Flows: A Novel Infrastructure for Communication in Distributed Systems

by

Andrew Kent Warfield

B.Sc., University of Western Ontario, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

April 2001

Abstract

We believe that the lack of advancement in the development of novel distributed systems is the direct result of a lack of necessary functionality to correctly describe and implement their communication requirements. Existing communication protocols, specifically the TCP/IP suite, cater strictly to static point-to-point data streams. The current state of the Internet clearly reflects the strengths and weaknesses of this model: Popular applications are almost universally structured as client-server.

The difficulties in realizing effective service location and client mobility are the consequence of a network abstraction in which only endpoints may be named and messages travel only from point to point. By naming individual data streams and allowing the network to resolve changing endpoint participation, these goals become very easy to address.

The existing communications infrastructure is the inevitable result of long-standing preconceptions of network and distributed system composition. The network is non-wholistically treated as a collection of disjoint endpoints. Messages are treated as second-class objects in an environment where only endpoints are named. Goals of transparency are implemented at the lowest possible point in the system through abstractions such as RPC [4] which, in an attempt to make procedure calls seem local, makes it impossible to publish distribution-related fault and control messages to applications.

The existing network infrastructure does not meet the needs of emerging distributed systems. For this reason, it is a relevant time to reconsider the desirable functionality of the network infrastructure.

This paper introduces the concept of a *communications flow*. The flow is in many ways an extension of previous work regarding data stream-centric communication [12] that has been augmented specifically to support the demands of large-scale distributed systems. A flow is a named entity that provides a handle on the network resources associated with a data stream in the same manner that a process ID associates local resources with a computational job [19].

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express sincere thanks to Norm Hutchinson for his advice and support with this thesis. Norm has been approachable and encouraging throughout this work, and has endured my frequent rantings with good humor.

Mike Feeley also deserves thanks as the second reader on this work, his suggestions helped to clear up several sections. Thanks also to Andrea Bunt, Bruno Godin, and Yvonne Coady for their discussion and ideas about flows and, of course, their proof-reading. Thanks also to Bruno for wrestling the VNC source into working over the flow middleware.

Finally, I would like to thank Alexander Fraser at AT&T research for planting the seeds of this thesis last year. Dr. Fraser has made many helpful suggestions along the course of this work. The idea of flow recursion is his.

ANDREW KENT WARFIELD

The University of British Columbia
April 2001

For my parents.

Chapter 1

Introduction

At the heart of all software systems lies the task of information movement. Data is generated at some point within a system and then moved to some number of other points, possibly being modified along the way.

Recently, many large-scale projects have begun to build very large, Internet-scale distributed systems. These projects aim to provide world wide access to distributed file systems [13], media delivery [22], and so-called ubiquitous [17], invisible [11], or pervasive [20] computing environments. In addition to these research systems, industrial projects such as Microsoft's '.net' [1] [6] framework attempt to allow the provision of distributed services across heterogeneous devices at the Internet scale.

With these ambitious projects, the mechanisms of information sharing are becoming more important than ever. Unfortunately, the architecture of the underlying communications infrastructure is not evolving as quickly as the demands of this new class of distributed application. Frequently, desired functions such as quality of service, group communication, and mobility [2] must be inefficiently provided, as they are not supported within the network.

[1] Pronounced 'dot net'.

[2] to name only a few...

The TCP/IP protocol suite has remained essentially unchanged since its inception, and has managed to scale well beyond all expectations. However, TCP/IP is unable to efficiently support applications that are not client-server in nature. Attempts to develop applications with a collaborative group structure result in excessive resource consumption, poor scalability, and difficulties in handling failure. This is very problematic, because group communications are exactly what is required of this emerging class of application.

This paper presents the communication flow, a stream-centric model for distributed communications. The flow is a named stream of communication that provides many properties crucial to the development of very large, finely distributed systems.

Our intention is that the flow model be considered as a network protocol that would operate in parallel with TCP/IP. However, as the deployment of a new network protocol is an unrealistic initial approach, we present a prototype of our model as a middleware layer above TCP/IP.

Chapter 2

Related Work

As flows are intended to act as a universal communications abstraction for distributed systems at the Internet-scale, they fit between several broad areas of existing work. This section presents summaries of pertinent work in each of these areas.

2.1 Stream-centric Communication Models

Uniquely naming communication streams provides many advantages: Named streams may be referenced to optimize data movement across a system. Names may be used as references to data across application domains. The following systems exemplify these properties.

2.1.1 Named Pipes

Named pipes were implemented very early in UNIX. The mechanism allows a named file handle to be created and accessed as a FIFO message queue. Any application on a single host could connect to the queue and send or receive messages [1]. This mechanism seems to be the first point in operating systems development in which it was possible to name a

[1] Note that UNIX named pipes still do not have any notion of support for group communications. Their behaviour with more than one sender or receiver is undefined and may be erratic.

specific communications resource, independent of endpoints. This decoupling provides a new degree of flexibility to concurrent applications.

2.1.2 Plan 9

Plan 9 [28] is an operating system developed at AT&T Bell Labs. The system carries the notion of named pipes further by treating every resource in the system as a file. All communication channels, including TCP streams, appear as files in the local file system. This is similar to the perspective presented by flows, in that streams are individually named at each local host. It is different though, in that the stream names have no relevance beyond the local host; they do not identify an end-to-end collection of resources who are participating in the stream.

2.1.3 Scout

Scout [12] is a communications-oriented operating system that uses *paths*, which are optimized routes for data across the system. For instance, a Scout system is able to set up a path for MPEG video that moves data from the network interface to the MPEG decoder and then to the display as quickly as possible.

Scout paths exist primarily as an optimization mechanism for data transfer, although they do represent a fundamental change in the way that this transfer is represented within system code. Path objects are created in the system and bound to by modules who participate in a particular data flow; this changes the perspective with which programmers must approach individual modules within the system.

For the most part, we see many of the properties of scout paths as a complimentary mechanism to flows. Indeed, flows address a fundamental problem in Scout paths, which is the need to use a packet classifier in order to determine which stream inbound packets

belong to. There are several aspects of Scout paths that would benefit from a reconsideration in order to provide a more general purpose system. First of all, paths are unicast and tend to be created within a highly local scope. We feel that a great degree of extensibility might be afforded by providing multicast support and a means of sharing PathIDs within the system. Additionally, it may be beneficial to provide paths [2] that are expressly willing to accept sup-optimal performance. An example of this might be a module that gathers statistics on a given path and is willing to accept a loss of messages, under high load, in order to preserve performance across the rest of the system.

2.2 IP Multicast

IP Multicast is an extension to the Internet Protocol (IP) to allow multicast transmission of IP packets. Many commercial routers currently provide some degree of support for multicast routing. The multicast protocols have evolved slowly over the life of the Internet, having been embodied by an overlay network called the MBONE. In recent years, there has been a strong drive towards providing integrated multicast support throughout the Internet and many RFCs have been put forward arguing the merits of varying approaches to routing traffic. IP Multicast will inevitably bring considerable benefit to distributed systems, who will be able to drastically improve their use of the network.

Despite the obvious benefits of IP multicast, concerns exist regarding its security, performance, and scalability. Routers must be trusted to forward packets appropriately, and almost all aspects of security are left to overlying applications. Due to the fact that routing is handled in a completely decentralized manner, join and leave latencies are significant. Finally, the name space reserved for multicast streams presents administrative difficulties and may also not be large enough to support the global community.

[2] or sub-paths...

2.3 Publish and Subscribe Event Architectures

During the past decade, much work has been done in demonstrating the usefulness of publish and subscribe event infrastructures to large distributed systems. One of the guiding observations of this research has been in the acknowledgement that publish and subscribe event systems are an extremely useful infrastructure for building large distributed systems [26, 16] but are difficult to scale effectively [5]. The publish and subscribe model is powerful because it provides a named handle on a conversation between any number of distributed parties. As messages are published to and received from the named conversation, endpoints are not as highly coupled and the system may easily be extended.

From a conceptual perspective, there is very little difference between issuing a subscription in a publish and subscribe system and joining a multicast communications channel. Both mechanisms decouple sender and receiver, while still allowing system-wide many-to-many message passing. In practice, the only real difference between these approaches is that existing network multicast techniques do not allow the same degree of message specification as typed event systems.

One intention of flows is to forge a middle ground between these two solutions. Group messaging systems clearly benefit from network level multicast and message filtering, however, network abstractions must provide structures that make them map appropriately to this sort of system.

A very large number of publish and subscribe systems have been developed and are in use today in commercial distributed systems. Several of these systems are presented briefly here.

2.3.1 The Information Bus

The Information Bus [26] was developed as a commercial distributed system infrastructure in the early nineties. The system provided publish and subscribe style distribution to applications requiring zero down time and upgradability. Sample applications cited by the authors are stock floor systems and integrated circuit manufacturing plant systems.

The architecture was built atop TCP/IP, and used specialized servers to handle message queuing. Ethernet broadcast was used as a optimization for group communication in local subnets.

2.3.2 Gryphon

Gryphon [16] is a departure from traditional subject-based publish and subscribe systems. In Gryphon, subscribers issue subscriptions as tuples, which describe the specific content that they would like to receive. These tuples are pushed down into the network and aggregated to form message routing filters at each routing node.

