Making Paths Explicit in the Scout Operating System

David Mosberger and Larry L. Peterson

TR 96-05

Abstract

This paper makes a case for *paths* as an explicit abstraction in operating system design. Paths provide a unifying infrastructure for several OS mechanisms that have been introduced in the last several years, including fbufs, integrated layer processing, packet classifiers, code specialization, and migrating threads. This paper articulates the potential advantages of a path-based OS structure, describes the specific path architecture implemented in the Scout OS, and demonstrates the advantages in a particular application domain—receiving, decoding, and displaying MPEG-compressed video.

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

Layering is a fundamental structuring technique with a long history in system design. From early work on layered operating systems and network architectures [HFC76, Zim80], to more recent advances in stackable systems [Rit84, HP91, HP94, RBF+95], layering has played a central role in managing complexity, isolating failure, and enhancing configurability. This paper describes a complementary, but equally fundamental structuring technique, which we call *paths*. Whereas layering is typically used to manage complexity, paths are applied to layered systems to improve their performance and to solve problems that require global context.

Although paths are easy to understand on the surface—a path is a vertical slice through a multi-layered system—a concrete realization of the idea is surprisingly elusive. We therefore develop a working definition of paths in an incremental fashion. First, consider that the term "path" is well entrenched in our vocabulary. For example, we often refer to the "fast path" or the "critical path" through a system, implying that the most commonly executed sequence of instructions have been optimized. As another example, we sometimes talk about optimizing the "end-to-end path", meaning we are focused on the global performance of the system (e.g., from I/O source to sink), rather than on the local performance of a single component. As a final example, we sometimes distinguish between a system's "control path" and its "data path", with the former being more relevant to latency and the latter more concerned with throughput.

Paths can also be loosely understood by considering specific OS mechanisms that have been proposed over the last few years. Consider the following examples.

- Fbufs [DP93] are a path-oriented buffer management mechanism designed to efficiently move data across a sequence of protection domains.[1] Fbufs depend on being able to identify the path through the system over which the data will flow.

- Integrated layer processing (ILP) [CT90, AP93] is a technique for fusing the data manipulation loops of multiple protocol layers. It depends on knowing exactly what sequence of protocol modules a network packet will traverse.

- Packet classifiers [YBMM93, MJ93, BGP+94] distinguish among incoming network packets based on certain fields found in their headers. In a sense, a packet classifier pre-computes the path that a given message will follow.

- Specialization is sometimes used to optimize common path code sequences [PAB+95, MPBO96]. Specialization, in turn, depends on the existence of invariants that constrain the path through the code that is likely to be executed.

- Spring defines a shuttle [HK93], which is an environment that allows a thread to migrate across a sequence of protection domains; others have defined similar mechanisms [FL94]. Like the previous examples, such mechanisms recognize that programs often follow the same path through the system more than once, and so establish some state that subsequent invocations can exploit.

Note that while all of these mechanisms either support paths or depend on the existence of paths, none of them explicitly define what a path is. In other words, while the idea of a path running through a layered system is widely recognized, to date, paths have only been implicitly defined.

The main contribution of this paper is to show how paths are explicitly implemented in the Scout operating system [MMO+94], and to demonstrate some of the advantages of this implementation by means of an example application. Scout is an experimental framework that provides the means to easily produce small, highly efficient, stand-alone kernels that are targeted at a particular I/O-intensive application domain. Scout is designed for both non-realtime and soft-realtime applications. A description of Scout, with a focus on its architecture for paths, is given in Section 3. The demonstration application, which involves receiving MPEG-compressed video over a network and then decoding and displaying it, is described in Section 4.

[1] Although layering does not imply multiple protection domains, systems often impose hardware-enforced protection at layer boundaries.

2 A Case for Paths

The claim is that paths are a fundamental abstraction in system design, one that provides a unifying framework for many of the mechanisms identified in the introduction. Before describing how paths are implemented in Scout, however, this section first introduces an abstract model for paths, and argues why paths are a good idea in principle. A later section (4) illustrates some of these advantages in the context of a concrete example.

Consider a layered system that consists of a collection of modules that depend on each other in a well-defined way. We call these modules *routers* for reasons that will become clear in a moment, but for now, think of each router as implementing a certain functionality (e.g., the IP protocol or MPEG decompression) and having a well-defined interface. These routers are then connected into a graph to provide exactly the functionality required for a given application domain.

Paths are dynamic entities. They are created and destroyed at runtime as a result of various events such as the arrival of a connection request packet on a network adapter, a user hitting the return key, or a timeout expiring. A path traverses a sequence of routers that are connected in the router graph. Figure 1 illustrates an example router graph with two distinct paths running through it; think of each path as corresponding to a separate network connection, for example.

