Streamline presents the Unix I/O interface and extends it where needed. Streamline extends pipelined I/O to the kernel through a virtual filesystem (3.2). Shell pipelines are its language of application construction (3.3) and Posix I/O calls give access to pipe contents. It implements Unix socket and pcap interfaces on top of its native interfaces for legacy support (3.4).
3.1 Overview

Because streaming I/O is a well established domain with many users, we prefer to reuse proven interfaces in Streamline over reinventing the wheel. We base our syntax on Unix I/O [Rit84] and only deviate when application or performance constraints demand it. More important than syntax, even, we follow the principles behind Unix, commonly referred to as the “Unix Philosophy”. Two principles are especially relevant. M. Douglas McIlroy, the creator of Unix pipes and pipelines, formulated the now famous maxim [Sal94]

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

When every resource takes the same form (a text stream) it can be represented through a uniform interface. A second well known principle in Unix, therefore, is that “everything is a file”, i.e., that (nearly, in the case of Unix) all resources live in the same filepath namespace and expose the same file interface.

To increase throughput, Streamline extends the reach of Unix design. Considerable high performance I/O takes place in kernel and device pipelines (network, video, audio and graphics processing). To open up all system I/O to users, Streamline presents all transient kernel streams as files in a virtual filesystem. It extends the Unix Shell pipeline syntax to kernel processing to avoid hand off between incompatible pipeline implementations. Systemwide interface uniformity ensures that the performance enhancements of the next chapters apply universally. This chapter does not tackle performance itself; it only describes the Unix interface refinements.

An I/O Filesystem (3.2) opens up otherwise opaque kernel I/O. In line with the Unix philosophy, Streamline exports all transient system I/O as pipes in a filesystem, so that all existing tools and languages can access it.

Extended Pipelines (3.3) can express applications that Unix shells cannot, among which any directed graph. Besides full digraphs, Streamline extends the language with optimization hints and conditional paths. The latter are the basis for fast filtering, Boolean selection and connection multiplexing.

Legacy Unix APIs (3.4) guarantee backward compatibility. Streamline implements the common sockets and packet capture interfaces, which we use for evaluation. Uniquely, it implements them on top of the native interfaces, to trivially reuse all their throughput enhancements.
The filesystem and pipeline language complement each other. The filesystem opens the kernel to I/O monitoring, control and access, but this interface is too inflexible to construct large tasks with: task creation takes many little actions and such micromanagement leaves little optimization freedom. Extended shell pipelines (ESP) solve these problems, because they are concise and amenable to be optimization. These, on the other hand, cannot be used to inspect or modify a running system.

3.2 An I/O Filesystem

To open up kernel and device logic to construction of I/O applications, Streamline extends pipelines to the kernel. It exposes kernel I/O through a virtual filesystem, so that all applications and tools can access it. PipesFS\(^1\) reuses Unix I/O calls to construct pipelines and access pipe contents. It presents a method for open kernel processing that, unlike other kernel level pipelines, is not domain-specific and can be accessed from userspace using all existing applications and languages. PipesFS serves a dual purpose. One, it exposes all active I/O in the kernel to user processes, opening up the otherwise opaque kernel to monitoring, data inspection and even live editing. Second, it directly makes available the low-level pipeline construction API that Streamline internally uses to implement applications. Applications can not only observe kernel pipelines, but modify them at runtime: something that the shell interface does not allow.

In use, PipesFS resembles the Streams example in Figure 3.1 more than the Streamline application interface. Similar to Streams [Rit84], Streamline presents a composition API for imperative programming. Contrary to Streams, however, PipesFS graphs can be modified with any existing tools, shell scripts and languages because the construction interface is based on a filesystem and Posix I/O calls. PipesFS is a Linux virtual filesystem that visualizes Streamline kernel filters and streams as a directory structure, where filters are directories and nesting symbolizes data flow from the parent to its child. Symbolic links enable arbitrary additional streams to expand into a full digraph. As a result, it becomes trivial, for instance, to log all webserver requests to a compressed archive. For PipesFS mounted at /pipes the shell command

```
cat /pipes/.../http/get/all | compress > log.Z
```

suffices, whereby we abbreviated the path for clarity. This command reads data produced by an in-kernel get request filter from that filter’s Unix pipe all. The filter first acquired data from another filter, one for http traffic,

---

\(^1\)Presented individually as PipesFS [dBB08c]
which received it from yet a lower layer. This pipeline continues until we reach data sources, in this case a network interface.

Modifying a pipeline, for instance to insert a layer 7 protocol filter when a vulnerability in the webserver is discovered, is as simple. Moreover, the complex protocol filter does not have to run in the vulnerable kernel: a userspace program, which can be as common as `grep` or `sed`, can be placed in between two kernel components. We will return to this example and discuss its performance implications shortly. First, we introduce the filesystem and its components (directories, pipes and files).

**Directories as filters** Each directory in PipesFS represents an active Streamline filter running in the kernel. To create meaningful applications or rewire existing ones, users add and remove directories with the `mkdir`, `rmdir` and `symlink` Linux system calls. Conceptually, data flows down the tree from parent to child directories. The implementation is, of course, based on the same runtime system that executes pipelines. To expand the FS graph type from a tree into the full digraph of Streamline, PipesFS interprets symbolic links to directories as streams.