The advantages to this approach can be seen in an example of a file sharing system. Individual nodes may issue sets of subscriptions describing the files that they are currently sharing. Query messages may be published within the network, and will be routed to only the nodes with matching files.

2.4 Remote Invocation and Middleware

TCP/IP provides a single data stream between two hosts. In order to add functionality above this simple abstraction, distributed systems typically provide a mechanism to remotely invoke or pass messages to applications on other hosts. This section describes several existing mechanisms for this, and attempts to identify how each model expresses failure and distri-

bution to overlying applications.

The difference that we hope to identify in this section is the capability that flows provide in enabling the extension of these approaches to handle faults. More on this later.

2.4.1 Remote Procedure Calls

The motivating idea behind remote procedure calls [4] is that the invocation of code on a remote host can be most easily represented if it is syntactically identical to a local invocation. The notion of RPC has existed since at least the late seventies, and has been an influential principle in the design of distributed systems ever since. In RPC, additional code is added behind the scenes, and may even be generated automatically, to package parameters and ship instructions across the network. The abstraction adds a tremendous amount of simplicity to application code, but at a cost: remote invocation behaves in a considerably different manner than local invocation. Latency is an issue, as invocation time in an optimized system is still typically several orders of magnitude longer when calling remotely. Moreover, errors cannot be expressed to applications through anything but the return value of a local call. This makes it very difficult to identify and resolve, from the application, problems with the network or remote host.

2.4.2 Distributed COM

DCOM is Microsoft's approach to remote invocation. Microsoft acknowledges that RPC masks some errors due to the local-seeming syntax, and compensates by defining a result field that is capable of representing a broader range of errors. The name of this result type is HRESULT, and non-distributed COM methods also return values of the same type. So, in attempting to broaden the scope of fault representation, Microsoft imposes the distributed error framework on local invocations. Still, this approach forces an open set of errors to

be represented within a single return value, which is a numeric field referring to a set of constants in an errors header file. As such, this approach allows only a slightly greater amount of expressiveness than the original RPC.

2.4.3 CORBA

The Common Object Request Broker Architecture (CORBA) [27] is a standard for distributed object middleware. The intention of CORBA is to overcome issues that stem from the heterogeneity of distributed systems by building a standardized overlying layer. The original CORBA specification did not address issues relating to fault tolerance. Due to the increasing demand for reliable commercial distributed systems, a new standard has been finalized as of early 2000 for Fault-Tolerant CORBA [10].

A great deal of effort has gone into the design of the FT CORBA specification. Indeed, it is certainly the case that a much higher degree of response to fault is embodied by the standard. However, the specification is very complex and the potential for faults to exist as a result of this approach are real. This fact is demonstrated in [33], which identifies potential problems in the interaction of FT Corba with legacy Corba components. Moreover, for the most part CORBA attempts to provide distribution transparently to applications; although faults are handled much more appropriately within the Corba middleware, applications do not necessarily have the opportunity to address them at all.

Chapter 3

The Flow Architecture

A flow is a uniquely named message stream within the network. Flows exist independently of specific endpoints and provide multicast, allowing any number of senders and receivers. Flows provide IP-like best-effort message transport with no guarantees on delivery, ordering, or flow control;these guarantees are left to overlying implementations.

This section presents the driving design ideas behind flows. Where possible, we avoid mentioning specific implementation details, which are presented in the Prototype chapter.

3.1 Naming

Flows are named by globally unique 128-bit FlowIDs. These IDs are composed of three components: A creator ID (64 bits), a location service ID (32 bits), and a local ID (32 bits). The goal of this naming structure is to provide a simple means of creating and locating

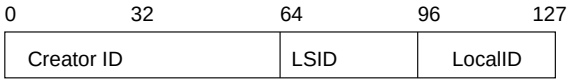


Figure 3.1: FlowID Composition

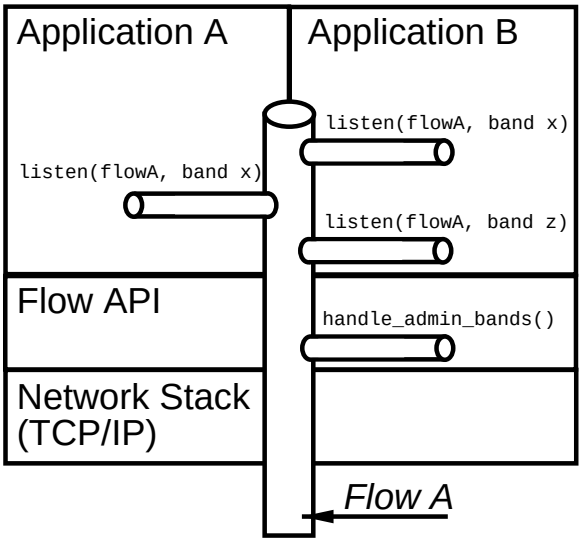


Figure 3.2: Bands allow a flow to cross layers

flows in a large distributed environment.

The creator ID represents the point within the network at which a flow was created. The purpose of this field is to divide the naming domain of flows across the network so that there is no need to test for naming conflict beyond the scope of the current host. Each host is responsible for the administration of the IDs within the 2^{32} entry local ID space, which includes avoiding ID conflicts across system reboots.

The location service ID (LSID) identifies a service that is responsible for maintaining the multicast routing of the flow. LSID's are mapped to full location service flowIDs through a lookup. Every router on the network must be configured with access to at least one location service. Location services, which will be expanded upon later in this section, form the administrative domains for routing within a flow-based network.

3.2 Banded Messages

Layering software involves a division of the system according to horizontal slices. Each layer exports an interface to be used within the layer above it, and in turn accesses the interfase of the layer below it. Operating system architects discovered very early that a layered approach to system structure provided many benefits [7]. Common libraries could easily be reused and user-level applications could be protected from one another. Layering has also proven useful in the development of data networks, the OSI specifies a seven-layer universal model [34] for network protocols against which all popular protocols may be mapped to some degree.

Although layers facilitate the architecture of systems in many ways, they impose interfaces that greatly constrain vertical information flow. In distributed systems, one of the greatest penalties that results from a layered architecture is the inability to express fault information appropriately. RPC hides the complexities of remote invocation by making them appear as simple procedure calls, a "well-known and well-understood mechanism for transfer of control and data within a program running on a single computer." [4] The downfall of this approach is that by making calls appear local, RPC forces a much broader realm of errors to be handled within the same local scope. This leads to difficulties in describing and responding to faults appropriately. The shortcomings of a strongly imposed layering are not limited to fault handling. Layering also effects system flexibility and extensibility by providing generalized, but non-universal interfaces [31]. Additionally, performance may be lost due to the overhead of procedure calls and data copying across layers.

These weaknesses of layering are not unknown to researchers. Several operating systems have been developed [9, 24] that attempt to minimize interfaces between applications and raw devices, providing only protection and multiplexing of interfaces to monolithic overlying applications. This approach, however, represents an opposite extreme: the

weaknesses of layering are eliminated at the cost of the benefits. These systems provide limited opportunities for horizontal integration between concurrent tasks, making system-wide services, such as disk and memory management, difficult to provide.

The Scout operating system [12] makes a significant contribution by recognizing the benefit of a generalized model for *paths.* By understanding where streams of data are generated and must be delivered within the OS, Scout provides the ability to optimize the transmission by providing the fastest path for the data to be delivered through the operating system. The authors of Scout term this optimization a *vertical integration* of the data path.

Flows include a mechanism, called banding, within the message structure that allows a vertical integration of communication streams within a system. Banding allows the contents of a data stream to be labelled, and allows the network and endpoint systems to filter for specific bands within a stream. Where Scout takes advantage of a vertical integration to optimize data transfer across modules, flows allow messages to pass across all layers of a system, potentially interacting with any of them, in an attempt to provide flexibility and extensibility. This is a considerable departure from the limited expressiveness and end-to-end design of TCP/IP.

In TCP/IP, data streams are one-dimensional pipes. It is an established practice to send typed messages within a stream, and recent feature-rich routers provide the ability to eavesdrop on packet payloads in order to make routing decisions (e.g., load balanced routing according to HTTP requests). This is an expensive operation within routers, as reading and re-addressing TCP data is complicated. Flows solve this problem by allowing data streams to be subdivided into *bands.* The flow header includes a field that allows messages to optionally be assigned to one of 128 bands within a data stream.

Bands allow a separation of concerns within the stream. The first thirty-two bands are reserved for administration and error-reporting. For instance, a flow's multicast routing

tree is described within the zero band. By separating multicast tree membership from application traffic, we allow routers to be aware of a flow and easily allow connected hosts to join without incurring the overhead of the entire stream's traffic. Moreover, bands allow routing elements to easily identify and handle administrative messages while simply routing normal traffic.

Band membership is currently represented throughout our system as a 128-bit mask. Routing nodes associate a band mask with each port, and push band subscriptions down into the network to reduce unnecessary traffic.

3.2.1 Band Filtering

Flow multicasts are organized as core-based trees [2] with a dynamically configurable multicast core. The use of bands allow a given flow to provide a range of content on a single shared multicast tree. Endpoints use bands to describe the specific content that they are interested in receiving and filters that describe these bands are pushed down into the network.

