Making Paths Explicit in the Scout Operating System

David Mosberger and Larry L. Peterson
Department of Computer Science
University of Arizona
`http://www.cs.arizona.edu/scout/`

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.

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 [12, 32], to more recent advances in stackable systems [27, 15, 14, 26], 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.

We begin by developing some intuition about paths. A path can be viewed as a logical channel through a multi-layered system over which I/O data flows, as illustrated in Figure 1. In this way, a path is analogous to a virtual circuit that cuts through the nodes of a packet-switched network. The only difference is that paths are within a single host, while virtual circuits run between hosts.[1]

Also, the term "path" is well entrenched in our vocabulary. For example, we often refer to the "fast 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.

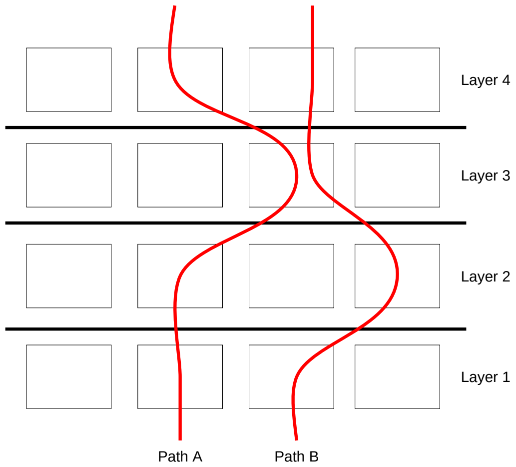

Figure 1: Two Paths Through a Layered System

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

- Fbufs [6] are a path-oriented buffer management mechanism designed to efficiently move data across a sequence of protection domains.[2] Fbufs depend on being able to identify the path through the system over which the data will flow.

[1] The obvious next step is to integrate paths through the end host with circuits between hosts, but for the purpose of this paper, we focus on paths *within* a single system.

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

- Integrated layer processing (ILP) [4, 1] 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 [31, 20, 2, 8] 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 [24, 23]. Specialization, in turn, depends on the existence of invariants that constrain the path through the code that is likely to be executed.

- The Alpha OS allows threads to migrate across a sequence of protection domains [5]; others have defined similar mechanisms [13, 9]. Such mechanisms recognize that tasks often span multiple domains, and so account for resource usage on a path basis rather than a domain basis.

The thesis of this paper is that these mechanisms are not isolated optimizations, but rather, that they can be unified and explained by the path abstraction. In a nutshell, these mechanisms all share the following fundamental idea: they expose and exploit non-local context.

Consider a layered system like the one illustrated in Figure 1. While the advantage of layering and modularity is to hide information, there are many situations when it would be beneficial for a given layer to have access to non-local context. For example, suppose one of the modules is processing an Ethernet packet. With only local context, the module knows nothing about the packet's relative importance compared to other packets. However, if it is known that the packet is part of a particular video stream, then it is easy to determine its processing deadline, what modules need to be executed to process it, how many CPU cycles this processing will require, where its data should be placed in memory, and so on. In other words, by knowing a certain set of invariants (e.g., that the packet is part of some video stream), the module is able to access and exploit global context that is available outside any one module or layer. Abstractly then, a path is defined by these invariants and provides access to the corresponding context.

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, facilitating the following benefits:

- The system can place data in a memory buffer that is already accessible to all the modules 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 module or layer boundary.

- 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, not segregating work into paths 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 often has to process several layers before knowing that continuing is of no value.

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 makes two contributions. First, it develops an explicit path abstraction; Section 2 explores the design space for paths, and Section 3 describes an implementation of paths in the Scout operating system. Second, the paper demonstrates how having a path abstraction leads to the first set of advantages outlined above, i.e., those that have to do with improvements in resource allocation and scheduling. In particular, Section 4 describes an application that receives MPEG-compressed video over a network and then decodes and displays it. A companion paper demonstrates some of the code-related improvements attributable to paths [23].

2 Path Abstraction

While it is tempting to view paths as an optimization that can be super-imposed on an existing layered system—and it is certainly the case that many of the ideas described in this paper can be applied in this way—we take a more "first principles" approach to defining paths. Specifically, this section develops a working definition of paths in an incremental fashion. Our goal is to explore the design space for paths, and in the process, to introduce the particular architecture that we settled on.

2.1 Basic Paths

A path is a linear flow of data that starts at a source device and ends at a destination device. While the data is moved from the source to the destination, it is transformed (processed) in some path-specific manner. That is, if the input data is represented as a message m, then the output message is $g(m)$, where g is the function that represents the path-specific transformation. In addition, each path has two queues associated with it, as depicted in Figure 2, thereby decoupling the input and output devices. A path scheduler determines when a given path is executed, that is, when $g(m)$ is evaluated.

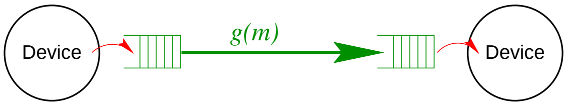

Figure 2: Simple path

While paths connect devices, there is no direct relationship between device pairs and paths: a given device pair can be connected by zero or more paths. Allowing for multiple paths to connect the same pair of devices is sensible since both $g(m)$ and the scheduling priority may vary. For example, one path may handle UDP packets, whereas another may handle TCP packets. Similarly, two paths that forward IP packets between a pair of devices may need to be scheduled differently if they provide a different quality of service. Thus, paths are dynamic entities that are created at runtime; there is no a priori limit on the number of paths that can exist in a given system.