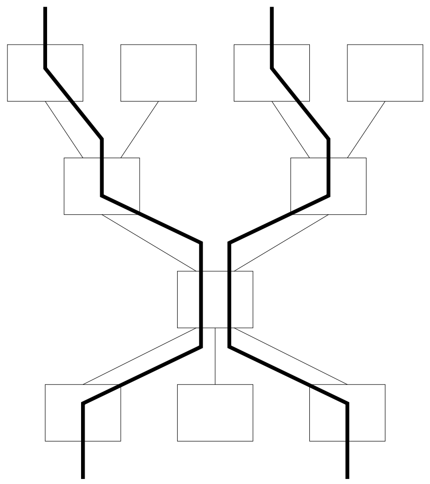

Figure 1: Two Paths Through a Router Graph

A path is created incrementally. It starts at a source router and is specified with a set of *invariants* that will be true for all data that flows over the path. The router uses these invariants to make a *routing decision* (hence the name router)—if the invariants are strong enough, it will be able to determine the next router that must be traversed by any path data. The next router then makes its own routing decision. As long as the invariants are strong enough, the path will keep growing. Eventually, either the path will reach a leaf router (e.g, a device driver), or the invariants will be so weak that no unique routing decision can be made at some router. Thus, a path can be interpreted as a *sequence of fixed routing decisions*. Because paths embody invariants, have an explicit representation, and are known at path-creation time, they can be the subject of optimization.

As we argue below, it is advantageous for paths to be as *long* as possible. However, just how long can a path be? In the best case, a path extends all the way from a source device (e.g., network adapter) to the destination device (e.g., framebuffer). In the worst case, each path spans only a single router, that is, the invariants are so weak that each router is unable to determine a unique subsequent router at path creation time. Note that a conventional layered system *without paths* is equivalent to this degenerate case in the sense that each layer makes a separate routing decision each time it is invoked.

Even in a system that explicitly supports paths, the best-case scenario is not always possible. If a path is created with the invariant that the source device is always the first Ethernet adapter, then it is normally possible to create a path that extends all the way to that Ethernet adapter. But if a path wants to receive packets from any other IP host in the world, then it is generally necessary to assume that *any* network adapter in the system could receive a relevant packet. Thus, the path could reach at most down to IP.

Figure 2 illustrates the worst- and best-case scenarios for a trivial router graph. The worst-case scenario pictured on the left corresponds to a conventional layered system where a routing decision has to be made at each layer (at only one layer in this example); it contains three paths, each of which span exactly one router. The best-case scenario pictured on the right corresponds to the invariants being sufficient to establish an end-to-end path;[2] it contains two paths, each of which span two routers. In both cases, we view a path as being represented by a code sequence that must be executed to move data (e.g., network packets) from a source to a sink. In the figure, we represent the ultimate sources and sinks as queues, which would be the actual implementation if they corresponded to devices.

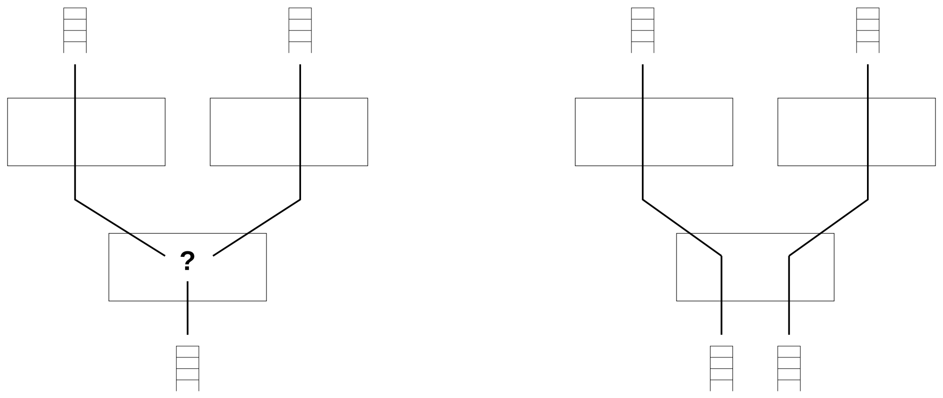

Figure 2: Router Decision at Each Layer versus End-to-End Paths

Making paths as long as possible—preferably end-to-end—means that more non-local context is available. To better appreciate the value of non-local context, consider an Ethernet packet, which by itself, implies nothing about its relative importance compared to other packets. However, if it is known that the packet is part of a video stream, then it is easy to determine the its processing deadline, how many CPU cycles need to be allocated to process it, where the its data should be placed in memory, and so on.

In general, having access to non-local context leads to two kinds of advantages: (1) improved resource allocation and scheduling decisions, and (2) improved code quality. In the former case, work is segregated early, meaning that distinct paths are fed by separate work queues rather than having to share a single work queue. For example:

- The system can place data in a memory buffer that is already accessible to all the routers along the path. This is essentially what fbufs do. In contrast, data often has to be copied (either logically or physically) from one buffer to another at each router where paths split.

- The system can know that a particular path needs to be scheduled for execution in order to meet a deadline; e.g., display a video frame. This is critical to being able to offer different Qualities of Service (QoS). In contrast, having a shared input queue means that low-priority work may need to be done to discover high-priority work that needs attention.

- If scheduling deadlines for a particular path are such that it is impossible to make use of a particular piece of work (e.g., network packet or video frame), then the system can discard unnecessary work early, that is, before executing the path. A conventional system has to at least process the work up to the router before knowing that continuing is of no value.

[2] Throughout this paper we use the term "end-to-end path" to refer to a path between a source device and a sink device *within a single system* , rather than across a network. In future work, we hope to seamlessly integrate intra-machine paths and inter-machine paths.