The filesystem root node, as only exception to the rule, does not correspond to a filter and therefore does not generate any output. Instead children of the rootnode must generate output themselves. Device drivers will for instance do this. The root only contains a single file, `/available`, which lists all available filter implementations. `mkdir` calls in the filesystem will only succeed with names from this list.

**Pipes as output** As filters operate on sequential streams, their I/O can be represented as pipes in the filesystem. PipesFS exports each filter’s output through a pipe in the filter directory. In its most basic form, all output from a filter `/pipes/rx` (for the filesystem mounted at `/pipes`) is not only forwarded to all children; it can also be accessed directly from the pipe `/pipes/rx/all`. PipesFS exposes stream classification by introducing a new subdirectory with its own all pipe for each class of content that is observed. Each logical substream is available through a private numbered pipe `/pipes/rx/N/all`, for each N between 0 and 65535. Subdirectories are only created after their class has been observed in the stream. Additionally, the runtime system can resolve numbers to names, e.g., to replace the lookup for TCP traffic `/pipes/rx/ip/6` with the more friendly `/pipes/rx/ip/tcp`. To make use of this feature, filters must (statically) register a lookup table. If the table exists, PipesFS automatically generates symbolic links from the name to the numbered class pipe.
Files as control  All but the most trivial filters take parameters. In PipesFS, communication between filters and applications takes place through files. In principle, all files in a directory are randomly-accessible memory regions (as opposed to pipes, which only give access to a window of the principally unbounded stream). Writable control files set parameters, read-only files communicate metadata to the application. Small files copy data during each call, large files are backed by the same shared memory regions as pipes (Chapter 4) to reduce overhead of high-throughput communication. Therefore files can serve as general purpose control channels, e.g., to transfer normally in-band socket timestamps. Filters can define as many files as they need and name them freely. To reduce clutter, the runtime system generates files for all the filter’s configuration options prepended with a dot. Because filters names may only consist of alphanumerical characters, the two types of objects cannot clash.

Configuration Management  Directory creation with mkdir constructs a new filter in the kernel and immediate connects its input to the output of its parent. If parameters need to be set prior to connection, the filter has to be created directly below the rootnode, configured and then manually moved to the correct parent directory. Once created, filters can be freely moved and copied. Moving a filter (executing the rename system call on the directory node) closes the input pipe (if any) and opens the new parent’s output pipe. Copying only involves a mkdir call and is therefore equivalent to creation of a new node: no data or state is copied at this point. Copying is useful to replicate whole subgraphs. Because a recursive copy is not atomic, input has to be temporarily severed to guarantee consistent behavior. Moving the source node to the root level for the duration of the operation suffices, but this must—again—be done manually. This demonstrates once more that PipesFS requires more administration than pipelines, because it operates at a lower layer of abstraction.

Read and write access  PipesFS not only exposes the kernel I/O configuration, it also makes all live streams accessible from userspace. As the logging example demonstrated, reading data requires the same actions as reading from regular pipes. Similarly, inserting only involves writing to a pipe. Editing the contents of an I/O stream is more involved. The stream must be severed — again by moving the child directory to the root level — and data manually forwarded from the (former) parent to the child. This entails reading from the output pipe, applying any required transformation to this data and then writing it to the input buffer. At this point we continue with the layer 7 (i.e., deep
inspection) protocol filter example that we introduced at the start of this section. Traditionally, to filter at this level, all network data must be intercepted (in the kernel) and parsed up to the protocol level, because no hooks in the regular socket handling code exist to attach to. With PipesFS, not only can we move processing out of the kernel (if we wish, this is not a requirement), protocol processing is not unnecessarily duplicated, because each phase of the socket pipeline can be attached to. Let’s say we want to drop all HTTP requests containing the unsafe double-dot (..) notation before they reach our server. The following shell script achieves this (albeit crudely).

```bash
mkdir /pipes/httpclean
mv /pipes/[...]/http/get /pipes/httpclean/
cat /pipes/[...]/http/all | grep -v '..' > /pipes/httpclean/all
```

It creates a new directory at the filesystem root, moves the HTTP GET filter in here, then manually reads the HTTP packets from the original parent, scans them for the double dot (with grep, which drops matches when -v is given) and writes the results to the outgoing filter’s parent’s pipe.

To minimize overhead from data duplication between filters, PipesFS uses indirection habitually, using Beltway buffers (Chapter 4). Each intermediate stream is represented using an equivalent of a pointer queue instead of a data queue. Read operations then result in the read of a pointer followed by the read of data block. Even though such indirect access greatly reduces the cost of presenting many streams, it is still not free. For this reason, all buffering is disabled by default. Buffering only commences when a process opens a pipe and stops immediately when the last process closes the pipe. By selectively opening streams users trade off cost for access.

### 3.3 Extended Pipelines

A good application interface is simple and concise. Many streaming I/O systems expose a graph composition API. Figure 3.1 reproduces a code snippet from the Solaris 10 STREAMS programming guide [SM05]. The example (2−2 in the manual) inadvertently drives home the point that explicit programming of streams and filters is verbose and error prone: it takes eight lines of code to insert one filter. More importantly, it leaks implementation details about its kernel operation, leaving no room for optimization. The virtual filesystem in Streamline exposes a similar API. Besides this, Streamline also presents the user with a more abstract declarative interface, that is more concise and amenable to optimization.
Streamline presents a pipeline interface and borrows its syntax from Unix shells. It goes beyond shell programming by applying the language also to protected kernel and device operations. For instance, the pipeline

```
rx | tcp | http | files
```

reassembles all incoming TCP streams, selects connections that contain HTTP requests, extracts these requests and saves them to individual files. Straight pipelines cannot express the SafeCard application that we introduced in the previous chapter. With an extended syntax, it is most naturally expressed as

```
rx | head | ruler | ok + (ok |! prospector) | flow | tx
```

although this is not necessarily the only form. As the SafeCard example shows, the syntax has to be expanded to encode some practical applications. This section presents the extensions in Streamline and explains their use-cases.