What are the properties of basic paths? First and foremost, once a message has been enqueued on the input queue of a path, it is already known what device the (possibly transformed) output message will arrive at. For the purpose of resource management, it is also known that m belongs to the path on which it is enqueued, and all execution is performed on behalf of that path. In other words, knowledge is available *early* and *globally*.

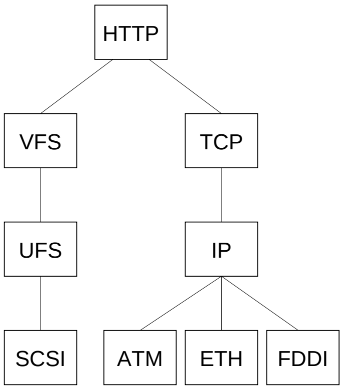

Figure 3: Example Router Graph

2.2 Creating Paths

The key problem in creating a path is how to specify $g(m)$. While it would be possible to write a specific function for each path with distinct functionality, it is more convenient to derive $g(m)$ automatically from a modularly organized system. This is because many of the paths are likely to have substantial functionality in common, such as various network, file, and windowing protocols. Generating a path from components does not preclude writing a specialized $g(m)$ for those cases that warrant the extra effort.

A *router graph* represents the modular structure of the system, where each router is a software module that implements a specific task, such as the NFS protocol or a SCSI driver. The reason we call these modules routers will become clear in a moment. As is common in a modular/layered system, individual routers provide their functionality based on the functionality of other routers, that is, it is possible to draw a dependency graph that represents the interdependence among the various routers. This also means that a router graph fully describes what kind of tasks a given system can perform. For example, Figure 3 depicts a router graph that could be used to implement a web server. Given this graph, a path that starts at the ETH router (Ethernet device) and ends at the SCSI router (disk device) would have a $g(m)$ that is the composite of the functions contributed by each router; i.e.,

$$g(m) = g_{\mathrm{SCSI}}(g_{\mathrm{UFS}}(\cdots(g_{\mathrm{IP}}(g_{\mathrm{ETH}}(m)))\cdots)).$$

However, this still begs the question of how a path is created. There are essentially two approaches: (1) the path is pre-specified externally, and (2) the path is discovered incrementally. This division corresponds to the two sources of "knowledge" that influence path creation: global and router-specific. Global knowledge is of the sort "for a web service, the following sequence of routers need to be part of the path." Global knowledge may also be used for optimization, for example, there may be an optimized $g(m)$ available for the web path that is preferred to an automatically derived composite function. In contrast, local knowledge is of the form "if invariant X is true for the path under construction, then the path can pass through this router" or "if invariant Y is false, then the path cannot go beyond this router."

Using global knowledge alone to create a path would be difficult since this knowledge often requires familiarity with the internal workings of the routers that are traversed by a path. In contrast, creating a path based on router-specific knowledge alone would limit the utility of paths considerably. (Recall that most advantages of paths are due to the global knowledge they afford).

In our path architecture, paths are created in two phases. First, router-specific knowledge is used to create a maximum length path. Second, this maximum length path is transformed (optimized) using global transformation rules, each of which is defined by a $\langle \textit{guard}, \textit{transformation} \rangle$ pair. If the guard evaluates to TRUE, the corresponding transformation is applied, resulting in a new path. This process repeats until all guards evaluate to FALSE.

To better illustrate the difference between local and global knowledge, consider the router graph given in Figure 3. Suppose there is a path that starts at SCSI and ends at ETH. Such is the case, for example, if IP can determine that the remote host is on the same Ethernet as the local host. If this is not true, then IP can not be sure whether data will go out through ATM or FDDI, since the routing tables may change in the middle of the data transfer. Clearly, this decision is completely IP specific, that is, based on local knowledge. On the other hand, there are several global facts that hold for a web path that could be exploited, for example, data is transfered through TCP in a predominantly uni-directional manner and accessed on the disk in a strictly sequential fashion. Note that each such invariant may affect the function g_i of one or more other routers. For example, the fact that data is accessed sequentially may mean that it is best to avoid caching in the file system (UFS). Similarly, the fact that TCP is in the path may mean that the IP fragmentation code can be omitted completely.

Finally, note that our definition of a path's semantics, which we denote as $g(m)$, should not be taken to mean that procedures constitute the fundamental building blocks of paths. It is equally legitimate to construct a path from a sequence of basic blocks, which is more in line with having the path abstraction represent the "fast path" through the system. In fact, procedures and basic blocks define two ends of a granularity spectrum. Scout implements a specific point on this spectrum, as described in Section 3.

2.3 Network View of Paths

We motivated routers as a means to automatically derive the path function $g(m)$. Alternatively, a router graph can be viewed as a set of interconnected nodes that forward messages along their links in the dependency graph. The operation of a router is to receive a message, process it, and then forward the resulting message to another router, as illustrated in Figure 4.