In the latter case—improved code quality—the system has more information available to it, making more aggressive code optimizations possible. Examples of such optimizations include the following:

- The more invariants the system knows about code to be executed, the more opportunities the system has to specialize the code path. For example, the system can do constant folding and propagation, dead-code elimination, and interprocedural register allocation.
- The more layers across which the system is able to optimize, the more opportunities there are to eliminate redundant work. For example, the more protocol layers available, the more loads and stores integrated layer processing can remove. Similarly, it is sometimes possible to merge per-layer operations. For example, instead of having each layer check for the appropriate header length, it is possible to check for the sum of all header lengths at the beginning of packet processing.

This paper focuses on the first set of advantages, that is, those that have to do with improvements in resource allocation and scheduling. A companion paper demonstrates the code-related improvements attributable to paths [MPBO96]. That earlier paper shows that paths permit code optimizations that improve node-to-node network processing latency by 21–186%. While the companion paper postulates the existence of paths, it does not explicitly define the requisite path architecture. Defining that architecture is one of the contributions of this paper.

3 The Scout Architecture

This section describes Scout's basic architecture, with an emphasis on its support for paths. In the discussion that follows, keep in mind that compatibility with existing standards such as POSIX is not a primary goal. The initial focus of Scout is on application domains such as network appliances—e.g., set-top boxes, file- and web-servers, or cluster computers. While taking the liberty of radically changing the inside of a network appliance, Scout does recognize the importance of interoperability; it does not assume that existing external communication protocols can be abandoned at will.

3.1 Routers and Services

The *router* is the unit of program development in Scout. A router implements some functionality such as the IP protocol, the MPEG decompression algorithm, or a driver for a particular SCSI adapter. A router implements one or more *services* that can be used by other higher-level routers. As is typical in a layered system, most routers themselves use other lower-level routers to implement their services. Scout does not, however, enforce strict layering. Cyclic dependencies are admissible as long as there is a partial (non-cyclic) order in which the routers can be initialized.

Each service in a router has a name and a type. The names are unique, but otherwise arbitrary and chosen by the programmer. The type of a service is used by the Scout configuration editor to ensure that only mutually compatible services are connected.

Figure 3 illustrates routers, services, and how they interact in a router graph. In this partial router graph, IP has three services: *up*, *down*, and *res*. The first two are of type *net* and the latter is of type *nsClient* (for "naming-service client"). The *net* service provides a function to deliver a message (data). Since IP and ETH are both connected through a link connecting a pair of *net* services, this implies that IP can send messages to ETH and vice versa. In contrast, the *nsClient* and *nsProvider* services are asymmetric: IP invokes ARP to translate IP addresses into Ethernet addresses, but there is no need for ARP to ever call back into IP. That is, *nsProvider* provides functions to resolve IP addresses whereas *nsClient* is essentially an empty service.

3.2 Paths

We can now describe the path object. It logically consists of three nested parts. We start at the middle level, which defines the *stage*. Stages provide a locus for storing state that is both path- and router-specific. There is one stage for each router that a path crosses, or said another way, a path contains a sequence of stages. At the innermost level is the

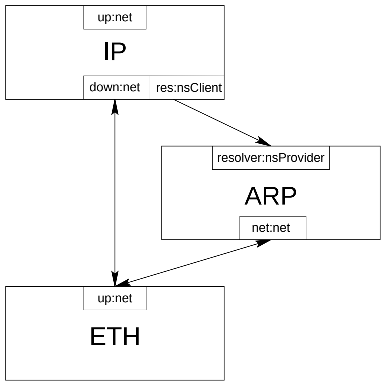

Figure 3: Routers and Services.

interface. Each interface is an instantiation of a corresponding router service. For example, if a path enters the ETH router via the *up* service, then the corresponding interface is of type *net*. Interfaces therefore provide the means for one layer (stage) to communicate with the next layer in a controlled manner. Interfaces are linked together to provide a chain corresponding to a flow of data. Since a path is bi-directional, there is one interface chain for each direction. As it is sometimes necessary to turn data around in a path, each interface also contains a back pointer to the next interface in the *other* direction of data flow. Finally, at the outermost level, is the actual path object—it contains path-global state such as the path id, pointers to the stages at the extreme ends of the path, and a list of attributes for the path. (More on attributes below.)

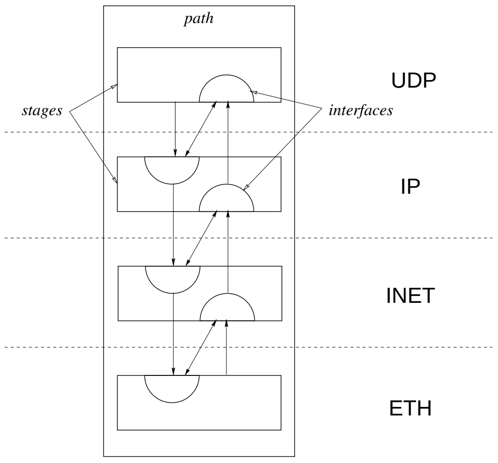

Figure 4: Path structure.

