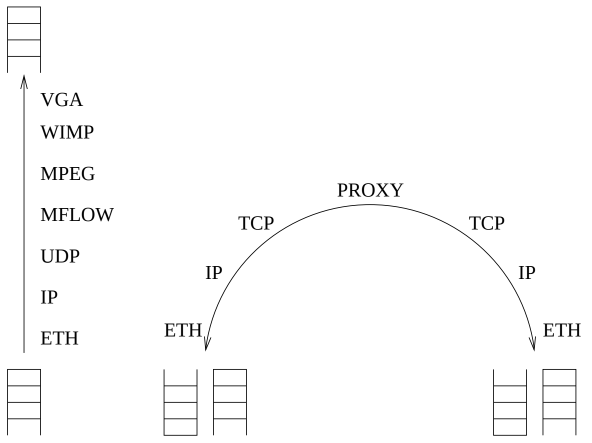

Figure 3: Example Paths

The path configurations shown in Figure 3 run in a standalone implementation of Scout on Pentium workstations. In the case of the router, however, we are running a version of Scout that has been integrated into the Linux kernel. This allows us to leverage exisiting Unix server and router code. When coupled with the hardware architecture described in Section 2, this means that paths can run in one of three different *operating environments*: (1) they run entirely on the line cards; (2) they run entirely in Scout, in kernel mode, on the Katmai processors; and (3) they run as user processes on Linux. The latter two environments have greater cycle bandwidth than the first (by a factor of four), but incur a greater startup latency (by a factor of ten to a hundred, respectively).

The environment in which a path executes is ony one dimension to the problem. In general, the challenge for the OS is to support a rich hierarchy of paths, ranging from fast/fixed paths to slow/programmable paths, with several interesting design points in between. To better understand the richness of this hierarchy, consider that there are two types of computation performed on each packet: *classification* and *processing*. The former determines which path the packet should traverse, while the latter corresponds to the processing that takes place along that path. Classification must happen for every packet; processing may or may not. Note, however, that classification and path execution are not completely independent. It may be necessary to first partially classify the packet based on header fields, thereby selecting a processing path. The processing path might then inspect the packet data in order to fully classify it (i.e., determine how to forward it). The OS supports both a classification hierarchy and a path hierarchy, each of which we now describe in more detail.

3.1 Path Hierarchy

We have already seen that paths may run in one of three different environments, each with different bandwidth and latency properties. There are two additional dimensions along which paths may differ. The first is how much effort has been put into optimizing the path's code. Possibilities range from just-in-time (JIT) generated code to way-ahead-of-time (WAT) generated code, with the latter more heavily optimized for a particular processor. Coarser-grained optimizations that exploit path-specific knowledge—e.g., eliminate redundant or unnecessary functionality—are also possible.

The second dimension is the complexity of the service provided by the path, which may range over the following:

- none (packet is forwarded from input port to output port);
- trivial transformations of header fields;
- logically re-arrange data in packet (involves copying), but no additional cycles per byte; and
- compute (multiple instructions) on every data byte.

Each of these two dimensions must be mapped onto one or more operating environments. For example, our experience shows that it is possible to push the first two levels of processing onto the line cards, but not the third and fourth. Similarly, for security reasons, it is likely that JIT code would be executed in user space, but WAT code could statically loaded into the kernel.

To tie all three dimensions of path processing together, consider Figure 4, which shows five different incarnations of the same path on a router that implements firewall extensions. The outer path includes dynamically-loaded code that implements a firewall proxy. It runs in user space. The second path is the same proxy implemented in a statically loaded module; it runs faster than the outer path since the proxy code is more heavily optimized and it runs in the kernel. The third path is an optimization of both of the first two. It by-passes the proxy, and forwards packets directly from the input device to the output device. This optimization is allowed when the proxy needs to inspect the first packet that flows through the path (e.g., to determine if the request operation was allowed), but after that it simply forwards packets from the input to the output [8]. The third path also runs in the kernel. The two inner-most paths run

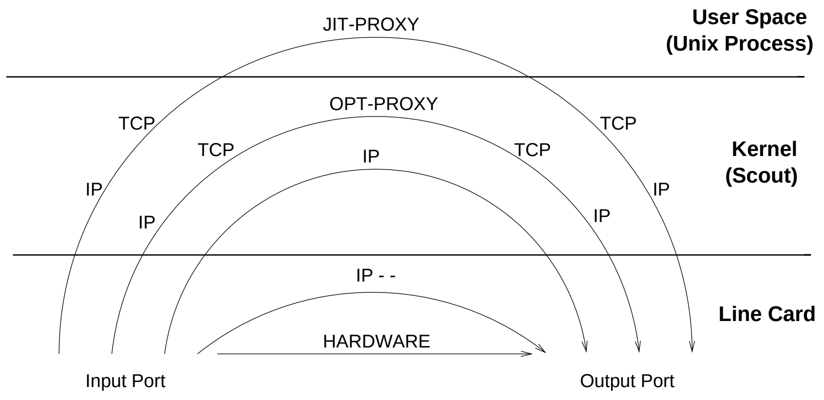

Figure 4: Various levels of path optimization

entirely on the line cards. The one labelled "hardware" is allowed only if packets can be forwarded without modification; they require only classification. The one labelled "IP – –" is more likely. It is an optimization of the third path that trivially augments the common case path by modifying select TCP header fields.