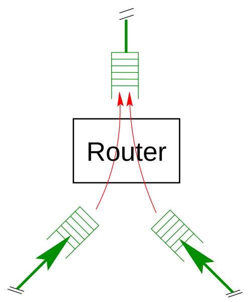

Figure 4: Message forwarding/routing

If a message is injected at some router, the trail it leaves in the router graph as it is processed and forwarded corresponds to a sequence of routing decisions. If a given trail is used very often, it may be worthwhile to explicitly encapsulate and optimize it. This is exactly what a path does: it represents a fixed sequence of routing decisions through the system's modules. This is not unlike a virtual circuit through a network: at connection establishment time, a set of invariants that are guaranteed to hold for the duration of a connection is specified. In return, these invariants permit the customization of the path in a way that is optimized for that particular connection. In the case of a virtual circuit, the set of invariants contains the address pair of the communicating peers, but it may include other parameters such as the desired quality of service. The same kind of invariants are useful for creating paths. For example, knowing what quality-of-service a path requires helps when choosing an appropriate scheduling policy and priority.

In this context, it makes more sense to let paths connect an arbitrary pair of routers rather than insisting that a path connects a pair of devices. The latter case is ideal in the sense that it provides a maximum amount of global knowledge. However, the maximum length of a path is related to the strength of the invariants. In general, the stronger the invariants, the more routing decisions can be frozen at path-creation time, and the longer the resulting path. While it is preferable to have long paths, a general model must allow for the degenerate case where invariants are so weak that not a single routing decision can be made at path creation time. This degenerate case roughly corresponds to a traditional layered system.

2.4 Generalized Paths

As defined so far, paths are simple and highly predictable: a message arrives at the input queue, the path is scheduled for execution, and the transformed message is deposited in the output queue. While this simplicity is ideal for the purpose of optimization, it also limits the usefulness of paths. Since it is our goal to define paths in a way that moves them from a purely performance-motivated concept into an abstraction with which a complete operating system can be built, we must extend paths to make them more widely applicable, but in a way that does not destroy the properties that make the path abstraction attractive in the first place.

2.4.1 Directionality

Processing in a path is usually bi-directional: a remote-procedure call arrives over the network, results in some computation, and an answer is sent back to the caller; the arrival of a network packet triggers the sending of an acknowledgment; or a disk block is requested and arrives asynchronously. Such bi-directional paths could be handled by creating two separate paths, but it seems more natural if a path that is used to make a request is also the one that yields the response. A similar argument can be made about resource management. A more technical argument for making paths bi-directional is that often the two directions are dependent on each other. For example, when sending a network packet to a remote host, it may be desirable to include a piggy-back acknowledgment in that same packet.

Therefore, we extend the path model as follows. Each path end has a *pair* of queues—an input queue for one direction, an output queue for the other direction. The path function $g(m)$ is also extended to take a second argument d that gives the direction (FWD or BWD) in which the path should be traversed. FWD is the direction in which the path was created, while BWD refers to the reverse direction. Each router-specific function is extended in the same way.

2.4.2 Complex Processing

The current path model assumes that the path transformation is "work-preserving," that is, for every input message, there is exactly one output message. This is limiting since it means that important operations such as packet reassembly and fragmentation cannot be accommodated. In the former case, most input messages do not result in an output message. Instead, the partial messages are buffered inside the router. In the latter case, every input message may result in many more than one output message. Similarly, a retransmission timeout may result in a new message being generated spontaneously from within the path.

For this reason, we loosen the evaluation rule for paths. Suppose that creating a path results in the routers contributing the functions g_1, ..., g_n. A message may now be injected at any one of these sub-functions and the invocation of g_i may result either in g_{i-1} or in g_{i+1} being invoked. That is, these sub-functions can be invoked in any order, subject to the rule that only neighboring functions are invoked, or that the message be enqueued at an output queue.

2.5 Remarks

In summary, a path is created incrementally by invoking a create operation on a router and specifying a set of invariants. The invariants describe the properties of the desired path, and are used to determine a next router that must be traversed by any message traveling on this path. The path reaches its maximum length when the invariants are no longer strong enough to make a unique routing decision. Each traversed router contributes a function g_i that is applied when processing a message. A path that traverses three routers is shown in Figure 5.

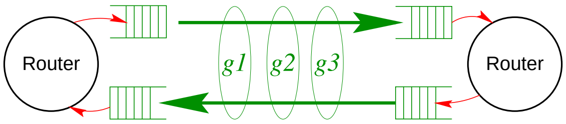

Figure 5: Example Path

Path execution is decoupled from the arrival and departure processes at the routers by four queues. For each direction, there is an input and an output queue. Typically, a path execution involves dequeuing a message from an input queue and evaluating the g_i functions in sequence until the other end of the path is reached. However, for generality, a message may get absorbed in the middle of a path, or turned around, or a new message may be created spontaneously inside a path.

Finally, keep in mind that policy issues—i.e., how to use paths for a given system—remain unspecified. There are two dimensions to this issue, which can be visualized as the "length" and the "width" of the path, respectively. The "length" of the path is simple to understand: it corresponds to the number of routers that the path traverses. A path's width is more subtle: a highly specialized path is narrow, whereas a more general path is wide. For example, a path that can only be used to carry non-fragmented messages for a specific host-pair would be considered narrow, while a path from a network adapter to the IP protocol that can handle any IP datagram would be considered wide.