Figure 4 depicts a path that consists of four stages. The stages were created by the UDP, IP, INET, and ETH routers. Each interior stage contains two interfaces (semi-circles), whereas the stages at the extreme ends of the path contain only one interface each. These final interfaces are used to terminate the path.

A path is created by invoking *pathCreate* on a router. The kind of path to be created is described by a set of *attributes*—name/value pairs that specify the invariants for the path. The router uses these attributes to create a stage and to determine the next router that will be crossed by this path, if any. If there is a next router, the *pathCreate* operation is propagated to that router. This process continues until a path reaches its full length, which happens either when it reaches a leaf router or when the attributes are too unspecific to make another unique routing decision. At this point, a sequence of stages has been created. These stages are then linked together into a path structure. Once the path is created, each stage is given a chance to execute initialization code.

Attributes are used in several places, not just as arguments to the *pathCreate* operation. As mentioned above, every path object contains an attribute list which allows storing arbitrary path-specific information. For example, a pointer to the input queue of a path can be stored in the `PATH_SOURCE` attribute. If set, a flow control protocol may use the value of this attribute to learn about the current length of the input queue. In other words, attributes allow efficient and anonymous exchange of information through a common object.

As described so far, path creation consists of three phases: (1) determine route of path and create stages, (2) combine stages into path object, and (3) establish (initialize) stages. During a fourth and final phase, transformation rules are applied to the path. Semantically, transformation rules have no effect but they typically result in better performance and better resource allocation or usage. For example, if a path invocation crosses a sequence of routers for which optimized code is available, the path is modified to use that code. A transformation rule should be thought of as a pair consisting of a *pattern* and a *transformation* that takes a path as an argument and returns a transformed path. A companion paper describes a set of code transformations that can be applied to a path [MPBO96]. In Section 4 we will discuss some transformations that improve resource management.

As implemented in Scout, paths are light-weight. For example, a path to transmit and receive UDP packets consists of six stages. Creating such a path on a 300MHz Alpha takes on the order of 200μs. The path object itself is about 300 bytes long and each stage is on the order of 150 bytes in size (including all the interfaces).

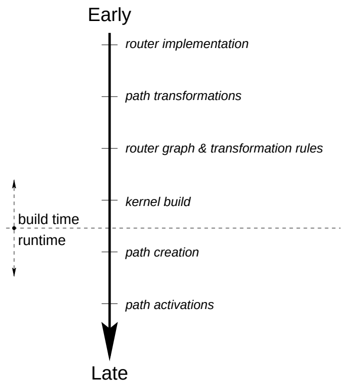

Figure 5: Scout Development Timeline.

To summarize, Figure 5 presents a timeline that illustrates when the various components of a Scout system are created or specified. At the earliest time, individual routers and path transformations are implemented. Then, a system is configured by specifying a router graph and appropriate transformation rules. The kernel is then built and booted. During runtime, paths are created, used, and destroyed in response to events.

3.3 Threads And Scheduling

Threads are the active entities in Scout. While threads usually execute in the context of a path, they are independent objects that exist in their own right. That is, threads execute paths, but paths do not normally own the threads. This implies that a thread can very efficiently cross from one path into a router and then back into another path—an important aspect given that degenerate paths can be short. However, this is not just a matter of efficiency (paths that are short are usually not performance critical anyway) but also of simplicity. Forcing a context switch at the end of every path would be cumbersome, as it would introduce additional synchronization points that serve no real purpose.

A thread is scheduled non-preemptively according to its scheduling policy and priority. Scout supports an arbitrary number of scheduling policies, and allocates a percentage of CPU time to each. The minimum share that each policy gets is determined by a system-tunable parameter. Two scheduling policies have been implemented to date: (1) fixed-priority round-robin, and (2) earliest-deadline first (EDF) [LL73]. The reason for implementing the EDF policy is that for many soft realtime applications it is most natural to express a path's "priority" in terms of a deadline. We present an example of this in the next section.

Scout uses a non-preemptive scheduler because it meets our needs and is easy to use. Programming preemptively scheduled threads that share memory is error-prone [Ous96]. More importantly, there are only two good reasons for using preemptive scheduling: (1) to exploit true concurrency, and (2) to guarantee fairness or timeliness in an environment where uncooperative threads share the CPU. Scout is designed to be a uniprocessor OS, so true concurrency is not an issue. Also, at present, Scout focuses on application domains where all threads are cooperative, so (2) is not a concern either. In the future, Scout will allow for uncooperative "threads," but since it is not a good idea to share *any* resource with uncooperative threads in an uncontrolled manner, those threads will not share memory either. That is, uncooperative threads will be isolated from each other in some manner (e.g., through separate address spaces, fault isolation, or a safe language). If uncooperative threads do not share memory, using a preemptive scheduler among them is trivial. Thus, scheduling is split into domains—within a domain, there is trust and hence a non-preemptive scheduler can be used. Across domains, there is no trust and a preemptive scheduler is necessary. In a sense, this is not unlike what many traditional UNIX kernels do—the kernel "threads" are scheduled non-preemptively whereas the user-level processes are scheduled preemptively.

Once a thread executes on behalf of a path, it can trivially adjust its own priority as necessary. However, there also needs to be a mechanism that allows inheriting a path's scheduling requirements to a newly awakened thread. For this purpose, a path can register a callback that is invoked whenever a thread is awakened to execute on behalf of that path. During the callback, the path can adjust the thread's scheduling policy and priority according to its own needs.