Pipeline programming is declarative in the sense that it defines application behavior without specifying how that behavior must come about, leaving it up to the implementer to decide. Unix shells do not actively exploit this freedom, but map pipelines unconditionally onto Unix processes and pipes. Neither does the operating system, although it could conceivably use behavioral pipeline information, for example, to make more informed scheduling decisions. It cannot, however, because this structural information is lost at the system interface where the pipeline is split with `fork` and `pipe` calls. Unix specifications on operating semantics and system (call) interfaces restrict operating liberty. Although Streamline borrows the syntax of Unix, it but does not limit itself to these rules.

**Extended Unix Shell Syntax** To be able to express common networking tasks, such as load balancing, protocol demultiplexing and connection handling, Streamline adds two base features to the pipeline: parallelism and conditional execution. On top of these, it then constructs optional higher level control flow, such as looping, Boolean expression and connection handling, without changing the basic pipeline and at no additional cost.
### Interface

#### Filter

<table>
<thead>
<tr>
<th>F=&lt;name&gt; [key=value]*</th>
<th>Native filter with optional control options</th>
</tr>
</thead>
<tbody>
<tr>
<td>F=&lt;name&gt; [param]*</td>
<td>Unix process</td>
</tr>
<tr>
<td>F=&lt;name&gt; [key=value]* &quot;[param]*&quot;</td>
<td>Unix process with optional control options</td>
</tr>
</tbody>
</table>

#### Selector

| S=ǫ         | Default, all non-zero classes |
| S="all"    | All classes                   |
| S=<0-9>+    | Select a class                |
| S=<0-9>+:<0-9>+ | Select a range of classes    |
| S=<a-zA-Z><a-zA-Z0-9>++ | Select a named class          |
| S=!S        | Inverse of another selection |

#### Expression

| A=F          | Atomic expression consisting of one filter $F$ |
| A=(B)       | Nested expression                            |
| A=B | S C | Pipeline with selector $S$ |
| A=B + C     | Parallel streams                             |
| A=B * B     | Logical union ($\lor$) of two streams         |
| A=B & C     | Logical intersection ($\land$) of two streams |
| A=!B        | Logical negation ($\neg$), equivalent to A $\mid$ B |
| A=B @<key>[,<key>]* | Expression with explicit optimization hint(s) |

Table 3.1: Extended Pipeline Syntax

Deviation from the proven model sets one up for criticism in the spirit of Henry Spencer’s famous quote

> “Those who do not understand UNIX are condemned to reinvent it, poorly” [Spe87]

Such skepticism is valid, but not pertinent here. Extended pipelines are fully backward compatible with Unix. Streamline accepts all Unix pipelines and Streamline pipelines can embed all Unix applications. The language is only extended when needed to introduce requested features that cannot be feasibly implemented another way. Here, too, we chose the simplest solution with the broadest application. To paraphrase an even more acclaimed admonition than the one above, we keep the model as simple as possible, but no simpler. Table 3.1 summarizes the complete syntax.
Before presenting the language extensions in detail, we illustrate that the existing pipeline language suffices to express many, if not most, applications. For one, Streamline can carry out all existing Unix pipelines. It prefers native filters over Unix processes, but if no native implementation of a filter can be found, it will search for a Unix process. Therefore, the Unix expression

```
ps ax | grep slcli
```

is a valid Streamline request (it lists all processes and selects those that contain the expression ‘slcli’, the streamline interpreter). The interpreter is not a full fledged shell, however: it lacks support for environment variables and backticks, for instance. In other words, the expression

```
wc ‘find *.txt’
```

will fail. Besides Unix processes, however, Streamline pipelines can call native library filters. For instance, the request

```
tcpsock 5050 | deflate | tcpsock "127.0.0.1:5051"
```

sets up a tunnel between two TCP sockets and compresses the data stream. These filters are not full processes, but share a single userspace task (we explain these implementation details in Chapter 5). Moreover, filters need not execute in a user process: this networking example involves kernel processing

```
rx | bpf "src port 80" | inspect | outfile log.txt
```

It reads all data arriving from the network, filters out packets originating from a probable website, generates packet information records and writes these to a logfile. This pipeline combines network processing in the kernel with text processing in userspace. At present, Streamline has more than 40 native filters that can be combined into pipelines with the larger set of Unix processes. We now identify recurrent application patterns that linear pipelines cannot implement and present language extensions that amend this situation.