While it might seem that one wants paths to be as narrow (specific) as possible, this is not necessarily the case. Such a strategy can lead to an explosion of paths—e.g., one per packet or one per request/response transaction—which also implies having to create paths too frequently. Since there is a cost associated with path creation, one clearly wants the path to have enough breadth to carry multiple messages. The strategy we have adopted is to define a modest number of long-lived paths (e.g., one per window, one per open file, one per TCP connection) and then to define a small number of "short/fat" paths to catch the exceptional cases (e.g., all fragmented IP packets).

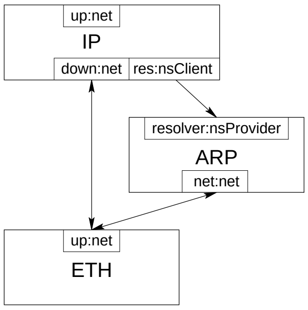

Figure 6: Routers and Services.

3 Implementing Paths in Scout

Scout is an experimental operating system designed for network appliances—e.g., set-top boxes, file- and web-servers, and cluster computers. Scout is designed around the path abstraction, supports both non-realtime and soft-realtime applications, and runs in a single address space. This section describes how the path abstraction is implemented in Scout.

Note that compatibility with standard application interfaces (e.g., POSIX) is not a major goal of Scout, except to understand how such interfaces either exploit or interfere with paths. On the other hand, interoperability with existing protocol specifications is an important requirement of Scout.

3.1 Routers and Services

Just as in the architecture, *routers* are 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 relevance of service types is explained in more detail below. For the purpose of configuring a router graph, it is sufficient to know that two services can be connected by an edge only if they are mutually compatible. Figure 6 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 "name-service client"). The *down* service is connected to ETH's *up* service. This connection is used by IP to send and receive IP datagrams. The *res* service is connected to ARP's *resolver* service. IP uses this to translate IP host numbers into Ethernet addresses. ARP itself is connected to ETH as well so that it can broadcast and listen to relevant ARP packets.

A router is implemented simply as a collection of C source files. These files, along with the external interface, are described in a *spec* file. The syntax for spec files is shown below:

```
router name {
    files = {filename, ...};
    service = {name:type, ...};
}
```

A service name may be preceded by a less-than marker (<) to indicate that the routers connected to that service must be initialized before this router can be initialized. The Scout infrastructure ensures that router initialization occurs in an order that is consistent with the partial order defined by these markers. For the purpose of router initialization, cyclic dependencies are forbidden. The Scout development environment includes a configuration tool that translates a router graph into C source code that creates and initializes the runtime view of a router graph when the system boots. This configuration tool checks for and rejects any router graph with cyclic dependencies.

At runtime, a Scout router is represented by a variable of the following structure:

```
struct Router {
    String           name;
    long             (*init)(Router r);
    CreateStageFunc  createStage;
    DemuxFunc        demux;
    RouterLinkList   links[NSERVICES];
};
```

That is, a router consists of its name (member *name*), three function pointers (members *init*, *createStage*, and *demux*) and a list of router graph edges that connect to this router to other routers (member *links*).

Each router r provides just one globally visible operation:

```
Router rCreate (String n, int c[]);
```

This operation is used to create a specific router with name n. The integer array c specifies how many times each router service has been connected to other routers. Once all routers are created, Scout initializes them in the partial order described above. A router is initialized by a call to its *init* function.

3.2 Path Object

Section 2 argued that it is preferable to create paths incrementally, with the resulting paths initially consisting of a sequence of sub-functions g_i. Likewise, Scout paths consist of a sequence of *stages*. Each router that is crossed by a path creates one such stage. Since a path enters a router at one service and leaves it through another, a stage effectively connects a pair of services. That is, it represents a fixed routing decision.

A stage is a rich object that contains at least the following members:

```
struct Stage {
    Iface  end[2];
    Path   path;
    Router router;
    long   (*establish)(Stage s, Attrs a);
    void   (*destroy)(Stage s);
};
```

Member *end* is an array containing pointers to the interfaces of the stage. These interfaces are derived from the services that a stage connects in a manner that will be explained below. The *path* and *router* members point to the path that the stage is part of and the router that created the stage, respectively. The *establish* and *destroy* function pointers are used during path creation and destruction and are explained in more detail in Section 3.3.

The relationship between interfaces and router services is as follows. Each router service type consists of a pair of interface types: the first element in this pair specifies what interface the service *provides* whereas the second element specifies the interface that the service *requires* to function properly. For example, the *net* service type is symmetric in the sense that it both provides and requires a net interface. This can be expressed as the pair:

```
servicetype net = <NetIface, NetIface>;
```

Scout supports simple single inheritance for interface types. This means that instead of the exact interface type required by a service it is possible to provide a more specific interface. Hence, the precise rule used to decide whether a pair of services can be connected in a router graph is that the interfaces provided must be identical to or more specific than the interfaces required.

All interfaces encountered when traversing a path in a particular direction are chained together. Since it is sometimes necessary to "turn around" the data flow inside a path, each interface also contains a back pointer to the next interface in the *opposite* direction. A third pointer provides access to the stage to which the interface belongs. Therefore, the most primitive interface is given by:

```
struct Iface {
    Iface next;
    Iface back;
    Stage stage;
};
```

This obviously is not a very interesting interface since it provides no way to deliver data. All real interfaces declare additional members that hold function pointers or other data. For example, the *net* interface is declared as follows:

```
struct NetIface {
    struct Iface i;
    long (*deliver)(Iface i, Msg m);
};
```

That is, the *net* interface provides a single function to deliver a message m to interface i. While Scout can technically support an arbitrary number of interface types, the intent is to keep this number as small as possible. For example, at present there is an interface type to asynchronously exchange messages (this is used both by filters and networking protocols), a window manager interface, a file system interface, and a few other, lesser interface types.