All routers in a flow track the upstream routing path for that flow. The upstream path is the route towards the flow core. All flow messages must be delivered to the core in order to ensure that they have the potential to reach all interested nodes, so the mask on the upstream path never performs any filtering. Downstream ports are where traffic in the network is filtered to reflect client interests. When a client registers interest in a specific band, a message is sent towards the flow's multicast core, resulting in a modification in the downstream band masks from the core to that endpoint. At each router, the downstream band masks reflect an aggregate of all interests below that port. All messages posted to a flow travel to the core directly, but are only percolated to participants in the shared tree who are interested in receiving them.

3.2.2 Multicast Tree Band

The multicast tree is maintained by a single band within each flow, band zero. Routers and endpoints may join a flow on this band in order to become a member of the shared tree for a flow, but will not receive any traffic beyond what is required in order to maintain the multicast tree itself. This approach allows tree membership and reconfiguration to be carried out in the absence of actual traffic. It also allows traffic subscription within a flow to be updated very effectively, as multicast routes have already been established for the shared tree.

3.2.3 Differentiated Services

Endpoints are responsible for local routing of messages once they are received from a flow. Many applications within a given endpoint may be interested in receiving messages from a flow, additionally individual bands may need to be demultiplexed and delivered to different points in an application.

Once an endpoint has subscribed to a flow, applications may add receive queues that filter for specific bands. The benefit of this approach is that the properties and behaviour of each queue may be specified separately, allowing for a differentiation on how inbound messages are delivered. In the current implementation, this differentiation is fairly limited, allowing the size of the inbound queue to be configured in order to avoid overrun and dropped messages. In a complete system, it is foreseeable to specify drop strategies and partial ordering to individual bands and push these properties down into the network as well. This is a significant benefit of the banded approach, because messages within a channel may be catered to differently. Costly operations, such as message ordering and delivery strategies, may be provided by an extensible set of endpoint data types and affect only the necessary traffic within a flow.

3.3 Locality

The ability to multicast is very important to allow a collection of hosts to efficiently communicate as a group. However for some applications, such as resource discovery, communicating with an entire multicast group may present far too much overhead to provide a sufficient solution. Additionally, in many cases it may be desireable to communicate with other endpoints who are 'local', according to some deffinition, to the current host. To address this, flows provide a message delivery option called *locality*.

Trying to find a resource in a peer environment involves communicating with other hosts on the network in order to find a desired resource that is available, and has an acceptable level of performance. This is difficult because of the potential need to contact a large group of connected hosts in order to find the desired resource. The brute force approach to this problem is to broadcast search requests to all participants. This approach is unappealing due to the large amount of traffic and correspondingly high processing requirements. The file sharing application Gnutella [1] has been shown to demonstrate the problems of scaling this approach within a unicast network. The use of multicast allows a partial solution to the problem by greatly reducing the volume of traffic generated by attempting to simulate multicast in an overlay, however multicast alone still requires that messages be delivered to all participants in a given multicast group.

Locality provides a further refinement of message delivery within a multicast channel by providing TTL-like limitations on multicast distance. Flow messages may specify a locality type and value as a delivery parameter. As a flow message is routed, routers apply a locality modifier to the locality value of the message. When a message's locality reaches zero, it is dropped.

There are many possible ways of considering locality, as the 'nearness' of two network endpoints can be expressed in any number of ways. A set of locality types are sup-

ported to describe different notions of locality when routing messages. Example locality types include bandwidth, delay, and geographic area. Bandwidth and delay modifiers are dynamic, and may be automatically tuned at flow routers to reflect current network conditions between neighboring routers. Geographic locality is hand configured to reflect the physical configuration of the network. For example, in our lab a geographic locality of zero represents the local application, one represents the local host, two represents the lab room, and three represents this floor in the building. Geographic locality allows a flexible representation of the physical network that has proven to be extremely useful in locating devices such as printers appropriately. Within a given locality, applications may use expanding ring-searches to locate resources at iteratively more remote areas of the network.

In addition, scopes of locality that represent the local host allow flows to be optimized for use as a fast message passing mechanism between applications with exactly the same interface as remote calls.

A similar notion to locality, known as TTL scope, exists in Internet drafts relating to IP addressing and multicast [21, 25, 32]. Scope allows the IP TTL field to determine how far multicast messages should be sent and defines a small number of geographic divisions within the 8-bit TTL range. As the number of hops between network endpoints is not necessarily indicative of geographic distance or expected performance, TTL is not a universal solution to expressing ‘nearness’ within the network. Locality provides a mechanism with which different measures of nearness may be described, and messages routed accordingly. Note however that the existence of locality does not displace the real goal of the TTL field, which is ensure that messages are not routed endlessly within the network.

3.4 Administration

This section describes operations required within the infrastructure to provide administration of flows. Hosts must be able to join and leave flow multicast trees, routers must maintain flow tables and garbage collect inactive flows. Routers must also be able to reconstruct flow routing tables in order to recover from failure.

Creating and Destroying Flows

The notions of flow creation and destruction exist strictly as conveniences to endpoint operating systems. As flows are named on a host-specific basis, there is no potential naming conflict that must be resolved within the network in order to instantiate a new instance. As such, the create and destroy functions exist only to manage flow-related data structures within the local operating system.

Joining and Leaving Flows

A host wishing to receive messages from a flow must join the flow's multicast tree. This join is an asynchronous operation during which the client sends a join request and expects to be attached to the flow or receive an error message shortly afterwards. The client join message has the following format:

```
flow_join(flowID, bandmask)
```

This command generates a source-routed message to the membership band of the flow describing the join request. A membership band host on the flow may respond to the join request message with a join approve message, which cascades back to the client extending the flow's multicast tree.

Hosts may also explicitly leave flows by issuing the flow_leave command:

```
flow_leave(flowID, bandmask)
```

This command results in the generation of a leave message on the membership band of the flow. This message is largely for the benefit of applications wishing to track membership. The leave command will also cause a routing update message to be sent from the host, indicating that it no longer wishes to receive the specified messages.

Note that join and leave messages both specify a bandmask as well as a flowID. Bands provide a flexible means of optimizing message flow within a shared multicast tree: All members of a flow participate in the multicast band, which is a band reserved specifically to mark the global multicast tree for a specific flow. The bandmasks that individual hosts specify may be pushed down along the multicast tree, allowing flow messages to be filtered at optimal points within the multicast tree.

Garbage Collecting Routing Tables

As flows are not explicitly created and destroyed at endpoints or through a central routing system, routers must garbage collect flow routing tables to remove entries for inactive flows. We choose this approach because we believe that it provides a scalable solution to maintaining routing tables without necessitating any sort of central administration.

In addition to the fields that describe participating ports and bandmasks, routers also maintain a field that marks the last epoch during which a message was seen on each port that a flow is routed on. This is a small integer, that is incremented to reflect the router's current epoch whenever a message is received from that flow on that port. The epoch is incremented periodically, and after a threshold number of epochs, a keep-alive ping is send to the flow on the port. After a second threshold has passed, the port is deleted from the flow's routing entry. Once all of the ports have been deleted, the flow's entry is removed.

This approach allows routers to maintain their own tables through a background task. Furthermore, the amount of traffic generated in order to maintain flows is very small.

3.5 Fault Expression and Handling

One of the primary benefits of banding is to provide a means to publish fault information within the context of a distributed system. Specific bands may be chosen to carry fault messages pertaining to some aspect of the distributed system, and anyone receiving the flow may choose to listen to messages on these bands.

Using flows in this way presents a fundamental difference to network communications. In TCP/IP, it is expected that only the endpoint applications will add and remove messages on the stream. The model can be considered as two queues, connected by a transport system. The transport system includes everything below the application, including middleware, the network stack, the operating system, the network interface and any devices, such as routers, that lie in the path between the two communicating applications.

There are two weaknesses with respect to fault handling in the TCP/IP model. First, the transport system is very complex. Many things can potentially go wrong in the transmission of messages, but this system has no means of interacting with the message stream to report or respond to problems. Second, the message buffers at either end of the stream have a very limited capacity to handle 'out of band' data – data that relates to the stream, but is not part of the expected application-specific protocol. TCP/IP provides two flags, the urgent bit (URG) and the push bit (PSH), to expedite data delivery within an active stream. The urgent bit is used to alert the receiver that a specific region of the incoming stream contains urgent data, and the push bit is used to indicate that individual packets should not be delivered as quickly as possible without buffering. Note that according to the TCP RFC [29], urgent data is not necessarily delivered out of band – it simply provides a means to alert the receiver of special incoming packets. Many flavors of UNIX extend the interpretation of the urgent bit, calling it TCP_OOB (out of band) data. In these systems, urgent data is stored separately from the receive buffer and may be read immediately by clients. These

systems additionally provide a signal, SIGURG, which may be used as an up call to alert applications of the arrival of urgent data. This interpretation of the urgent bit is not by any means universal across existing operating systems; the Microsoft Windows TCP/IP stack obeys the push bit, but takes no special action whatsoever with relation to urgent data.