### 3.3.1 Parallelism

A pipeline is a directed graph with computation *filters* for vertices and communication *streams* as edges. In the strictest definition, the network topology is constrained to a linear alternating sequence of the two elements. As this strict definition is the basis of Unix shells, it can hardly be called impractical. Nevertheless, shortcomings have been observed in practice. CMS pipelines [Har07] extend Unix pipes with task parallelism. A similar feature,
multiplexing, was also introduced to Streams [Rit84] in a language revision. About the lacuna in the original language, Ritchie himself writes:

“It seems likely that a general multiplexing mechanism could help [...], but again, I do not yet know how to design it.” [Rit84]

We equally observed the need for expressing parallelism. Many network tasks have mutually independent series of operations, where filters in both pipelines want the same copy of data, not each other’s output. For instance, a virus scanner applies multiple filters side-by-side. Moreover, explicitly expressed disjunctions can be executed concurrently on multiprocessors. Perhaps because our use of streams is limited to a single host, introduction of parallelism posed none of the serious philosophical or technical challenges Ritchie alludes to.

Conceptually, Streamline introduces task parallelism by extending filters with connection points called ports (a common moniker, not our coinage, and a proven method for introducing task-parallelism). By default, filters have 32 ingress and 32 egress ports. Ports are different from the Unix concept of file descriptors and how these are used in standard pipes (stdin, stdout and stderr) in that filters do not explicitly transmit data on a single output, but multicast the same pair of data and classifier onto all output ports. This model simplifies the implementation of filters, which can remain unaware of the number of inputs and outputs in use and, instead, only concern themselves with processing and sorting data. Figure 3.2 depicts a task-parallel graph of streamline filters across a set of computation spaces (Section 2.2.3), connected through ports. The picture zooms in on one filter to show that the filter implementation is internally unaware of the number of ports. It also shows that filters can optionally have private state that persists across invocations. Finally, it shows how Streamline can map multiple filters onto a single computation space, contrary to Unix shells that always map a program onto a userspace process.

Syntax

Streamline syntax is based on that of Unix shells. It introduces two language extensions to expand processing to full directed graphs. Split operators deviate only minimally from established practice and are intuitive. Because these cannot encode all directed graphs, we also introduce named nodes.

**Split/Join** The plus symbol (‘+’) identifies parallel compositions, as opposed to the sequential compositions of the pipeline (‘|’). The following request, for instance, defines a firewall that only accepts three protocols.
Each pair of split and join node encapsulates a subgraph that is otherwise disjoint from the greater network. Since the minus operator is binary, each occurrence introduces one such pair of graphs in principle. With multiple operators, requests can become ambiguous; Streamline adds parentheses to explicitly express evaluation order. If no parentheses are used, the plus ranks above the pipe. Thus, the chosen example is equivalent to this following interpretation.

In practice, Streamline prunes unnecessary levels of nesting to speed up execution prior to instantiation. Another concern in parallel graphs is that each split duplicates data (logically, not necessarily physically). Unless these are pruned, duplicate blocks will then be observed downstream from where the streams join. The following figure shows a pipeline with split and join nodes. The filter behind the join observes duplicate streams:

For some filters this is useful information. For instance, anti-virus scanners execute a set of tests and correlate results. Most filters want these duplicates filtered out, however. The next section shows that besides the unfiltered plus operator, the language has Boolean And and Or operators for calculating stream union and intersection.

Backlinks

Pipelines with only split-join parallelism cannot execute all potential applications, because the language does not express a full digraph. Because they
do express the majority of practical tasks concisely, Streamline does not replace pipeline syntax in favor of the common digraph notation $G = (V, A)$, but extends the language with a simple construct that removes the last shortcoming. Extended pipelines can express the class of series-parallel digraphs. That this is the representable class is easy to see, if we consider the binary nature of the operator: each additional plus symbol must also introduce one new parallel and otherwise disjoint subgraph. Similarly, each pipe introduces a new sequential subgraph that is otherwise disjoint. Since these graphs are not connected to the rest of the graph by any means but the forward arc (for a pipe) or forward split-join construct, no cycle back into the previously expressed graph can ever originate from within this new graph. Note that the term back here refers to the entire previously described graph, not just the part upstream from the current node. Figure 3.3 shows a DAG that cannot be expressed with only split-join parallelism. It clearly marks the backlink, which in this case points to a vertex that is not upstream and therefore does not create a cycle. Because each subgraph is either an atomic filter or recursively constructed from other nested subgraphs, no subgraph can ever contain a cycle or a crosslink to another subgraph. Nor can the top-level graph, because it is just a special instance of the recursive process.

Streamline extends split-join graphs into full digraphs through explicit backlinks: arcs that flow back to earlier defined parts of the graph to form cycles. To express these, it extends the language with named nodes: sets of vertices in the request graph that correspond to a single vertex in the runtime graph. Named nodes are identified in the request graph by their shared name. As a result, they make it possible to express a backlink as a forward arc. For example, the expression

\[
\text{name}=X \mid b \mid \text{name}=X
\]

maps onto the cycle

```
A \rightarrow B \rightarrow A
```
With backlinks, it becomes possible to create applications with unlimited loops and correspondingly infinite execution time. With conditions, safer bounded loops can be built, instead. The next section explains how Streamline supports conditional execution and then presents a conditional loop example.

### 3.3.2 Conditional Execution

In streaming I/O, conditional execution means the selective forwarding of data along a subset of all possible paths. Load balancing, protocol demultiplexing and packet filtering are example applications of this feature. These tasks do not flood the graph, but forward data from one filter to the next only if certain constraints on data contents and system state are met. They are common tasks in I/O processing and require an efficient implementation.