Given the definition of stages and interfaces, it is now easy to describe the actual path object:

```
struct Path {
    Stage     end[2];
    long      pid;
    void      (*wakeup)(Path p, Thread t);
    PathQueue q[4];
    struct Attrs attrs;
};
```

The array *end* contains two pointers to the stages at the extreme ends of the path. A path can set the *wakeup* function pointer to request that a specific function gets executed when a thread t is awakened to execute in a path p. This is discussed more in Section 3.4. The four path queues are stored in q. These queues are generic in the sense that the queuing discipline is unspecified. The two properties that are defined for any such queue is the current length and the maximum length. Finally, *attrs* is a set of name/value pairs (attributes). Attributes allow to attach arbitrary state to a particular path. For example, this enables stages to exchange and share information anonymously (without knowing exactly what stage is the source of the information and what stages are the consumers).

A path can therefore be visualized as shown in Figure 7. The path shown there consists of four stages. The stages were created by the TEST, UDP, IP, 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 extreme stages are, strictly speaking, not part of the path but they are used to connect to the routers that manage the path queues.

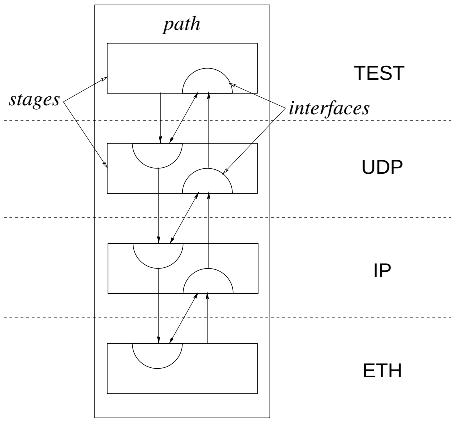

Figure 7: Path structure.

3.3 Path Creation

Paths are created and destroyed using the following functions:

```
Path pathCreate(Router r, Attrs a);
void pathDelete(Path p);
```

A path is created by invoking *pathCreate* on a router *r*. The kind of path to be created is described by the set of attributes *a*. These attributes are arbitrary name/value pairs that specify the invariants that hold true for the path being created. The *pathCreate* results in an invocation of the *createStage* function in router *r* (see Section 3.1). The *createStage* function has the following type:

```
Stage (*CreateStageFunc)(Router r, int s,
                         Attrs a,
                         RouterLink* n);
```

Here, *r* is the router on which *pathCreate* was invoked and *s* is the number of the service through which the path being created enters the router. Since *r* is the first router in the path, there is no such service, so the value is set to -1 (not a valid service number). Argument *a* is the set of attributes passed to the *pathCreate* operation. Once *r* creates a new stage and makes a routing decision, it sets *n* to the router/service pair that the path must traverse next, if there is such a pair, otherwise it sets it to NULL.

Given the next router/service pair, the *createStage* operation is invoked on that next router. Now, argument *s* is set to the index of service through which the path enters the router and *a* is the (possibly modified) set of attributes. This process continues until a path reaches its full length, which happens either when it reaches a leaf router or when the attributes are so weak that no unique routing decision is possible. When either event occurs, a sequence of stages has been created. The stages and the interfaces contained therein are then linked together into a path structure. Once the path object is fully created, the *establish* functions in the stage objects are executed in the order in which the stages were created. This gives each stage a chance to perform initialization that depends on the existence of the entire path.

As described so far, path creation consists of three phases: (1) create sequence of stages, (2) combine stages into path object, and (3) establish (initialize) stages. During a fourth and final phase, path transformation rules are applied to the path. This provides the means through which Scout uses global knowledge to transform and optimize paths. Semantically, transformation rules have no effect, but they typically result in better performance and better resource allocation or usage. For example, if a path contains a sequence of interfaces for which there is optimized code is available, then the function pointers in the interfaces can be updated to point to this optimized code. More details on such code-related path-transformations can be found in a companion paper [23]. Section 4 discusses some transformations that improve resource management.

When a Scout system boots, there are typically a few routers that create a handful of paths, e.g., to receive key strokes or network packets. All other paths are either directly or indirectly created by these initial paths. In other words, path creation and destruction is under control of the routers that are present in a given system. The Scout infrastructure never creates or destroys paths implicitly.

3.4 Path Execution

Paths are executed by threads—the active entities in Scout. A router starts execution of a path by dequeuing data from the input queue and invoking an interface-type dependent data-delivery function.

Since threads are independent objects and since path queues can often be optimized away, it is possible for a thread to execute a path, enter a router, and then continue execution in another path without any context switches. This is important because degenerate paths can be short, so forcing context switches at every path/router crossing could result in an excessive number of context switches, and therefore, less than optimal performance.

In Scout, threads are scheduled non-preemptively according to some 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) [18]. 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. 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. 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 a newly awakened thread to inherit a path's scheduling requirements. For this purpose, a path can set the *wakeup* function pointer in its path object to a function that selects the appropriate scheduling policy and priority for a newly awakened thread.