3.4 Networking and Paths

Networking poses a special challenge to using paths since when a packet arrives at a network adapter, it is not immediately known which path that packet belongs to. Ideally, each network message would carry a path id of known length and at a known position. The path could then be determined by mapping this unique id into a pointer to the path structure. Since Scout is designed to interoperate with existing networking protocol architectures, it is usually not possible to achieve this ideal case. Instead a *packet classifier* maps arbitrary packets into the paths to which they belong. Scout places the following requirements on such a classifier:

- Efficient enough to handle peak-loads. The maximum arrival rate is dictated by the number of network adapters in the system and by the particular network technology in use. We expect a classification latency of approximately 5μs to be sufficient for network technology on the immediate horizon.

- Provide relaxed (best-effort) classification accuracy. Traditionally, packet classifiers have to deal with fragmented messages, since if a packet cannot be classified all the way to the end user, it will be dropped completely. In contrast, Scout can tolerate best-effort classification. For example, if an IP datagram is fragmented, the fragments are simply delivered to a path that leads up to the IP protocol. IP reassembles those fragments and then runs its own classifier on the complete datagram. Not only is this simpler, but it is the *only* way to guarantee that arbitrarily fragmented messages can be classified eventually.

- Classification specifications must be modular. In Scout, each networking protocol is implemented as a separate router. Since the router graph is configured relatively late in the process of building a Scout kernel, it would be awkward if the classification specifications would have to be programmed for a particular router graph. It is much more sensible to provide a partial specification when programming a networking protocol. If these partial specifications are modular, it is easy to combine them into a global specification, either at configuration time or at run time.

- Classification must be side-effect free. In the desirable case where a path id can be transmitted as part of some protocol stack, it is not necessary to continue running the classifier once that path id has been discovered. A side-effect free classification process guarantees that once the path id is found, the remaining classification can be safely short-circuited.

- The language for classification specifications must be expressive and convenient to use. Variable length and extension headers in particular need to be accommodated since they will become increasingly common with the adoption of IPv6 [PD96].

Many packet classifiers have been proposed (e.g., [YBMM93, MJ93, BGP$^+$94]), but none of them address all of Scout's requirements satisfactorily. The solution was surprisingly simple—rather than inventing a new language and implementing a good compiler for it, Scout mandates that each router that implements a networking protocol provide a callback function that is used to classify a packet. Such callbacks are executed at interrupt priority and must be side-effect free. Given a packet, it returns the path to which the packet belongs, or a NULL pointer if no such path exists. If a router cannot uniquely identify what path a packet belongs to, it will ask the next higher-level router to further narrow down the set of possible paths. That is, the classification process is modular. Furthermore, since the partial classification specifications are written in C, performance can be expected to be good. Indeed, the first unoptimized implementation of this scheme is already able to demultiplex a UDP packet in less than 5μs on a 300MHz Alpha workstation. Since regular C is used to express classification specifications, arbitrarily complex header schemes can be accommodated. But what is even more important is that since the same language is used for programming the classifier and the router, the same declarations for network headers can be used. This greatly reduces the risk of introducing errors due to inconsistencies.

3.5 Other Issues

There are many other aspects of Scout that space does not permit us to describe; most of them are orthogonal to paths. For example, software-based fault isolation [WLAG93] can be imposed on top of paths by defining each router to be in a separate fault domain. Similarly, hardware-enforced protection can be imposed between paths. Note that the horizontal partitioning (SFI) is possible because Scout routers have well-defined interfaces, while the vertical partitioning (hardware protection) is enabled by explicit paths.

Also, the Scout router graph is configured at build time, and as currently defined, it is not possible to extend the graph at runtime. However, it is possible to configure an interpreter into the router graph, thereby supporting extensibility. For example, we are currently implementing the Java API (and interpreter) in Scout [GYT96]. This will make it possible to download Java applications into Scout at runtime.

4 Demonstration Application

This section demonstrates the use and benefits of paths with a simple, but realistic application implemented in Scout. The application consists of receiving, decoding, and displaying MPEG encoded video streams. MPEG encoding is able to reduce the size of a video by a factor of 10 to 100, but this compression ratio comes with a computationally expensive decompression algorithm. Workstations have only recently become fast enough to perform this task in realtime. Since MPEG decoding involves substantial computation, it is an application that demonstrates some of the advantages of paths related to resource management.

4.1 MPEG Router Graph

The Scout router graph for the demonstration application is shown in Figure 6. The topmost router, DISPLAY, manages the framebuffer. The bottom of the graph is formed by three routers implementing standard networking protocols: UDP, IP, and ETH (a protocol providing access an Ethernet device). In the middle are the three interesting routers: MPEG, MFLOW, and SHELL.

The MPEG router accepts messages from MFLOW, applies the MPEG decompression algorithm to them, and sends the decoded images to the DISPLAY router. There, the images are queued for display at the appropriate time. The MPEG router uses application-level framing (ALF) [CT90] to avoid internal buffering. That is, the MPEG source sends Ethernet MTU-sized packets that contain an integral number of work-units (MPEG macroblocks). This ensures that the MPEG decoder does not have to maintain complex state across packet boundaries and obviates the need for undesirable queueing between MPEG and MFLOW.