Streamline implements conditional execution as a language feature. This decision extends the syntax, but enables faster execution of common operations. Conditions dictate data movement through the system: one of the main causes of overhead in an I/O system. Because the runtime system has a bird’s eye view, it can optimize transport in ways that filters cannot. For instance, it can replace storage to a queue with pointer forwarding if filters operate in the same space, while filters would always have to enqueue data in the same manner. In Unix pipelines, filters (processes) handle all data directly through reading from and writing to pipes, which causes considerable copying between processes. This cost would be unavoidable if each filter edits data beyond recognition, but that does not seem to be the case. Although some filters thoroughly edit stream contents, many only want to read, group, split, discard or append it, all of which can be implemented more efficiently than through repeated copying of the majority of unmodified bits. For instance, in networking, many filters touch data only for reading, such as in protocol demultiplexing, or to make minor changes, such as in header stripping. There is no reason in either case to write any bytes – let alone to duplicate all.

Streamline increases speed over Unix pipelines by minimizing data touching by successive filters for all these cases. Chapter 5 classifies filters in behavioral classes and selects fast implementations for each. The fast implementations of grouping, splitting and discarding are all based on conditional execution: on sorting of data items into groups and subsequent selection (splitting and discarding) of arcs to traverse based on the data’s group.

**Sorting** Filters attach meaning to data blocks by grouping them according to features. Groups are identified through a scalar classifier. The meaning attached to groups is not dictated by Streamline, but task-specific to support
many uses. Scalars are a cheap way to communicate data type, connection number or filter state. For instance, an MPEG video decoder can announce the frametype of a data block to downstream processors, a TCP reassembler the hash of a stream and a virus scanner whether an intrusion has been seen.

Sorting is implemented by running a stream of classifier values (a “metadata stream”) parallel to and in lockstep with each data stream, as depicted in Figure 3.4. Alongside the data, Streamline makes these numbers visible to all downstream filters. Besides a classifier value, this metadata item, or tag, contains an offset and length pair to be able to express interest in a subsection of a block (used, for instance, to remove protocol headers cheaply; we will see more uses later). It also embeds a lookup structure, but that is immaterial now (we return to that point in Chapter 4). The smaller the tag, the more fit in a cache together; in the current implementation, the classifier is a 16 bit scalar, which satisfies all tasks so far (specifically, it is long enough for TCP port splitting). The total tag is 16 bytes. Some filters can perform operations solely on the contents of tags, greatly reducing data access per operation. For instance, a zero-copy TCP reassembly filter (Section 5.3.2) only looks at the offset and length pairs of segments to reorder them. By avoiding the data load and store phases at each filter, base overhead drops, enabling more fine-grained operations and thus increased opportunity for optimization.

The scalar communication channel is similar to the return value passed by Unix processes, except for the fact that a scalar value is produced not at process exit, but for each data item. For instance, an HTTP GET filter signals after each block whether it found an HTTP GET request embedded. Downstream, a firewall filter decides based on this outcome whether to forward, log or drop the block. In line with this semantic change, the values change: where in Unix non-zero values commonly represent error codes and zero signals success, in Streamline non-zero classes represent valid data groups and zero is generally used for the group of data that should be discarded. The interpretation of the scalar value is application-specific, however, and the responsibility of individual filters.

To ensure compatible interpretations of data and metadata streams, filters can explicitly label their in- and output types. The simplest use is to differentiate between record- and bitpipe semantics, but types are further specializable. The Ethernet frame, for instance, extends the record with semantic information about the first bytes; a TCP stream combines a bitpipe with connection multiplexing based on the classifier value. In the reference system, most filters process arbitrary data, but some accept only Ethernet contents, such as a network traffic monitor.
Selection  To complete conditional evaluation, each arc can constrain which selected groups to forward. The simplest type of filtering is a binary decision to forward or drop data. A default pipe forwards all classes of data except for class zero. More interesting applications explicitly restrict forwarding, by annotating the pipe `|' symbol. Applications writers set selection constraints on streams by appending one of the following `selectors': a single scalar \( m \) representing a single class, two values \( m : n \) representing a range of classes or a literal string representing a named class, which uses the lookup table described in the next section. For example, to forward traffic to destination port 22 to an SSH filter, we write `\( \text{dport} \mid 22 \ C S \ D 4 \ D 3 \ D 6 \ D 8 \ D G \ B E \ B E \ D 7 \ D 7 \ C W \)`. Ranges are given by a pair of scalars and a colon: `\( \text{rx} \mid 22:23 \ C D \)`. Additionally, selection may be inverted by inserting an exclamation mark between the pipe symbol and the main expression. For instance, on a webserver device that only accepts port 80 traffic, all other data is reflected to an intrusion detection system using `\( \text{dport} \mid !80 \ C X \ C S \ D 7 \)`. Ranges and inversion may be combined, as in `\( \text{rx} \mid !1:1023 \ C A C T \)`.  

At runtime, each time a filter produces data Streamline compares the accompanying class value to the ranges of all outgoing arcs and only schedules the downstream vertices for matching edges. Classification can forward and filter out data without having to explicitly copy data. Since data passing through streams is one of the most executed parts of the runtime system, keeping this task cheap is paramount.

Application
Conditional execution can spread data across parallel arcs and filter out traffic. In networking, metadata stream filtering is the basis for cheap reception handling and data demultiplexing. With parallel paths and stateful filters, conditional paths can also support other popular data manipulation operators. In Streamline, they implements Boolean queries, connection multiplex-
ing and conditional looping.