3.5 Finding Paths

In many cases, knowing the path that should be used for a given set of data is trivial. For example, an application might create a path to a graphics window and then use that path to draw lines and paint text. In some cases, however, the path to be used is determined implicitly by the data itself. For example, when a packet arrives at a network adapter, it is not immediately known which path that packet belongs to. For this reason, each Scout router provides a *demux* operation that maps the data into a path that can be used to process that data. This problem is identical to what is referred to a "packet classification" in the networking literature. Since Scout uses packet classification in a context that is somewhat unusual, it is worth enumerating the specific requirements that it places on this process:

- Efficient enough to handle peak-loads. Classification must take a short amount of time relative to the typical path execution time. Otherwise, the advantage of improved resource management due to paths would be lost.

- Provide relaxed (best-effort) classification accuracy. Unlike traditional classifiers, the Scout classifier just has to find a path that is "good enough" to process the given data. This is best illustrated with an example: suppose the data to be classified is an IP fragment. Traditional classifiers defer classification until the entire IP datagram has been reassembled. For the purposes of Scout, it is acceptable to hand off IP fragments to a path that knows how to reassemble the fragments. Once the full datagram is available, the IP protocol can rerun the classifier to find the next path.

Many packet classifiers have been proposed (e.g., [31, 20, 2, 8]), but none of them address all of the Scout requirements satisfactorily. For this reason, Scout adopted the simple solution of requiring each router to provide a function that performs a classification. Any given router typically implements only a small portion of the entire classification process. If a router cannot make a unique classification decision, it may ask the next router to refine that decision. This continues until either a unique path is found or until it is determined that no appropriate path exists. In the latter case the offending data is simply discarded.

3.6 Remarks

Figure 8 summarizes the Scout timeline. At the earliest time (top), individual routers and path transformations are implemented. Later on, a system is configured by specifying a router graph and selecting appropriate transformation rules. The kernel is then built and booted. During runtime, paths are created, executed, and eventually destroyed when no longer needed.

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. This time does not include the application of any transformations. 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). Also, packet classification is reasonably efficient. The first (unoptimized) implementation of the Scout classification scheme is already able to demultiplex a UDP packet in less than 5μs.

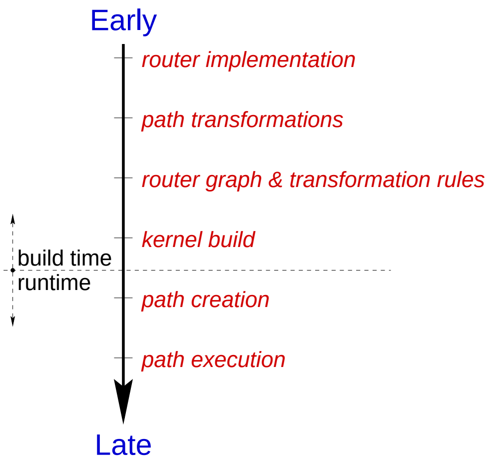

Figure 8: Scout Development Timeline.

There are many other aspects of Scout that space does not permit us to describe; most of them are orthogonal to paths. For example, we believe software-based fault isolation (SFI) [30] could be imposed on top of paths by defining each router to be in a separate fault domain. Similarly, hardware-enforced protection could 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 [10]. 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 9. 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. In the middle are the three interesting routers: MPEG, MFLOW, and SHELL.

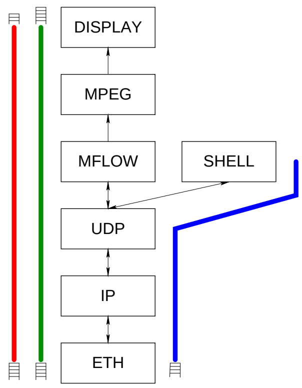

Figure 9: Router graph for MPEG example.

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) [4] to minimize 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.

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 dynamically. It is configured on top of UDP so it can receive command requests via the network. SHELL is not unlike a UNIX shell in that it waits for a command request which it then maps into a command "invocation." In the context of Scout, this involves mapping the command name into an appropriate path create operation. To create a path, SHELL requires two pieces of information: the router on which the path create operation is to be invoked on and a set of attributes (invariants). In the current implementation, an `mpeg_decode` command always results in a path create invocation on the DISPLAY router. In general, SHELL might consult an environment variable to select the graphics display to be used. SHELL creates MPEG paths with the following two attributes:

PA_NET_PARTICIPANTS=⟨*ip-addr*, *udp-port*⟩: This attribute specifies the network address of the process that sent the `mpeg_decode` command request. SHELL assumes that the network address of the video source is the same as the address that originated the command request.

PA_PATHNAME="MPEG": The value of this attribute is a string that, in its simplest form, is interpreted as a sequence of router-names. It is used either to force a specific routing decision or to supply routing information when there is no other routing information available. In the case of an MPEG path, SHELL sets this attribute to the string "MPEG" to force DISPLAY to forward path creation to the MPEG router.

Another attribute that is used during MPEG path creation is PA_PROTID. Unlike the other attributes, this one is not specified by the SHELL router. Instead, it is reset by each router that implements a networking protocol. The value of this attribute is the protocol id of the next-higher level networking protocol. This id is normally needed during packet classification. For example, IP packets with a protocol type of 6 are TCP packets and TCP packets with a port number of 21 are normally FTP packets. So when FTP forwards a path create operation to TCP, it sets PA_PROTID to 21. If TCP decides to forward path creation to IP, it resets the value of PA_PROTID to 6 to let IP know that it is dealing with a TCP path.