By supporting bands, flows are able to address these problems. Bands allow fault messages to be associated with, but kept separate from, the message stream. This means that messages may be inserted within a flow at any device or layer within the transport system without worrying about conflicts with application-specific protocols. Provided that fault messages exist on their own band, they essentially represent a completely separate stream of communications. Fault messages may be generated on flows from anywhere within the system, and selectively received anywhere else. This allows applications to see deep within the system, if they so desire, and take action in response to faults that might otherwise need to be resolved at lower layers in order to maintain transparency. By no means do applications have to deal with low level messages; existing approaches to systems continue to be applicable with flows. However, in cases where applications would like to deal specifically with system messages, flows provide the ability to do so.

In addition to this, flows allow message delivery to be demultiplexed across a set of queues at each endpoint. An application that is written for TCP may be moved to flows by mapping its TCP accesses to a specific band within a flow. The receive queue will deliver only messages from the remote application. However, the application may then be extended to handle faults by adding additional handlers and queues to respond to messages on other bands. These bands may publish application-specific faults, or may contain faults generated within the transport system, link errors for instance.

As an example, a video streaming application may use an extra band to advertise overflow messages back to the server. The client may be moved to flows, and then have

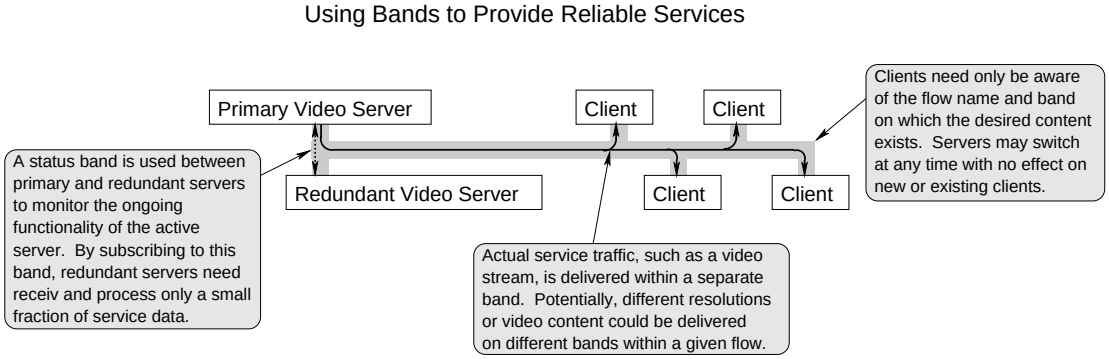


Figure 3.3: Using Bands to Provide Reliable Services

additional code added to generate fault messages in the case that its receive buffer should overflow or underflow. These messages would be sent back to the server, where a handler could tune the application to reduce the delivery rate of the stream. In addition, network midpoints that supported flows could publish messages to this band in the case that they too were experiencing buffer overflow.

A second example of using bands to survive faults is shown in Figure 3.3. The figure shows how redundant servers may be provided on a flow by using a separate band to monitor the state of the active server. In the figure, a primary server multicasts a video stream to a collection of clients. Meanwhile, on a separate band, this server communicates state information with a redundant backup server. Through this band, the backup server can be kept aware of the current position in the video stream and the active server's well-being without the overhead of receiving the entire video stream. Should the backup server receive a shutdown notification or timeout on this band, it will take over the responsibility of multicasting the stream data. If this timeout and switch can be executed more quickly than the clients exhaust their receive buffers, then service may continue uninterrupted, with the clients completely unaware of the server reconfiguration.

3.6 Location and Routing

In order for flows to be usable in a global scope, issues regarding their location and routing must be resolved. These are difficult problems, and the subject of our ongoing research. In this section we present a partial solution to the routing of flows. The shortcomings of our solution are outlined as pointers towards what areas need to be addressed.

The Internet is currently plagued with problems relating to difficulties in routing. Primary among these is the fact that the network core is unable to cope efficiently with the volume of traffic that is being sent. In addition to this, it is extremely difficult to provide differentiated services and guarantee a specific quality of service to any given connection. There are many reasons for these problems: for example, the naming of endpoints on the net is very disorganized and makes message forwarding difficult. Also, individual streams cannot be easily distinguished leading to difficulty in differentiating service. The recent development of the Multi-Protocol Label Switching Architecture (MPLS) [30] addresses many of these concerns within the network core. MPLS is not, however, an end-to-end connection representation.

By naming specific streams, flows provide a convenient means of differentiating service. However, the increased name space of flows greatly compounds issues in routing. Moreover, as individual flows are not bound to specific network endpoints, their location is more complex than that of IP.

We address the issue of flow location through the use of *Location Services* (LSs). Location services are collections of hosts, residing on a single flow, who manage the multicast routing of a collection of flows. In many ways, LSs are analogous to the cores of core based multicast trees (CBTs) [2]. A flow's location service is identified through the location field within the flowID. A flow is bound to a given location service for its entire life time. Location services solve many problems. First, they act as a point of administration for the

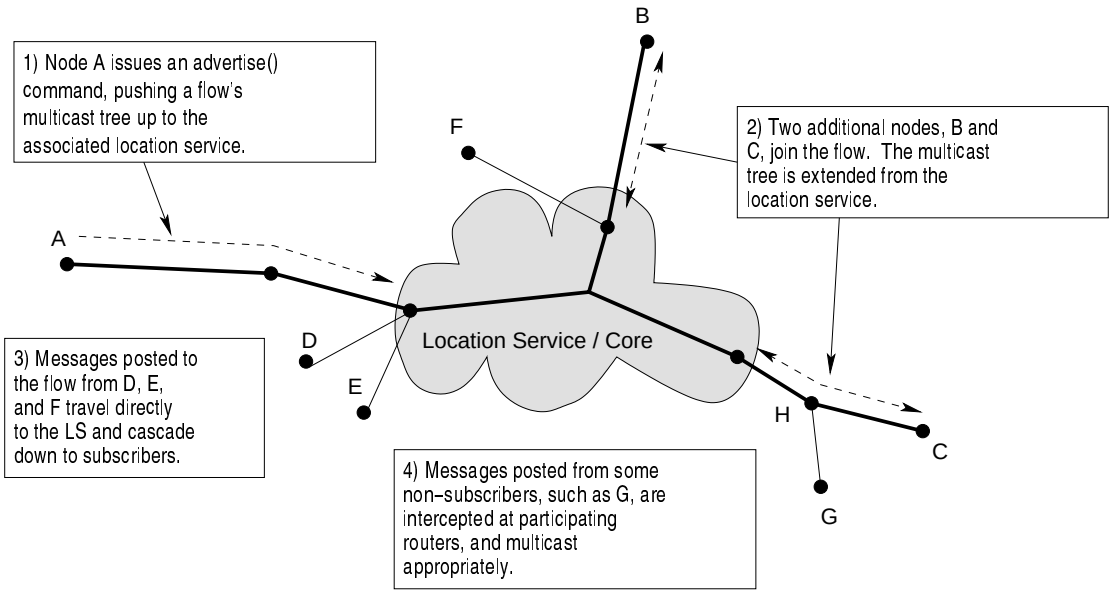


Figure 3.4: Multicast routing with flows

provision of flows. One of the few assumptions made of the network is that a given location service is able to find any other location service. As such, a user need only be able to contact a single LS in order to gain access to the global network. The second benefit of the location service is to act as a network midpoint in order to locate flows. Although flows are labelled with their originator's unique ID, the originator is not bound to participate in them forever. The location service can be counted on as an available entity within the network through which a flow can be located.

In order to send a message to an unknown flow, the flow must first be *advertised* to its associated LS. The advertisement of a flow involves pushing a zero-band subscription of a flow towards its associated location service. The advertisement establishes a link between the endpoint participating in the flow, and the location service to which the flow will belong. Once advertised, any endpoint on the network can send a message to the flow.

Sending a message from a node that has subscribed to a flow is very simple. When

a flow is advertised, the upstream path to the location service is configured to accept all messages on all bands. The downstream paths are configured dynamically to reflect the band subscriptions of the leaf nodes below them. Band flags are turned on as endpoints request subscriptions, and pruned using the garbage collection mechanisms described above. This approach ensures that all messages will reach the core of the multicast tree, and then be distributed appropriately according to band subscriptions across the network. Moreover, the location service may also use bands to filter messages passed across the multicast core. As each participant of the LS is aware of the subscriptions of the hosts below it, it may advertise that mask to the other core routers and reduce traffic within the core accordingly.

If a message is sent to a flow from a host that is not a subscriber, routing is only slightly different. Intermediate routers, that do not have routing information specific to that flow, forward the message towards the location service. In the case that the message arrives at a router on the flow's multicast tree prior to reaching the LS, it is routed normally. Otherwise it is routed downwards on all paths from the core. This is identical to the forwarding approach proposed in the core-based tree [2] strategy.

The use of location services provides a benefit beyond the CBT mechanism in that it addresses the provision of a multicast core. As LS IDs name specific cores within the network, while still decoupling those cores from specific endpoints, location services may adaptively reconfigure to provide optimal routing within the network. A full exploration of the exact function of the location service, at a global scope, is beyond the realm of this thesis. There are issues that need resolution, such as the actual mechanisms in reconfiguring LS membership, and the partition of heavily loaded LSs, which we have not yet addressed. Our prototype, described in the next section, implements simple, hand configured LS participant groups and does not address the task of dynamic reconfiguration.

A final shortcoming of our approach lies in the potential size of routing tables within

the network core. As mentioned earlier, flows compound the degree to which IP already suffers from this issue within the core. We discuss this issue briefly in the future work section later.