**Boolean Selection** Often, users want more fine-grained filtering based on the intersection, union or inversion of streams. *Not* subtracts one stream from another, *And* calculates the intersection and *Or* the union of two streams. Streamline introduces notation for Boolean selection and implements these on top of class-based filtering, i.e., without further changes to the basic pipeline. We append another plus symbol (`+`) for split-join parallelism with duplicate removal – in other words, as the union of a set of streams which is equivalent to a logical *Or* operation. Analogously, we denote intersection, or the logical *And*, by adding a multiplication symbol (`+*`) and negation with the exclamation mark (`!'`). All three can be seen in the example

```
rx ++ tx | ip | mpeg ++ !http
```

Here, all network traffic is sent to two protocol filters; only video over anything but HTTP is kept. Intersection only passes blocks for which all parallel filters return a non-zero class, and passes each block at most once. Union passes blocks that have at least one non-zero path, again no more than once. Logical negation, `!B`, passes all blocks dropped by a filter:

![Diagram](image)

The operators are implemented as graph rewrite rules (templates, Section 6.2.1) from the Boolean-augmented grammar to the base grammar. Logical *Or* is rewritten as a join operation. The join filter passes the *i*th occurrence of the same block, where in this case *i* = 1. Logical *And* is replaced by the same filter, but passes not the first match, but the *N*th. Negation could be implemented using a join, but is not, because a much simpler method is available. The operation is translated into condition inversion on the edges, so that `!A` | `B` becomes `A` || `B`.

**Implementation** Join operations can be computationally expensive. They need to compare blocks to identify duplicates and store copies of all blocks for which duplicates may still arrive. In most cases, execution can be sped up, however. The simplest approach is to always forward the first occurrence of each block and drop all additional copies. This can be implemented cheaply without requiring state or even data access, if the filter is executed in a round-robin fashion by all disjoint subgraphs. Then, no two blocks can cross each other. The filter simply continually passes one block and then drops the N-1
next (for an N-way split). Because the approach requires ordered filter execution and maintaining strict order on parallel hardware is difficult, it can only be applied when the entire subgraph executes as a single thread of execution.

We could (but have not, at this point) implement yet another optimization. All types of joins can benefit from ordering execution so that only one path needs to be traversed. When all but one of the disjoint subpaths are computational kernels and the runtime system executes filters in depth-first order, the first parallel path will execute the join filter before any other path is even started. If the join succeeds, the alternate paths do not have to be scheduled. This kind of evaluation optimization is similar to the strict expression evaluation order in computational languages such as C.

Connection Multiplexing  Many network and video streams applications operate on connections, such as web servers. For applications with many short-lived connections (the quintessential example of which is a web server), addition and deletion of pipes at runtime for each connection is too expensive. Connections are more efficiently handled by multiplexing them over a single long-lived stream. For example, the expression (written in Unix shell syntax)

```
 in | ip | tcp | pcap | compress | disk
```

encodes a request that filters IP traffic, reconstructs independent TCP streams and stores each to an individual compressed tracefile.

Connection multiplexing is difficult to implement in a generic fashion, because it requires bookkeeping (and hence state) and because this state must be shared between otherwise independent operations across protection boundaries. Layer-3 routers, including those based on streams such as Click [KMC+00], have no need for connection multiplexing because they act only on discrete packets. Other network stacks differentiate streams internally within subsystems, such as within the TCP protocol handler, but lack a common interface to share this information with other components. Such solutions are task-specific and localized, causing multiple code blocks to reimplement the same session bookkeeping and to execute the same logic to assign blocks to sessions. In Streamline, instead, filters can agree to use the classifier to discern among logically independent streams. Then, only the first filter in a pipeline has to assign a filter to a session; all successive filters refer to the classifier value to group blocks into sessions. The TCP stream reassembler, for instance, sets the tag to differentiate among many independent TCP streams over a single pipe. By exposing the class field downstream, it can form the start of connection-oriented pipelines. All connection-oriented filters understand multiplexing.
Stateful connection-handling filters will segment state between connections. For instance, we will see later that centralized state multiplexing makes it possible to automatically extract data parallelism from stateful connection-oriented filters. (Section 5.4). For now, we simply mention that Streamline can automate the task of segmentation and lookup, because the runtime system can access the classifier. It exposes a special session-aware allocation function to filters that resembles calloc:

```
slsession_alloc(size_t nmems, size_t size)
```

Through this, filters request not a single memory region, but enough memory to hold a fixed number of fixed size session structures (when the filter is first loaded). During processing, Streamline maps classes onto sessions. Each time a session-handling filter is ready to be scheduled, it looks up the corresponding session region by the class value and updates a pointer in the filter datastructure to point the filter to the correct region.

**Conditional Looping**

Some applications want to process data repeatedly, for instance multi-pass image filters. Streamline combines parallelism, conditional execution and stateful filters to enable conditional looping, which lifts pipelines to the class of Turing complete languages. This is most easily demonstrated by showing equivalence with other known TC languages. The extended pipeline can implement the basic control flow statements of imperative programming: conditional branching and looping. In this context filters are computational basic blocks – instruction sequences uninterrupted by jumps or branches – and control flow operations govern movement between these blocks.

A conditional branch is constructed by giving a vertex two outgoing paths with opposing constraints. Either data travels along one path, or it travels along the other, but never on both (or neither). Each binary filter, therefore, is a branch. Unbounded loops, then, are conditional branches with one condition pointing back to the start of the current basic block. Or, in graph terms, a subgraph with a backlink to itself.