Figure 9 shows two video paths (from ETH to DISPLAY) and a shell path for receiving commands (from ETH to SHELL). Note 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 MPEG's use of ALF. Second, ALF—along with explicit paths—enable integrated layer processing. Since MPEG reads the network packet data in units of 32 bits, it would be straight-forward to integrate the (optional) UDP checksum with the reading of the MPEG data. This would require a path-transformation rule that matches for MPEG being run directly on top of UDP. If this pattern matches, the path can be transformed by replacing the UDP and MPEG receive processing functions with functions that implement the UDP checksum computation as part of MPEG's reading of the packet data. 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 second 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. The dominant costs in this example are the decompression of the MPEG stream and the dithering and displaying of the video frames. That is, practically all time is spent in the MPEG and DISPLAY routers.

Video	# of frames	max. rate [fps]	
		Scout	Linux
Flower	150	44.7	37.1
Neptune	1345	49.9	39.2
RedsNightmare	1210	67.1	55.5
Canyon	1758	245.9	183.3

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—the two systems have a very different scope, level of functionality, and maturity. Still, the comparison is useful to establish that a path-based system such as Scout can easily achieve performance that is consistent with the machine on which it runs.

4.2 Queues

As Figure 9 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 significant 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. If processing a single packet requires more time than it takes to request a new packet from the source, then an input queue that can hold two packets is sufficient: one slot is occupied while the last received packet is being processed, and the second (free) slot is advertised to the source. If the round-trip time (RTT) is greater than the time to process a packet, then the input queue needs to be two times the RTT $\times$ bandwidth product of the network. MFLOW can 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 [3]. 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 once a packet is under control of the software, there is no danger of priority inversion due to low-priority packets being processed ahead of high-priority packets. This is one of the most significant advantages of paths. For example, the early separation makes it possible to run a video stream while flooding the network adapter with small Ethernet packets.

This is demonstrated in Table 2, which shows how the maximum decoding frame rate for the Neptune video drops when load is added to the Scout and Linux systems, respectively. The additional load consists of a flood of ICMP ECHO requests (generated with `ping -f`). In the Scout case, the video path is run at the default round robin priority, whereas the path handling ICMP requests is run at the next lower priority. In contrast, Linux handles ICMP and video packets identically inside the kernel. As the table shows, adding the ICMP load has little effect on the frame rate for Scout, while the maximum framerate for Linux drops by more than 42%. Clearly, the early separation afforded by paths can have significant benefits. This is not to say that paths are the only way to solve this particular problem (e.g., [22]), but it does support our claim that paths can be an effective solution to such problems.

	Framerate [fps]		
	unloaded	loaded	Δ
Scout	49.9	49.8	-0.2%
Linux	39.2	22.7	-42.1%

Table 2: Frame Rate Under Load

While the advantages of paths due to early separation are important, 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 adversely affecting realtime streams running in the foreground.

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 sub-optimal.
- 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, Scout uses EDF scheduling for real-time MPEG paths. For example, this 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 large number of missed deadlines if the output queues for the Canyon movies are large.[3] 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. The reason for the poor performance of round-robin scheduling is that it keeps scheduling the 8 Canyon movies as long as their output queues are not full, even at times when the Neptune movie needs the CPU much more urgently.

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 some number of packets (n) should be in the transit to keep the network pipe full, MFLOW must be able to advertise an open window of size n. This means that the deadline is the time at which the input queue would have less than n free slots. This time can be estimated based on the current length of the queue and the average packet arrival rate.

[3] Because it is difficult to compute globally meaningful priorities for RM scheduling in a dynamic system—i.e., one where different rate videos come and go—single-priority round-robin is the next best alternative. Therefore, we compare EDF to this case, rather than to RM.

Since the path object provides direct access to both queues, 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 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

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, this is due to the fact that it is easy to compute the execution time spent per path—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 Related Work

At a superficial level, Scout paths are similar to UNIX pipes [28] and Pilot streams [25]. While all three abstractions have in common a linear sequence of "components" (processes in UNIX, Mesa modules in Pilot, stages in Scout), neither pipes nor streams provide any global context to the individual modules. Neither do they attempt to optimize the code along a "path." It is also the case that UNIX pipes are more coarse-grain and uni-directional.

As mentioned in Section 1, there is a wealth of mechanisms that offer point-solutions to the more general problem of exploiting paths, both as a structuring framework and as an optimization technique. This related work falls broadly into two categories, depending on their primary objective:

- optimizing code along the "fast path," or
- improving resource management.

Examples of fast path optimizations include Synthesis [19], Synthetix [24], PathIDs [17], Protocol Accelerators [29], and integrated layer processing [4, 1]. Examples in the second category include processor capacity reserves [21], distributed/migrating threads [5, 9], and Rialto activities [16]. Because space does not permit us to contrast all of this work in detail, we simply point out that the path abstraction as presented in this paper is an attempt at unifying these various ideas. In particular, the proposed abstraction allows us to reason about both the fast path and resource management issues. Our claim is that the unifying principle behind this abstraction is the global knowledge that paths afford. The rest of this section discusses the related work we consider most relevant in more detail.