Chapter 4

Prototype

We have designed and built a prototype implementation of flows. The prototype is a middleware library that is used in conjunction with a software router to form an overlay network above TCP/IP. The prototype has been developed in C and has been written for use within Linux.

This section describes the design and implementation issues of the prototype. The client library and router are presented separately, followed by an explanation of the flow message structures.

4.1 Client Library

The prototype middleware provides a client library with which applications may use flows. Applications include the library and initialize it to connect to an active flow server. Once connected, clients communicate using only flowIDs, completely independent of endpoint locations. The API instantiates a thread to attach to the flow router and handle inbound messages. The details of the client API and the message queues are provided in the remainder of this section.

Table 4.1: **Flow API - Core Functions**

Function	Description
flow_create(flowid)	Create a new flow with the specified ID.
flow_advertise(flowid, localtiy)	Advertise this flow.
flow_addlistener(flowid, bandmask)	Add a listener to the specified flow.
flow_removelistener(flowid)	Remove a listener from the specified flow.
flow_get(flowid)	Get a message from a flow message queue.
flow_get_blocked(flowid)	Get a message from a flow message queue.
flow_send(flowid)	Send a message to the specified flow.

4.1.1 Client API

The interface to the flow library from application code is intended to be very simple. Flows are administered from the local host using the create and advertise functions. Message queues are attached to flows using the add and remove listener functions. Finally, a set of message send and receive functions are provided. Consider each of these functions individually:

`flow_create(flowid)` - Create a flow with the specified flowID. This function registers a routing entry for the flow in the local flow table. It also ensures that the flow has a valid ID by checking the creator ID and the location ID, and ensuring the local ID does not conflict with any existing flows.

`flow_advertise(flowid, locality)` - Advertise this flow beyond the local host. This function forces an extension of the flow multicast tree (band zero subscription) towards the location service by a distance specified by the locality. In most practical cases, we imagine that flows would be advertised all the way to the location service by using a global locality value. However, this does provide a mechanism for flows to be advertised only within the local area. In retrospect, it may prove to be a better approach to always advertise all the way to a location service, and to provide separate location services for the administration of local flows.

`flow_addlistener(flowid, bandmask)` - Attach a message queue to the specified bands on the specified flow. This function instantiates and returns a pointer to a message queue. The queue is registered in the local flow table to receive inbound messages matching the flowID and bandmask provided. Additionally, the local subscription bandmask is aggregated to accommodate the new subscription. If new bands are added to the flow membership, a message is passed to the router to change the band mask there.

If this is the first listener added to the specified flow, the library will issue a join request message to the flow, negotiating the extension of the flow's multicast tree to the local host.

`flow_removelistener(flow_queue)` - Detach the specified message queue and delete it. The local flow table is modified to reflect the deletion and the multicast message paths will be garbage collected to make appropriate changes on their own.

`flow_get(flow_queue)` - Asynchronously remove the next message from the flow queue. If no message is available, null is returned.

`flow_get_blocked(flow_queue)` - Synchronously remove the next available message from the specified queue. If no message is currently available, block until one is.

`flow_send(flowid, msg)` - Send a message to the specified flow. The client need not be a member of the specified flow to send a message to it, all messages are simply routed towards the associated location service (flow core) for transmission along the flow. Currently, authentication mechanisms that would protect clients from receiving messages that do not belong within the flow are left to applications. It remains to be

explored exactly how much security and authentication can productively be included within the flow network.

4.1.2 Linking to the Flow Overlay

The only location dependence in the flow prototype is the bootstrap problem of connecting to a flow router. Client applications must use the `flow_connect(server, port)` command to connect the local host to a specified flow router. Ideally, the client should connect to a router with low load and high performance. Unfortunately, this is a difficult problem to resolve.

As we have been assuming an implementation of flows that will model a network-layer system, the location of a router is not a huge problem. In a flow-enabled network, a host would simply send messages to its next-hop router. Unfortunately, there is considerably more complexity in addressing this problem in an overlay: There is no easy way to pick an optimal router from a large collection without a high performance overhead. Furthermore, router performance may change over time and there may be benefit, in the case of an overlay, to migrating the uplink to an alternate router. Migrating uplinks would require modifying multicast trees across the network and, if not performed carefully, could result in an implosion of administrative load across the system.

A final issue here involves surviving failure. If the uplink router crashes, the client is dropped from the flow network. It is possible to compensate for this problem partially by providing the host with information about other routers within the network that may be used for fail-over. Unfortunately, failing over to a second router could involve issuing requests to join multicast trees and result in a partial traffic loss. Solutions to these problems remain to be explored.

4.1.3 Receive Queue

When clients use the add_listener command to subscribe to a flow, a circular receive queue is created in user memory. This queue may be arbitrarily sized by the client. As messages arrive at the local listen thread, they are matched against the local listener list and delivered to the appropriate client queues.

In our prototype, this does not represent an optimal message delivery mechanism as the data is copied three times as it is passed through the system to the client. First, the kernel moves the data from the receive buffer of the network interface to kernel memory for processing. Once processed, the network stack copies the message into user memory, where it is received by the flow library. Finally, the flow library delivers a copy of the message to each listener queue.

The goal of this work has not been to achieve best case message delivery within the local host. A large amount of research has already been done in this area [3, 8, 12], and the goal of this prototype has been to demonstrate the general functionality of flows. However, it is easy to imagine how an incorporation of the flow library into the kernel network stack could eliminate one of these copies, making flow message passing similar in local overhead to TCP. A kernel incorporation would have the further benefit of demultiplexing and delivering messages to all applications on the local system, instead of requiring an individual router connection for each. Finally, the implementation of a copy-on-write mechanism would allow efficient delivery of flow messages to a collection of local listeners, eliminating the need for extraneous copying.

The intention of using circular receive queues in this implementation is to maintain message recency. As flows do not guarantee delivery, we felt that it made more sense to allow applications access to the most recent window of inbound messages as opposed to filling the buffer and just dropping new arrivals. This approach is not universal, and definitely

achieves worse buffer performance than dropping any messages that cannot be accommodated. Future work could explore providing a more expansive set of buffer primitives that would allow applications to clearly specify the message delivery behaviour that they desire.

Buffers provide an interesting example of how bands can be used to express faults. In the current implementation, an error band is used only to express receive queue overflow. When a buffer overflows and an existing buffered message is overwritten, an overflow notification message is delivered to this band. This allows a handler to be provided to respond to the buffer overflow, perhaps by resizing the buffer or performing application-layer flow control with the remote end. Of course, caution must be taken in responding to an overflowed state by generating additional messages. For this reason, overflow messages are generated very infrequently. After a notification has been sent, the buffer waits for a large number of inbound messages to pass (currently 2000), prior to generating another notification. Additionally, overflow messages are never generated from buffers who are subscribing to the flow stack message band, as this would be counter-productive.

An interesting property of the distributed benefits of flows is shown in this case. As overflow messages are generated on their own band, it is foreseeable for a server to listen for remote buffer overflow messages, achieving a basic end-to-end flow control feedback mechanism. Unfortunately, this approach runs a risk of generating a large amount of extraneous multicast traffic. Currently, the flow stack band is not forwarded beyond the local host and serves only as a local administrative mechanism. The flow control approach just described could be explicitly achieved by copying overflows to a separate, forwarded band.

4.2 Flow Router

The flow middleware uses an overlay network of software routers to deliver messages between flow participants. This section describes the design and implementation of a proto-

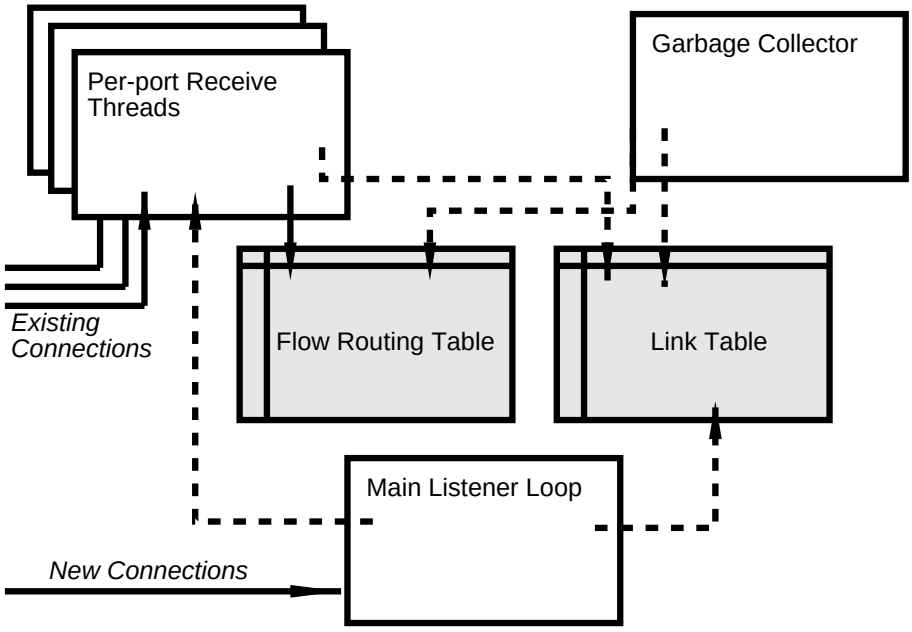


Figure 4.1: Structure of the Flow Router

type router to accept and deliver flow messages.