Loops are bounded by introducing special bounding filters. The following example executes a certain subgraph, $G_{inner}$ ten times:

```
G_{pre} | for name=\text{X} 10 | G_{inner} | for name=\text{X} 11 | G_{post}
```

Here, the `for` filter is shown twice, both with the same name. The first occurrence also sets a parameter: a limit of 10, which controls the number of times the filter must accept a block before dropping it. The filter keeps an internal counter that is decremented each time the same block is observed. Blocks are
scheduled depth-first in Streamline, which ensures that no two blocks can intersect, simplifying counting. Each time the counter is not zero and the same block is seen, the filter classifies the block as a positive number, which the regular pipe will forward to the subgraph $G_{inner}$. Once it reaches zero, the class becomes zero and $G_{inner}$ is not called. $G_{post}$, on the other hand, is attached to a pipe with an exclamation mark. That inverses selection, causing all blocks classified as zero to pass.

**Safe Execution** Like any language that supports unbounded looping, the extended pipeline can trivially express non-terminating applications. Because control flow is explicit, all loops can be identified. Bounded loops are safer, because they are guaranteed to exit at a given iteration. The distinction between bounded and unbounded looping cannot be made automatically, but a practical solution can be devised in two stages. First, strict safety guarantees can be given based on loop detection: directed acyclic graphs can be detected and are certain to be bounded for any given input. This method excludes some safe graphs (i.e., it suffers from false negatives): those that contain loops, but are still safe. Second, trustworthy pipelines with bounded loops can be constructed by only allowing DAGs or graphs where looping is implemented using trusted loop filters. Section 6.5 introduces a security policy framework with these properties. Note that this guarantee says nothing about the safety of the computational filters themselves. Ensuring their safety is a related topic treated in the same section.

### 3.3.3 Options

Because in Streamline the mapping of pipeline to hardware is less clearcut than in Unix shells, occasionally options have to be passed to the control logic. Besides their own parameters, filters may be given one of the control options summarized in Table 3.2. *Export* and *exportdata* force buffering of data even when transport optimizations would normally avoid this. Users pass these options when they want to be able to inspect or modify a stream from their application logic. *Share* and *reuse* govern whether multiple pipelines can share filters. When many applications share subgraphs, such as in the case of socket processing, sharing avoids work duplication. Default behavior is to share stateless filters and to wipe state when filters are updated or at runtime (e.g., through manipulating the virtual filesystem). *Space* overrides Streamline's optimization algorithm and forces implementation in a given space. *Mode* overrides the access permissions as discussed in the access control section (6.5). The *binary* and *file* options enable indirect parameter passing to filters. Some filters take parameters that cannot be practically expressed on the command line, such as interpreters that accept bytecode
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>&lt;string&gt;</td>
<td>Uniquely identify filter and optional stream</td>
</tr>
<tr>
<td>export</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>exportdata</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>size</td>
<td>&lt;len&gt;</td>
<td>Explicitly set buffer queue length</td>
</tr>
<tr>
<td>share</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>reuse</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>space</td>
<td>&lt;id&gt;</td>
<td>Force a space, can also be 'local' or 'kernel'</td>
</tr>
<tr>
<td>mode</td>
<td>&lt;perm&gt;</td>
<td>Override default access permissions</td>
</tr>
<tr>
<td>file</td>
<td>&lt;name&gt;</td>
<td>A file whose contents to pass as parameter</td>
</tr>
<tr>
<td>binary</td>
<td>&lt;id&gt;</td>
<td>A blob from a repository to pass as parameter</td>
</tr>
</tbody>
</table>

Table 3.2: Filter Options

applications. For this reason Streamline accepts native filter (i.e., not Unix processes) parameters expressed as references to the contents of a file or a binary object (blob, previously loaded into process memory). For instance, to avoid retyping regular expressions, these are loaded from a file using

```
file in.txt | regex -file=regex.txt
```

**Optimization Hints** Users may also annotate whole (sub)graphs to guide optimization. Table 3.3 lists the current set of options that Streamline understands. These are passed to a graph as an appendix behind the special symbol (@), as in

```
rx | ip | smtp -file=pattern.txt @throughput,background
```

Here, the whole pipeline is annotated as being a background task whose main goal is to maximize throughput (as opposed to, say, latency).

Different subgraphs can be given different annotations. For example, optimization can change the initial load balancing based on explicit hinting of expected hotspots:

```
tcp | (dport |80 http @busy) + (dport |180 ids @quiet)
```

Just like compiler hinting, graph annotation must be reserved for special situations where the automated system fails. Overuse is a form of premature optimization [Knu74](p.268), which produces fragile applications as a result of hardcoding optimality assumptions that may not be universally true. For normal operations, the automation logic should be left to do its task.
<table>
<thead>
<tr>
<th>Hint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>throughput</td>
<td>Maximize communication throughput</td>
</tr>
<tr>
<td>latency</td>
<td>Minimize worst-case communication latency</td>
</tr>
<tr>
<td>single</td>
<td>Execute in a single space</td>
</tr>
<tr>
<td>private</td>
<td>Run on private (i.e., non time-shared) resources</td>
</tr>
<tr>
<td>realtime</td>
<td>Guarantee at least soft realtime scheduling</td>
</tr>
<tr>
<td>hrealtime</td>
<td>Guarantee hard realtime scheduling</td>
</tr>
<tr>
<td>background</td>
<td>Execute with lowest scheduler priority</td>
</tr>
<tr>
<td>quiet</td>
<td>A relatively silent part of the graph</td>
</tr>
<tr>
<td>busy</td>
<td>An operational hotspot. Opposite of 'quiet'</td>
</tr>
</tbody>
</table>