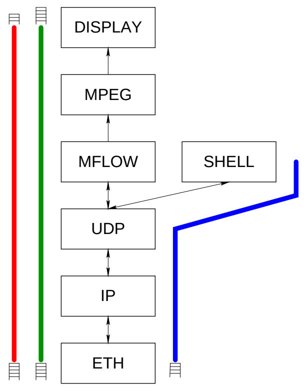

Figure 6: Router graph for MPEG example.

The MFLOW router implements a simple flow-control protocol. MFLOW advertises the maximum sequence number that it is willing to receive based on the sequence number of the last processed packet and the input queue size. MFLOW uses sequence numbers to ensure ordered, but not reliable, delivery of packets to MPEG.

The SHELL router is used to create paths. It waits for commands to arrive on its input service. Since SHELL is configured above UDP in this particular router graph, this means that such commands originate from the network, as opposed to a keyboard. In response to an `mpeg_decode` command, the SHELL router invokes *pathCreate* on the DISPLAY router with the following set of attributes:

PA_PATHNAME: The pathname attribute is used to force routing decisions. For example, DISPLAY may have other routers connected to it. By appropriately setting this attribute, the SHELL router tells DISPLAY to forward path creation to the MPEG router.

PA_NET_PARTICIPANTS: The SHELL router sets this attribute to the network address of the peer that requested the path creation. This attribute is used by networking protocols. For example, IP uses it to select the proper network interface.

PA_PROTID: This attribute is used between routers to communicate their unique numbers. These numbers are used in demultiplexing protocols such as UDP, IP, or ETH. This illustrates that the attribute set is not necessarily static during path creation: routers may add, modify, or delete attributes to the set as they see fit.

In other words, these are the attributes (invariants) that create MPEG video paths.

Figure 6 shows two video paths (from ETH to DISPLAY) and a control path for receiving commands (from ETH to SHELL). Notice that the video paths take their input from, and deposit their output into, a queue. These queues are serviced by interrupt handlers. In ETH, the queue is filled in response to a receive interrupt, and in DISPLAY, the queue is drained in response to the vertical synchronization impulse of the video display. Output to the display is synchronized to this impulse because there is no point in updating the display at a higher frequency.

There are three points worth emphasizing about this example. First, there are no queues other than the ones in ETH and DISPLAY. As mentioned above, this is due to the MPEG router's use of ALF. Second, ALF—along with explicit paths—enable integrated layer processing. In our example, the (optional) UDP checksum is easily integrated with the reading of the bit-stream. This is simplified by the fact that the MPEG bit-stream is read in 32-bit units. Third, without queuing in the middle of the path, scheduling is simplified—if the output queue is full already, there is little point in scheduling a thread to process a packet in the input queue. This implication would not hold in the presence of additional queues.

Table 1 gives measurements that indicate the performance a Scout MPEG kernel can achieve. The table lists the maximum decoding rate in frames per seconds for a selection of four video clips. To put these numbers in perspective, the table also gives the corresponding numbers for Linux. The numbers are comparable in the sense that both systems run on the same machine (a 300MHz 21064 Alpha), use essentially the same MPEG code, and receive the compressed video over the network.

Video	# of frames	normal rate	maximum rate Scout	maximum rate Linux	description
Flower	150	29.97	44.7	37.1	Drive along flower garden.
Neptune	1345	30.00	50.5	41.8	Turtle diving in sea.
RedsNightmare	1210	25.00	67.1	55.5	Computer animation.
Canyon	1758	30.00	245.9	183.3	Flight through canyon.

Table 1: Coarse-Grain Comparison of Scout and Linux

While the playing field was as level as we could make it, it must be understood that this is an apples and oranges comparison, since the two systems have a very different scope, level of functionality, and maturity. Still, the comparison is useful for two reasons. First, it establishes that Scout achieves realistic performance that is consistent with the machine on which it runs. Second, it illustrates that if the goal is to build a network appliance that displays MPEG encoded video, and nothing else, then a configurable system like Scout is the better choice. A simple application like this one results in a Scout kernel that is both small (roughly 7 times smaller than the Linux kernel) and fast (20-30% faster than Linux for the benchmark videos). The difference in performance is all the more impressive considering that MPEG is a compute-intensive application. What remains to be seen is whether Scout can be pushed up the complexity curve all the way to a full-featured system that provides functionality equivalent to a POSIX system.

4.2 Queues

As Figure 6 shows, two queues exist at the ends of the MPEG path. These queues are in the ETH router (the input queue) and in DISPLAY (the output queue).

The input queue is required for two reasons: (1) for high-latency networks it may be necessary to have multiple network packets in transit, and (2) because of network jitter, these multiple packets may all arrive clustered together. Since the peak arrival rate at the Ethernet is much higher than the MPEG processing rate, the queue is needed to absorb such peaks.