4.2.1 Router Architecture

The architecture of the flow router is illustrated in Figure 4.1. The router is composed of three types of thread: a main listener loop to accept inbound connections, a garbage collector, and a set of threads to handle messages for individual inbound ports.

4.2.2 The Link Table

The router simulates a physical link environment by maintaining a table that describes data links to other routers and endpoints within the overlay network. Each of these links is a TCP stream along which flow messages may be sent. The link table is akin to a port list, and describes all of the router's current connections.

Each link is handled by a listening thread. The thread receives inbound TCP mes-

Table 4.2: **Flow Router - Link Table**

LinkID	sock	linkType	linkState	locDeltas	flows	remoteHost
1001	4	L_ROUTER	L_UP	{x, y, ...}	{A, C, ...}	{...}
1002	5	L_ROUTER	L_DOWN	{p, q, ...}	{C}	{...}
1002	5	L_ENDPOINT	L_UP	{s, t, ...}	{A, C}	{...}

sages, which it buffers and breaks into flow messages. The flow messages are either routed immediately according to the flow table, or processed locally. The routing algorithm will be described in more detail later in this section.

The link table maintains a collection of information for each connection. The structure of the link table is shown in Figure 4.2. Each entry is described briefly here.

Link ID - This is an arbitrary identifier that uniquely describes the link. This identifier is negotiated at connect time between the two participants, and is used in conjunction with the remote host ID to reestablish a lost connection.

TCP Socket (sock) - This is a handle on the socket for this connection. TCP sockets are mutexed for write and all threads write directly to each other's outbound ports. It remains to be seen if this is an acceptable strategy to manage routing under load. A better option may be to associate outbound message queues with each link, but this would mean an increase in complexity within the router code.

Link Type - The two acceptable values for this field are `L_ROUTER` and `L_ENDPOINT`. The value is used to differentiate routing behaviour and reconnection mechanisms between the two types of node.

Link State - This field has three acceptable values, `L_UP`, `L_DOWN`, and `L_DELETED`. A link that is marked as up is connected to a remote host and is actively routing messages. A down link does not currently have a connection, but will be able to

resume routing if a connection is established. Finally, a deleted link is flagged for garbage collection.

Locality Modifiers (locDeltas) - This field represents an array of locality modifiers for each link. Locality modifiers currently reflect geographic area, latency, and bandwidth. In the prototype, these fields are assigned manually. Additional work to the router could add the functionality to tune these values dynamically.

Dependent Flows (flows) - This list maintains references to entries in the flow route table that are routed on the link. In the case of link error, this list may be used to generate error messages to the affected flows. Additionally, links may not be deleted in the current prototype until all hosted flows have been removed.

Remote Host - The remote host field contains host and port information that may be used to reconnect in the case of a lost connection. This structure also stores a host ID that is used to prevent concurrent flow clients on a single host from interfering with one another.

4.2.3 The Flow Route Table

The flow routing table is a simple structure, used by all messaging threads within the router to forward messages appropriately. The table is designed to allow a route to be looked up quickly and to allow forwarding decisions about a message on that route to be made as easily as possible. The structure of the table is shown in Figure 4.3, and its fields are described here.

FlowID - This is the 128-bit ID of a flow. Each flow that is being routed by the current router has an entry in the table. A lookup function is used to retrieve a specific entry from the table.

Table 4.3: **Flow Router - Routing Table**

FlowID	Aggregate Bandmask	Link List
Flow A	{...00100111}	{(LinkA, {...11111111}), (LinkB, {...})}
Flow B	{...11101011}	{(LinkA, {...11111111}), (LinkC, {...}), ...}
Flow C	{...11111111}	{(LinkB, {...11111111})}

Aggregate Bandmask - This field stores the aggregate of all downstream subscription bandmasks. By aggregating this value across all ports, the router is able to do a single comparison on messages in order to drop unneeded messages immediately. This also facilitates replies to subscription tests during garbage collection, as upstream replies only need to be tested against this single field.

Link List - This is a list of all links currently participating in the flow. The upstream link is always listed first, followed by each other participant. Associated with each link is a bandmask that describes the bands that should be forwarded to that link. The upstream bandmask is always set to forward all messages.

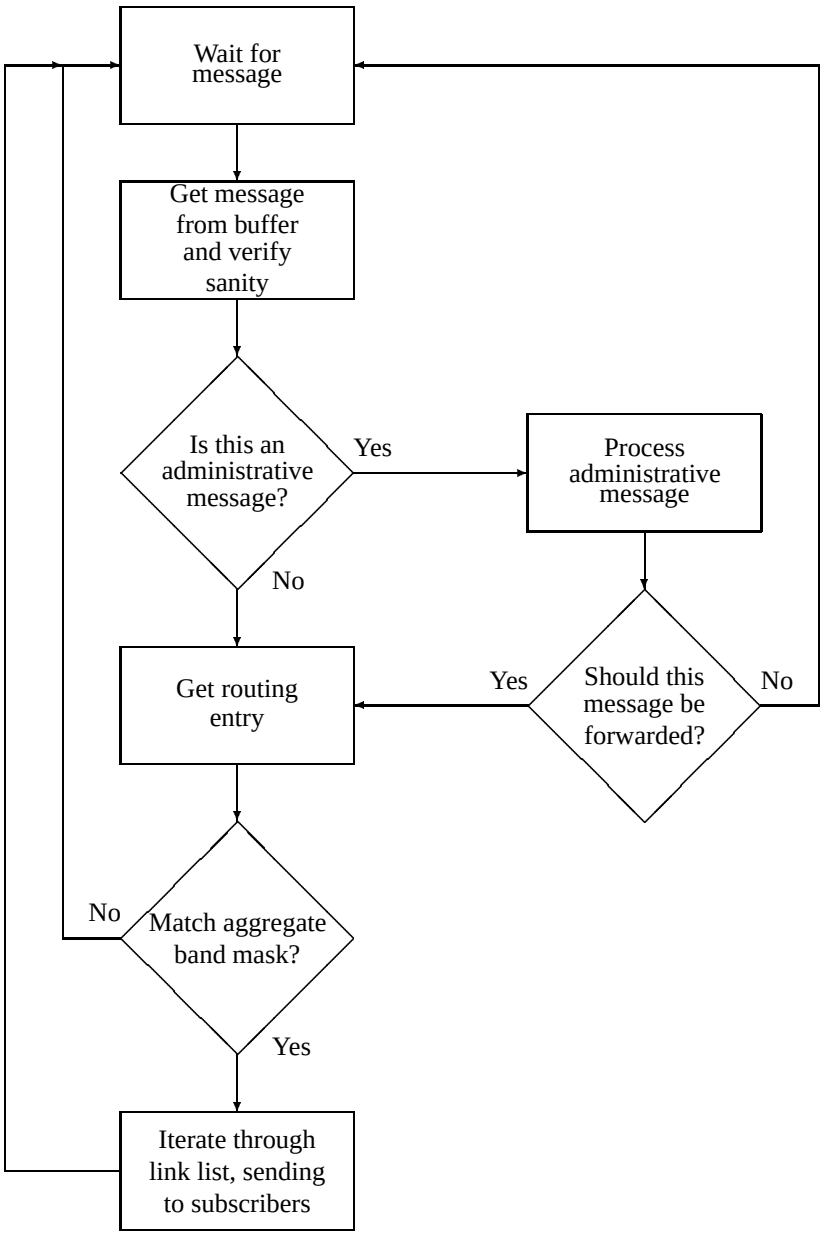
4.2.4 The Location Service Table

The location service table is used as a second-level routing lookup for messages sent to flows that are not listed in the routing table. This table equates 32-bit Location ID fields from FlowIDs to specific entries in the link table. In the current prototype, all active location services must be registered within this table. Future work might involve allowing wild card entries to specify default routes.

4.2.5 How Messages are Routed

As TCP messages are received by threads attached to an active link, they are placed in a per-thread receive buffer. The thread parses this buffer, removing flow messages and

Figure 4.2: Prototype Flow Router – Routing Mechanism



performing a cursory verification on the sanity of the message structure.

Messages are removed individually from the receive buffer and tested for administrative content. Administrative messages may be flagged in one of two ways: (1) They may have a band value lower than 16, or (2) they may have a Location ID value below the reserved location range. The low sixteen bands are used to pass messages that may be relevant to flow routers. In our prototype, these messages include multicast tree route updates, garbage collection, and link errors. The reserved location IDs are not currently used, but are left to provide support for special services, for instance routing and network management.

Administrative messages are passed to specific handlers and may either be dropped, or returned for forwarding.

To forward a message, its destination flowID is looked up in the flow routing table. The message's band is tested against the aggregate band for the routing entry, and if there is no match, the message is forwarded only to the upstream port. Note that a message is never forwarded on the arrival port, so messages from the core that do not match the aggregate band mask are dropped immediately.

If a message does match the aggregate band mask, the router will iterate through the list of links in the flow routing table and forward to all links with matching masks. As messages are forwarded, their TTL is decremented by one, and their locality is modified by the value specified in the link table.