Table 3.3: Pipeline Options

### 3.4 Legacy Unix APIs

Few existing applications can natively use Streamline’s filesystem or pipeline interfaces. To support a large set of legacy applications, Streamline implements two popular domain specific interfaces on top of its pipelines. For end-to-end networking this is the socket API and for packet filter the aptly named packet capture interface, or pcap. Applications that use these interfaces only delegate the part of their computation behind the interface to the runtime system. As a result, obtainable speedups are lower than with applications that are completely expressed as native pipelines. These interfaces are therefore principally aimed at legacy applications. We will make use of them for one-to-one performance evaluation against Linux, speedup results of which can be similarly interpreted as conservative estimates of achievable optimization.

**Sockets**  
Sockets handle common protocol processing tasks for networked applications. A socket performs multiplexing, fragmentation, stream reassembly, checksumming and header generation. As sockets handle both ingress and egress traffic, each socket is implemented as two pipelines, e.g., for a DNS server, the reception pipeline is

```
rx | udp | dport | 53 socket export=yes name=rxport_53
```

It reads all arriving network data, filters out UDP traffic to port 53 and attaches it to a socket endpoint. This last filter saves all contents to a buffer named `rxport_53`. The socket calls (recv, etc.) operate directly on this buffer. The transmission pipeline is shorter:
The first filter sets up a buffer for transmission, txport_53, and begins to listen on all socket calls on this buffer. On writes, it forwards data to the transmission filter. These pipelines are very short, because a lot of protocol processing is implemented behind the socket calls and rx/tx filter templates (rewrite rules from longer pipelines, Section 6.2.1). The network stack is perhaps the most commonly implemented composable stream program [Rit84, HP91, MMO′94]. Our functional logic does not differ significantly from other systems; the major change lies in the choice of well known Unix interfaces to processing and data. Because they are implemented on top of Streamline pipelines, sockets benefit from the performance enhancements introduced in the chapters 4 and 5. For instance, socket calls execute in the process’s context and thus incur no kernelmode switch.

**Packet capture**  Packet capture is increasingly common. It is the enabling technology, for instance, for network intrusion detection, which has grown tremendously with the uptake of Internetworking, and traffic auditing, which is used by corporations, Internet service providers and national security services to monitor their employees, customers [Sve] and citizens [EU 06].

The most established packet capture interface, simply known as *pcap*, applies a Berkeley Packet Filter [MJ93] to all layer 2 network traffic on a host and sends matching frames to a userspace process. Traditionally, and in Linux today, each frame requires a mode-transition and a copy for each interested process. Shared ring-buffers implement a more efficient capture solution, because they remove all runtime copying and virtual memory mapping overhead (as we will show quantitatively in Section 7.3). Streamline directly maps all layer 2 data to all interested applications, together with the metadata streams that holds the result of a BPF filter (this mechanism is discussed in Chapter 4). It implements the task as a pipeline, so that Streamline can offload the BPF filter to a fast space or replace it by a compiled equivalent as part of its optimization step:

```
rx + tx | bpf_bytecode binary=$BLOB | user
```

The request takes all network sources, applies BPF and ensures that the filtered results are mapped into the application’s memory protection domain. One faster alternative to BPF that compiles to native bytecode is the FFPF Packet Language [CdBB05]. For this language, the reference system currently has x86 and Intel IXP network processor compilation targets [BdB C ′04]. Applications have to be modified to use it, because it is not language compatible with BPF.
3.5 Summary

This section presented the programming model of Streamline. Streamline reuses the proven I/O interfaces of Unix, including filepaths for system inspection and pipelines for application construction. This interface is clear, simple and well known. Moreover, the model supports introspection, a vital ingredient for application tailoring. Streamline exposes all pipelines in protected domains to all users by modeling them in the most common Unix namespace: the filepath. All kernel streams are exposed in the filesystem as Unix pipes. It refines the interfaces where necessary to support its specific goals of filter offloading to the best environment and data sharing across domains. Pipeline syntax is extended to express full directed graphs and conditional paths, to support such common application patterns as load balancing, protocol demultiplexing and filtering. On top of these base operators, Streamline further adds optional Boolean selection, looping and connection handling. To guide optimization, it enables filter and graph annotation. Streamline supplements these native interfaces with legacy socket and

Figure 3.5: Streamline architecture with interfaces
pcap support.

Figure 3.5 depicts the interfaces and their integration with the runtime system (the topic of the next two chapters). Besides PipesFS and extended pipelines, it shows the socket and pcap (and Sun RPC, not discussed) application interfaces and Unix I/O data access interface (Section 4.4.1). The figure shows how these interfaces give access to the computation spaces at all system layers and to their interconnecting Beltway buffer and control messaging systems.

Together, the ideas presented in this chapter contribute to the central goal of a high-throughput, flexible I/O system by enabling construction of composite tasks (a prerequisite for reconfiguration) for a wide range of practical streaming I/O tasks: a superset of Unix pipeline I/O.