Whereas the input queue absorbs bursts that are limited in size, the job of the output queue is to absorb jitter at a more global level—decompression itself introduces significant jitter. Depending on the spatial and temporal complexity of a video scene, the encoded size of any particular video frame may be orders of magnitudes different from the size of the average frame in that stream. The network may also suffer from large-scale jitter, e.g., due to temporary congestion of a network link. Finally, the sender of the MPEG stream itself is likely to add jitter since the video may, for example, be read from a a disk drive. Just how big should these queues be? Obviously, they should be “just big enough,” but is it possible to put some quantitative limits on their sizes?

First, consider the input queue. Clearly, if processing a single packet takes more time than the time it takes to request a new packet from the source, then an input queue that can hold two packets is sufficient. One queue slot will be occupied when the last received packet is being processed; the second, free, slot can be advertised to the source. By the time processing of the last packet has finished, the new packet will have arrived already. In other words, if the amount of processing time needed for a packet after sending a window update is bigger than the round-trip time of the network, then a queue of size two is sufficient. The same logic applies if the round-trip time is large. However, since in that case the work per packet is too small to cover one network round-trip, “super-packets” consisting of n physical packets need to be used. The value of n is chosen such that n times the average processing time per packet is greater than the average round-trip latency of the network. The input queue then needs to be able hold two super-packets, or $2n$ physical packets. That is, given the average round-trip latency of the network and the average rate at which packets are consumed, it is easy to calculate an upper bound for the useful size of this queue.

MFLOW could measure the round-trip latency by putting a timestamp in its header. The important point from the perspective of this paper, however, is that accurate measurement of the peak processing rate is enabled by paths—it is a simple matter of specifying the appropriate transformation rule to ensure that the average time spent processing each packet is measured. For MPEG, this means that the initial function in the ETH-stage of the router is modified to measure processing time and to update the path attribute that keeps track of the average processing time.

In the case of the output queue, the factors influencing queue size are more varied and complex. A complete analysis is beyond the scope of this paper. In general, bounding the size of this queue requires cooperation with admission control and would typically employ a network reservation system, such as RSVP [BCS94]. The current implementation leaves this parameter under user control to facilitate experimentation.

4.3 Scheduling

Since each video path has its own input queue and since the packet classifier is run at interrupt time, newly arriving packets are immediately placed in the correct queue. This means that there is no danger of priority inversion due to low-priority packets appearing ahead of high-priority packets. This is one of the most significant advantages of path, but as the following discussion illustrates, paths play an even more intimate role in scheduling.

As explained in Section 3, a path can register a wakeup callback that can be used to adjust a thread's scheduling policy and priority according to its own needs. The MPEG path uses this facility to ensure that any thread that is ready for execution in the path will be scheduled with the proper realtime constraints. In combination, separate input queues and proper scheduling guarantee that the MPEG Scout kernel has no difficulty in delivering and processing realtime MPEG packets even under severe background loads. For example, an arbitrary number of low-priority MPEG streams (or some other non-realtime background work) can be displayed without affecting realtime streams running in the foreground. It is even possible to run realtime video streams while flooding the network adapter with minimum sized Ethernet packets.

The default Scout scheduler is a fixed-priority, round-robin scheduler. Since video is periodic, it seems reasonable to use rate-monotonic (RM) scheduling for MPEG paths. With RM scheduling, a (periodic) realtime thread receives a priority level that is proportional to the rate at which it executes. That is, the frame-rate at which a video is displayed would control the priority of the corresponding path. However, there are several reasons that make earliest-deadline-first (EDF) scheduling more attractive than RM scheduling. These include:

- The frame-rate must be under user control to support features such as slow-motion play or fast forward. This implies that a large number of priority-levels would be necessary. Otherwise, two MPEG paths that have similar, but not identical, frame-rates could not be distinguished scheduling-wise. If the number of priority levels is large, EDF scheduling is just as efficient as RM scheduling.

- MPEG decoding is periodic, but not perfectly so. Consider playing a movie at 31Hz on a machine with a display update frequency of 30Hz. Given that only 30 images can be displayed every second, it will be necessary to drop one image during each one second interval. When the drop occurs, there is no need to schedule that path, so a fixed priority would be suboptimal.

- While not a quantitative argument, probably the strongest case for EDF scheduling is that it is the *natural* choice for a soft realtime thread that moves data from an input queue to an output queue. For example, if the output queue drains at 30 frames/second and the queue is half full, it is trivial to compute the deadline by which the next frame has to be produced.

For these reasons, the Scout MPEG decoder uses EDF scheduling for realtime MPEG paths. EDF scheduling achieves significant benefits. For example, it allows Scout to display 8 Canyon movies at a rate of 10 frames per second, together with a Neptune movie playing at 30 frames per second, all without missing a single deadline. In contrast, the same load with single-priority round-robin scheduling leads to a massive number of missed deadlines if the output queues for the Canyon movies are large. For example, with a queue size of 128 frames, on the order of 850 out of 1345 deadlines are missed by the path displaying the Neptune movie.

One question that remains is how the deadline is computed. Here again, paths play a central role. If path execution is the bottleneck, then the output queue should be kept as full as possible. In this case, it is best to set the deadline to the display time of the next frame to be put in the output queue. In contrast, if network latency is the bottleneck, then the deadline should be based on the state of the input queue. Since at any given time n packets should be in the network pipe, the deadline is the time at which the input queue will have filled up so that less then n slots remain free. This can be estimated based on the current length of the queue and the average packet arrival rate.