The system that is probably closest to Scout, at least in terminology, is Da CaPo (dynamic configuration of protocols) [11]. It defines an infrastructure for building multimedia protocols. While Da CaPo has a notion of paths and stages, there are important differences at all levels. At lowest level, Da CaPo paths are uni-directional and stages have very restricted functionality (they are essentially non-blocking event-handlers). As a result, Da CaPo is just powerful enough to accommodate common networking tasks. Also, interoperability with existing protocols is not a goal of Da CaPo. Another important difference is that path creation is left completely to an external "configuration manager." As pointed out in Section 2, this means that in any reasonably complex system, the configuration manager will be burdened by detailed knowledge of the internal workings of particular protocols. In contrast, our path abstraction makes it easy to exploit both local and global knowledge during path creation. At a higher level, Da CaPo focuses completely on *automatically selecting* appropriate protocol functionality; performance and resource allocation appear to be secondary issues.

Kay [17] introduces the notion of a PathID, which is designed specifically to reduce the latency of receive-side network processing. Fundamentally, a PathID is similar to a fine-grained virtual circuit identifier (VCI) in ATM networks [7]. Since PathIDs are stored in a known location in the header of network messages, packet classification becomes trivial (in the worst case a table-lookup). In [17], packets with a PathID are processed by highly optimized, handwritten code. Since this code is manually tuned, maintainability and ease of use are problematic. In fact, the paper suggests that PathIDs should be used for the rare cases where having to maintain two parallel branches of source code is a justifiable cost. Finally, PathIDs do not attempt to elevate paths to a fundamental OS structure, and the problem of creating paths without human direction is not addressed.

6 Concluding Remarks

This paper makes two contributions. First, it describes how paths can be made an explicit OS abstraction, and shows how this abstraction has been implemented in the Scout operating system. Second, it makes a case for why paths should be made explicit. This case includes both the intuitive arguments made in Section 1, and a demonstration of how paths proved beneficial in one particular application—receiving, decoding, and displaying MPEG-compressed video. On this latter point, we showed how paths are used to:

- segregate work early 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, active network router, 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, Patrick Bridges, David Larson, and Rob Piltz. Thanks also to the reviewers, especially our shepherd, Rich Draves. This work supported in part by DARPA Contract DABT63-95-C-0075 and NSF grant NCR-9204393.

References

[1] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, Oct. 1993.

[2] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PathFinder: A pattern-based packet classifier. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, pages 115–123, 1994.

[3] R. Braden, D. Clark, and S. Shenker. RFC-1633: Integrated services in the Internet architecture: an overview. Available at ftp://ftp.internic.net/rfc, July 1994.

[4] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. of SIGCOMM '90 Symp.*, pages 200–208, Sept. 1990.

[5] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. In *1993 Winter USENIX Conf.*, pages 127–146, Apr. 1993.

[6] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the 14th ACM Symp. on Operating System Principles*, pages 189–202, Dec. 1993. ACM.

[7] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proc. of SIGCOMM '94 Symp.*, pages 2–13, Aug. 1994.

[8] D. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. of SIGCOMM '96 Symp.*, pages 53–59, Aug. 1996.

[9] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *1994 Winter USENIX Conf.*, pages 97–114, Jan. 1994.

[10] J. Gosling, F. Yellin, and The Java Team. *The Java Application Programming Interface*. Addison-Wesley, Reading, MA, 1996.

[11] A. Gotti. The Da CaPo communication system. Technical report, ETHZ, Switzerland, June 1994.

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

[13] G. Hamilton and P. Kougiouris. The Spring nucleus: a microkernel for objects. In *Proc. of the Summer 1993 USENIX Conf.*, pages 147–159, June 1993.

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

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

[16] M. Jones, P. Leach, R. Draves, and J. Barrera. Support for user-centric modular real-time resource management in the Rialto operating system. In *Proc. of the 5th Intl. Workshop on Network and OS Support for Digital Audio and Video*, pages 55–66, Apr. 1995. ACM.

[17] J. S. Kay. *Path IDs: A Mechanism for Reducing Network Software Latency*. PhD thesis, University of California, San Diego, 1995.

[18] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1(20):46–61, Jan. 1973.

[19] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY, Sept. 1992.

[20] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *1993 Winter USENIX Conf.*, pages 259–269, Jan. 1993.

[21] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: An abstraction for managing processor usage. In *Proc. of the 4th Workshop on Workstation Operating Systems (WWOS-IV)*, pages 129–134, Oct. 1993.

[22] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *1996 Winter USENIX Conf.*, pages 99–112, Jan. 1996.

[23] D. Mosberger, L. Peterson, P. Bridges, and S. O'Malley. Analysis of techniques to improve protocol latency. In *Proc. of SIGCOMM '96 Symp.*, pages 73–84, Sept. 1996.

[24] C. Pu et al. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. of the 15th ACM Symp. on Operating System Principles*, pages 314–324, Dec. 1995.

[25] D. D. Redell et al. Pilot: an operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, Feb. 1980.

[26] R. V. Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing*, pages 80–89, Aug. 1995.

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

[28] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[29] R. van Renesse. Masking the overhead of protocol layering. In *Proc. of SIGCOMM '96 Symp.*, volume 26, pages 96–104, Stanford, CA, Aug. 1996. ACM.

[30] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symp. on Operating System Principles*, pages 203–216, Dec. 1993.

[31] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *1994 Winter USENIX Conf.*, pages 153–165, 1994.

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