4.2.6 Garbage Collector

All endpoint initiated operations in the network of flows move towards creating new message delivery paths. Flows are never explicitly deleted, nor are multicast trees pruned or band filters explicitly narrowed. The motivation for this approach is that operations from clients should move the network towards a desired state of message delivery, but that clients

can not be trusted to clean up after themselves. As such, network endpoints are in a steady state of requesting and using flow services, while the network itself attempts to remove services which are not being used. Clients need only be aware of the flows that they are involved in and want to remain available.

The garbage collector in our prototype operates on a single message band. Garbage collection involves broadcasting membership pings along each flow that has not transmitted traffic for a period of time. The strategy uses the notion of epochs, described in the architecture chapter.

Parallel to the flow table is a garbage collection table, which is associated with each link registered for each flow. The garbage collection entries list two epoch values in association with each link. The first value represents the last epoch during which traffic was received from that link for the current flow. The second value indicates the last epoch during which a ping was sent along the flow on that link.

As traffic is received from a flow, the garbage collection values are updated to reflect activity. The collector thread wanders the table and generates ping messages to inactive flows. If no reply is received to the ping message, the garbage collector will generate a delete message that is sent down the inactive link and remove that link from the flow routing entry.

Note that only endpoints generate replies to garbage collection messages. This allows routers to share the use of these messages, by updating collection table entries to reflect pings generated elsewhere on the network. Garbage collection pings are only sent and forwarded to downstream entries in the flow routing table.

As an additional optimization to reducing unnecessary traffic in the network, a mechanism has been included to request and advertise per-flow band masks to next-hop routers. These messages are not forwarded, but may result in a cascade of updates. If a

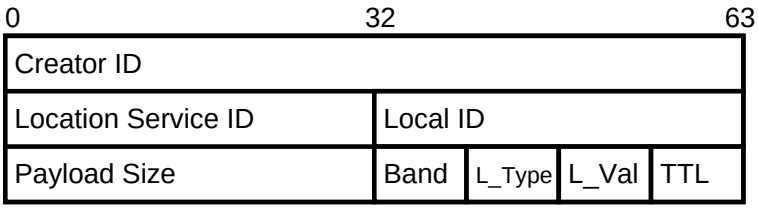


Figure 4.3: Prototype Flow Message Header

client or router realizes that it is receiving messages that do not match its forwarding band masks, it may advertise the actual band mask to upstream nodes. This allows a client to immediately reduce inbound traffic as subscriptions are dropped.

4.3 Message Structures

Flow messages in the prototype are prefixed with a 192-bit message header, shown in Figure 4.3. The first 128 bits of this field contain the flowID. The three components of this ID were described earlier in this thesis. The remaining 64 bits of the flow header provide additional messaging information. Each of these fields is discussed briefly here.

Payload Size - The size, in bytes, of the message payload that follows this header.

Band - The band on which this message has been published.

Locality Type (L_Type) - This field indicates what type of locality to use, if any, while routing this message. Currently defined values include `LOC_BANDWIDTH`, `LOC_DELAY`, `LOC_GEOGRAPHY`, and `LOC_NONE`.

Locality Value (L_Val) - The locality value of this message.

Time to Live (TTL) - A flow-level TTL field was implemented to ensure that messages would terminate in the prototype overlay network. Every routing node decrements

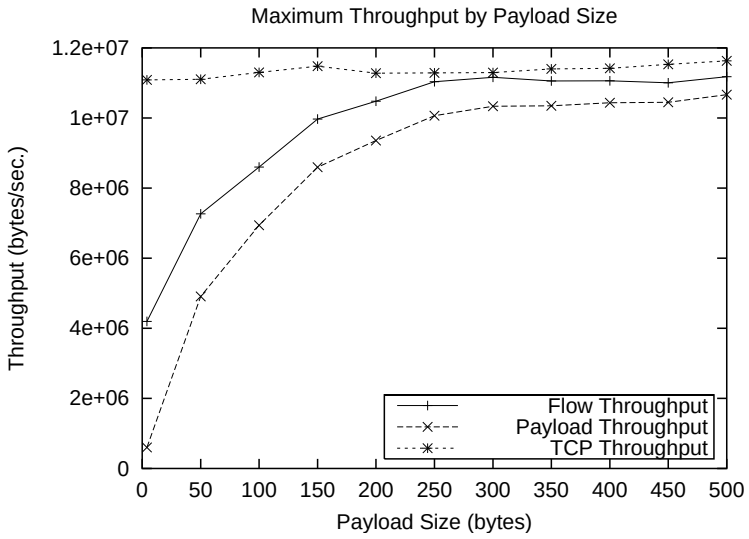


Figure 4.4: Flow Throughput by Payload Size

this value by one each time the message is forwarded. The message is dropped when its TTL reaches zero.

4.4 Performance

This section presents the results of performance tests on the implemented flow router and libraries on a three-node system. The tests describe the efficiency with which the flow middleware is capable of delivering messages.

Each host used for these tests is a 450MHz Pentium III with 128 megabytes of RAM and a 100 megabit Ethernet interface (Intel 82557). Each host runs Linux 2.2.16. The tests are intended to explore the overhead that the existing implementation of flows, written completely at the application layer, represents above raw TCP/IP transport.

In order to test throughput, an application was written to generate traffic on a specific flow. This traffic was forwarded to the router and then on to an application on a receiving node, which verified that messages had arrived intact and calculated throughput

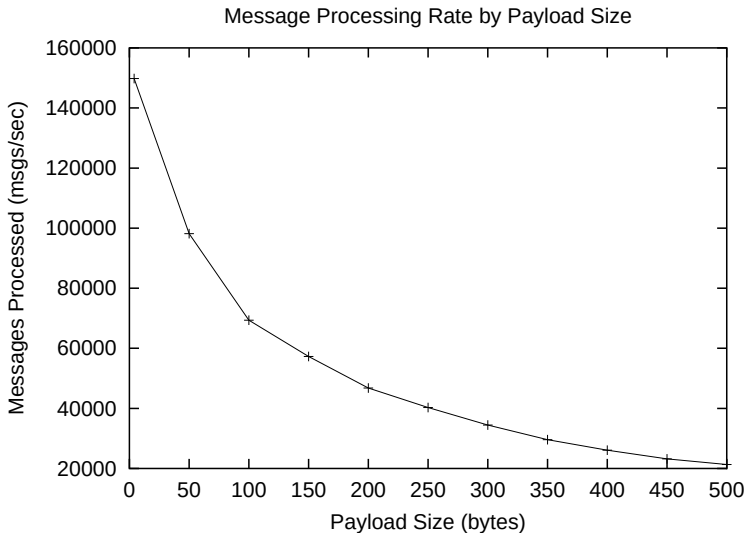


Figure 4.5: Message Processing Rate by Payload Size

statistics. The results of this test are shown in Figure 4.4. Flow throughputs are compared to maximum TCP throughputs, calculated using netperf [23]. The TCP throughputs reflect throughput tests of TCP packets with payload sizes of the specified flow payload, plus the 192-bit flow header.

As can be seen from these results, flow throughput converges with TCP/IP at a payload size of approximately 250 bytes. At this point, the computational overhead of message routing does not inhibit delivery rate. This is further exemplified in Figure 4.5, which shows the rate at which the flow router processes messages as message size increases. This rate is initially limited by the routing node, but then decreases and stabilizes as the network interface becomes saturated.

In addition to these results, we have tested flow latency overhead by comparing round-trip time (RTT) between flow messages and TCP/IP pings. On average, we found that our implementation incurs a 100 μs end-to-end latency on round-trip message delivery.

Chapter 5

Application Examples

The prototype implementation of flows has been used to develop two sample applications that demonstrate the benefits of the flow model. The first application is a general lookup service that is used to provide names for flows within the network and store information about the location services. The second example is a variable-resolution multicast video streaming system.

5.1 Flow Directory Service

The fact that flows have very long numerical identifiers and the need to track location services have led to the development of a simple directory service for use in the network.

The flow directory service is currently a single flow that many network endpoints participate in. Nodes may subscribe to the service and receive requests for directory information. All information takes the form of tuples, no further structural specifications are imposed.

Requests to the service take the form of tuples containing wild cards, and list a flowID to which replies should be forwarded. For instance, a request to the directory service

to find the location of a flow called VNC-1, and to have the reply forwarded to flow A would take the form `((''FLOWID'', ''VNC-1'', *), FlowA)`. This request could be sent to the directory service and all hosts with matching entries would send replies to flow A.

This approach allows locality to be used to determine how far search requests should be sent. Locality may be used to perform an expanding ring search within the directory service, iteratively reaching larger groups of endpoints.

Future work on the directory will need to address issues of scale. It will not be acceptable for every host on the network to be able to flood this peer-based service with requests. A suggested approach to this will be to use linked clusters of hosts, and institute forwarding heuristics at cross-domain links to eliminate messages from poorly-behaved hosts.

5.2 Video Over Flows

A clear benefit to flows that has been articulated repeatedly throughout this document are the potential benefits to delivering distributed services within a large heterogeneous network. We have modified the VNC package, developed by AT&T Cambridge, to use our prototype middleware instead of raw TCP/IP for delivery. By using flows, we are easily able to extend the functionality of VNC to provide multicast delivery, variable service, and mobility.