Since the path object provides direct access to either queue, the effective deadline can simply be computed as the minimum of the deadlines associated with each queue. Alternatively, the path can use the path execution time and network round-trip time to decide which queue is the bottleneck queue, and then schedule according to the bottleneck queue only. The latter approach is a slightly more efficient but requires a clear separation between path execution time and network round-trip time. The implemented MPEG decoder is currently optimized for the case where the output queue is the bottleneck, so scheduling is always driven off of that queue.

4.4 Admission Control and Resource Accounting

Finally, paths enable admission control. As all memory allocation requests are performed on behalf of a given path, it is a simple matter of accounting to decide whether a newly created path is admissible or not. Before starting path creation, the admission policy decides how much memory can be granted to a new path. As long as each router in the path lives within that constraint, the path creation process is allowed to continue. (Note that admission control has not yet been implemented in Scout.)

Paths are also useful in deciding admissibility with respect to CPU load. Again, it is the fact that it is easy to compute the execution time spent per path that is helpful—our experiments show that there is a good correlation between the average size of a frame (in bits) and the average amount of CPU time it takes to decode a frame. Naturally, the model that translates average frame size into CPU processing time is parameterized by the speed of the CPU, the memory system, and the graphics card. Rather than determining these parameters manually, it is much easier to measure path execution time in the running system and use those measurements to derive the required parameters. That is, the path execution timings are used to derive the model parameters, which in turn, are used for admission control.

Finally, if admission control determines that a video cannot be displayed at the full rate, a user may choose to view the video with reduced quality. For example, the user may request that only every third image be displayed. Thanks to ALF and paths, it is possible to drop packets of skipped frames as soon as they arrive at the network adapter. This avoids wasting CPU cycles at a time when they are at a premium.

5 Concluding Remarks

This paper makes two contributions. First, it gives a concrete example of how paths can be made an explicit OS abstraction. Specifically, we described how paths are implemented in Scout, a configurable OS kernel that can be specialized to support particular I/O-intensive applications. Second, it makes a case for why paths should be made explicit. This case includes: (1) pointers to related work that presume the existence of paths, (2) the intuitive arguments made in Section 2, and most importantly, (3) specific examples of how paths proved beneficial in one particular application—receiving, decoding, and displaying MPEG-compressed video. On this third point, we showed how paths are used to:

- segregate work early, so as to avoid priority inversion;
- schedule the *entire* processing along a path according to the bottleneck queue, and to automatically determine the bottleneck queue in the system;
- provide accountability to decide the admissibility of a memory allocation request; and
- discard unnecessary work early to minimize the waste of resources.

What remains to be done is to demonstrate Scout—and the utility of paths—on a wider set of domains. For example, work on a Scout-based Java-box and scalable storage server are under way.

Acknowledgments

We would like to thank the other members of the Scout group, particularly, John Hartman, Brady Montz, David Larson, and Rob Piltz.

References

[AP93] Mark B. Abbott and Larry L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5), October 1993.

[BCS94] Robert Braden, David Clark, and Scott Shenker. RFC-1633: Internet architecture: An overview. Available via ftp from ftp.nisc.sri.com, July 1994.

[BGP$^+$94] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. PathFinder: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, 1994.

[CT90] David Clark and David Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of SIGCOMM '90 Symposium*, September 1990.

[DP93] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, December 1993.

[FL94] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proc. of the Winter 1994 USENIX Conference*, pages 97–114, January 1994.

[GYT96] James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface*. Addison-Wesley, Reading, MA, 1996.

[HFC76] A.N. Habermann, Lawrence Flon, and Lee Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, May 1976.

[HK93] Graham Hamilton and Panos Kougiouris. The Spring nucleus: a microkernel for objects. In *Proc. of the Summer 1993 USENIX Conference*, pages 147–159, Cincinnati, OH, June 1993.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[HP94] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[LL73] Liu and Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

[MJ93] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *1993 Winter USENIX Conference*, San Diego, CA, January 1993. USENIX.

[MMO$^+$94] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical Report 94/20, University of Arizona, Tucson, AZ 85721, June 1994.

[MPBO96] David Mosberger, Larry Peterson, Patrick Bridges, and Sean O'Malley. Analysis of techniques to improve protocol latency. In *Proceedings of SIGCOMM '96 Symposium*, September 1996.

[Ous96] John K. Ousterhout. Why threads are a bad idea (for most purposes). In *1996 Winder USENIX Conference*, 1996. Invited talk.

[PAB$^+$95] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 314–324. Association for Computing Machinery SIGOPS, December 1995.

[PD96] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[RBF$^+$95] Robbert Van Renesse, Ken Birman, Roy Friedman, Mark Hayden, and David Karr. A framework for protocol composition in horus. In *Proc. of the Fourteenth ACM Symp. on Principles of Distributed Computing*, pages 80–89, August 1995.

[Rit84] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, October 1984.

[WLAG93] Robert Wahbe, Steven Lucco, Tom Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 203–216. Association for Computing Machinery SIGOPS, December 1993.

[YBMM93] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. July 1993.

[Zim80] Hubert Zimmermann. OSI reference model—the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, COM-28(4):425–432, April 1980.