Supporting such a hierarchy requires two things from the OS. First, it must be possible to *replace* one representation of a path with another, thereby making it possible to dynamically optimize a path. Scout already supports such a feature. Second, it must be possible to *load* a path into a particular processing environment. Scout supports such an interface between user space and the kernel; we are currently developing a corresponding interface between the kernel and the line cards.

3.2 Classification Hierarchy

To support fast forwarding of most packets and complex routing (at lower speeds) of some packets, we use a hierarchy of packet classifiers to identify which path to invoke for a given packet. Packets start at the lowest level in the hierarchy where relatively simple and fast algorithms attempt to classify the packet. If a level fails to classify a packet, it is handed off to a higher level in the hierarchy. The higher levels perform more complex and more complete routing decisions than do lower levels. When the proper path has been identified for a given packet, classification completes and that path is invoked.

Figure 5 depicts one level of the classification hierarchy. The level is invoked when a lower level is not able to classify the packet. The lower level passes up a *Packet Description Record* (PDR) for the packet. The PDR includes the packet header, the address of the packet data, and intermediate results for the classification processing done so far. If the classifier is successful, it invokes the resulting path. If not, it passes the PDR (possibly modified with intermediate results) to the next higher level classifier. Because the routing tables are dynamic—and lower levels often cache recently used information—higher levels of the classification hierarchy are allowed to load new state into a particular classification level. This is indicated by "Classifier Updates" in Figure 5.

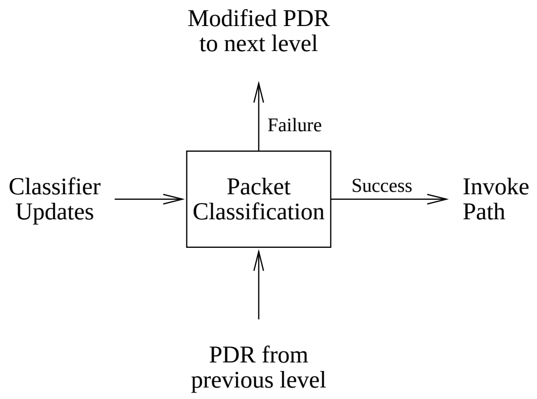

Figure 5: A Classification Level.

In order to constrain the maximum latency and the minimum bandwidth of the system, each classification level is given a *cycle budget* for each packet. For mature classification algorithms, there will be known checkpoints in the algorithm where the amount of time between these points will be known to some degree of certainty. When the algorithm itself determines that it will exceed the cycle budget, the classification is aborted and spills to the next level. The cycle budget can be capped by the use a watchdog timer to prevent experimental classification algorithms from monopolizing the CPU resource.

The classification hierarchy ranges over the following possibilities:

- cache of recent routes;
- pattern-based classification using one or more header fields;
- arbitrary classification code, but limited to header fields; and

- complex classification based on packet data.

Note that the last case is implemented by path; a partial classification of the packet selects this path for execution. We expect the first level to be implemented on the line cards, with the higher levels implemented on the Katmai processors.

References

[1] A. Fox, et. al. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of ASPLOS-VII*, pages 160–170, Oct. 1996.

[2] C. Partridge, et. al. A 50-Gb/s IP Router. *IEEE/ACM Transactions on Networking*, 6(3), June 1998.

[3] D. Tennenhouse, et. al. A Survey of Active Network Research. *IEEE Communications*, pages 80–86, Jan. 1997.

[4] G. Gibson, et. al. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of ASPLOS-VIII*, pages 92–103, Oct. 1998.

[5] J. Hartman, et. al. Joust: A Platform for Liquid Software. *IEEE Computer*, April 1999.

[6] N. McKeown. Fast Switched Backplane for a Gigabit Switched Router. Technical Report Unpublished whitepaper, Cicso, 1998.

[7] D. Mosberger and L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the 2nd OSDI Symposium*, pages 153–167, Oct. 1996.

[8] O. Spatscheck, et. al. Optimizing TCP Forwarder Performance. Technical Report TR98-01, Department of Computer Science, University of Arizona, Feb. 1998.

[9] V. Pai, et. al. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of ASPLOS-VIII*, pages 205–216, Oct. 1998.

[10] S. Walton, A. Hutton, and J. Touch. High-Speed Data Paths in Host-Based Routers. *IEEE Computer*, 30(11):46–52, Nov. 1998.