VNC allows the display of a computer to be forwarded to a remote host on the network. This remote host is able to interact with the machine as if it were local, all keyboard and mouse interactions are passed back to the VNC server and applied locally. VNC allows any remote host to act as a thin client for the server machine.

5.2.1 The VNC Flow Protocol

In moving VNC to flows, we have split the protocol to allow it to take advantage of banding. Two bands are used to advertise and request specific screen resolutions. New clients may connect to an active VNC flow and subscribe only to the resolution advertisement band. This band periodically broadcasts a list of available resolutions and the associated bands on which those resolutions are displayed. If the current resolutions do not satisfy the client, a new resolution may be requested by posting a message to the request band.

To join the session, a client need only subscribe to one of the active streams at the desired resolution. This will result in that band being forwarded to them along the existing multicast tree.

This division also results in a useful reorganization of the VNC server code. Accesses to the communications stream may be spread across the software to the appropriate places. An advertiser thread spins, publishing a list of the currently active broadcast threads. Each of the broadcast threads is identical, except for parameters describing resolution, and the band to send data on. Finally, a separate thread handles inbound requests and instantiates new broadcast threads whenever necessary. A very simple reorganization of the code, and the use of flows provides both multicast and variable service.

Mobility is also achieved for free via this approach. We are using VNC strictly as a delivery application and have disabled remote mouse and keyboard interactions. As the application is stateless, mobile clients may simply rejoin the flow from new locations in order to continue to receive the same stream. In more advanced applications, a mechanism would be required to move the stream across two locations and synchronize, this would need to be provided at the application.

5.2.2 Future work with VNC

The integration of flows with VNC has lead to many interesting ideas for ongoing work. It would be very interesting to explore the option of extending VNC to provide a collaborative work environment for a set of participants. We imagine extending the system to allow all participants to forward mouse and keyboard interactions to the remote host, possibly interacting with some sort of overlay on the remote host. Each user could have a separate mouse pointers and interact with different, or the same window all concurrently. The implications for collaborative environments using this strategy are very exciting.

Chapter 6

Open Problems and Future Work

We feel that the current prototype has demonstrated the usefulness of the properties that flows provide to communications in a distributed environment. There are, however, many unresolved problems that have come to light during the work to date. There are also several interesting future directions that could be explored as an extension of this work. These topics are discussed in this section.

6.1 Security

In a large distributed environment, security represents a very hard problem. One of the major reasons for this is that in the case of wide distribution, very few assumptions may be made about the trustworthiness of resources, or even the network itself. In the case of our prototype, overlay network nodes may potentially be scattered across the Internet, and could potentially be compromised. Some specific concerns in the case of a communications infrastructure involve eavesdropping (sniffing), impersonation (spoofing), and malicious attacks.

The flow model described here is vulnerable to all three of these problems. In our

prototype, any host may join a given flow and subscribe to receive messages on it. Any host may send a message to a flow, and the model described in this paper does not explicitly require that messages include a source address. Finally, a denial of service (DoS) attack would be quite easy to carry out within the prototype, and would likely have the initial result of overwhelming the routers, bringing the network down. It is worth noting that IP multicast is susceptible to these same problems to a similar degree as flows.

Solutions to these problems are especially difficult to solve within the domain of a communications infrastructure for two reasons. First, routing needs to be easy. Routers form an inevitable bottleneck within communications systems, and addressing security by moving computationally intensive tasks, such as key validation, into the network is probably not a good idea. Secondly, distributed system access control is almost universally based on some sort of key scheme. In order for an infrastructure to remain useful over a long period of time, it must not commit to security mechanisms that could potentially become weak. For this reason, we feel that security within this sort of infrastructure should be largely left to overlying applications.

Unfortunately, leaving all assurances recarding security to applications is insufficient. In the case of data privacy, it seems fair to leave applications responsible for incorporating the appropriate degree of encryption. In this sense, we side with the end to end argument [1] [31]. However, in order for a communications infrastructure to be successful in an environment such as the Internet, data privacy is not the only concern. Denial of service has proven to be a substantial issue in recent years, and to date this is only within a unicast network. The risks of a denial of service attack within a large multicast tree are much more substantial. It is our opinion that the network needs to be capable of somehow ensuring some fundamental properties of message passing and access control to the end nodes, even

[1] Although this argument may fundamentally disagree with the flow abstraction to begin with.

if the authentication and decision mechanisms are implemented at a higher level. Based on these observations, we suggest some ideas towards a model of using capabilities to provide access control for flows. The remainder of this section briefly discusses this model first by identifying the aspects of flows for which access control may be required, and second by outlining how capabilities might present a useful solution.

6.1.1 Limiting Access

The obvious aspects of communications within flows to which it would be beneficial to control access are posting messages and subscribing to receive messages. Beyond these properties though, it may be desirable to provide a finer grained degree of access control. For instance subscription and publication limitations might be beneficial at the individual band granularity. Also, in order to prevent the generation of huge amounts of traffic it may be useful to limit the maximum locality value that may be placed on a message.

6.1.2 Capabilities

We feel that, used properly, capabilities present an excellent solution for the management of flows. If administrative decisions about flow management, such as approving join requests, are left to higher level (above the router) applications, a flexible capability scheme may be used. Moreover, as the scheme is implemented outside the flow protocol itself, capability mechanisms may evolve over time, ensuring that the network retains the ability to provide good access control. Finally, as location services can potentially be implemented as very powerful distributed clusters of hosts, we expect that a reasonable performance can be achieved in response to capability-enabled administrative requests.

In order to implement this model, a capability would be associated with a flow at the time of creation. As creation occurs locally, this could be carried out with no security

concerns. Next, a key exchange would need to be negotiated between the creator and the location service at the time of advertisement. Finally, this exchange would have to be extended to allow key exchange with additional administrative hosts. An intelligent approach to this would likely be to continue with our efforts to decouple aspects of the system; much in the same way that the directory service is provided above flows, a service to authenticate and exchange flow capabilities could be implemented within the network.

6.1.3 Flow Name Space Management

By allowing endpoints to each own an explicit section of the overall 128-bit flow name space, we avoid any overhead that might otherwise be involved in requesting names from a centralized service. There remains an unresolved flaw in this approach that became apparent during implementation. If an endpoint creates a flow and then unsubscribes from it while other endpoints are still connected, the client must avoid using that ID for newly created flows. This presents an additional security consideration, as it must be clear within the system that an endpoint does not necessarily have administrative control over all flows that are labelled with its unique endpoint ID.

In a capability enhanced model of flows, a solution to this problem might be to allow clients to request a list of all active flows within their name space from location services. Alternatively, clients might be able to test for conflicts at creation or advertisement. Capabilities should allow some assistance in solving this problem, as they should prevent two unlike flows with the same name from being inadvertently joined together.

6.2 Performance and Scalability

As alluded to at several earlier points in this paper, there are some concerns as to how this definition of flows will behave at a global scale. The two most prevalent concerns here are

the behaviour of core routers in forwarding huge numbers of active flows and the ability to efficiently garbage collect the resources of huge numbers of short lived flows.

6.2.1 Routing Flows at an Internet Scale

A significant concern in the existing structure of the network lies in the fact that routing tables within the network core have become extremely large. The speed with which existing routers can process and forward packets is considerably slower than the throughput available from the transport medium. Flows, as presented here, make this problem worse due to the fact that all traffic is potentially multicast and the range of flows(2^{128}) far exceeds the range of IP addresses (2^{32}). As each active host could potentially be a member of many flows at once, the size of core routing tables would inevitably become very large.

Ipsilon networks proposed a solution [14, 15] to the IP routing table problem that has more recently been incorporated along with other approaches and embodied by Multiprotocol Label Switching (MPLS) [30]. We feel that these approaches to traffic engineering are well matched to flows. Their solution involves resolving paths across the network backbone at edge routers, allowing packets to be labelled with switching instructions to form a virtual circuit across the network backbone. This approach offloads routing resolution and table management away from the core to ingress points, where traffic is considerably lighter.

The flow model presented here could be extended to allow individual flows to contain other flows. This would present a fantastic administrative benefit in that flow traffic could be routed hierarchically, by wrapping messages at the edges of the backbone and sending them across a small set of flows that traverse the edge points of the network core. Moreover, if a hierarchical implementation were efficient enough, it could be used to the exclusion of banding. This would allow a extensible separation of concerns within network

streams, while also providing individualized management and administration for each flow.

6.2.2 Garbage Collecting and Short Lived Flows

The garbage collection mechanism described in this paper functions well within a well-behaved network. As flows become unused, endpoints simply stop replying to pings from the garbage collector, and they are removed from routing tables.

One concern about this approach lies in the fact that a router could potentially receive a huge load of requests to create new flows. A single client could choose to advertise its entire address space of 2^{32} allowable flowIDs. Each flow that is actively routed commits router resources, specifically a single FRT entry, from the time it is created until it is garbage collected. This time is at least the sum of the two thresholds used for garbage collection. If one or more clients were to start issuing a very large number of advertisements, they would likely be able to overwhelm a routing table of any reasonable size. This form of denial of service attack is very similar to the TCP SYN flood [2], but would consume resources within the network instead of at endpoints, potentially compromising service for the user community as a whole.

A second concern with this situation is that garbage collecting a huge set of connections could potentially produce waves of high administrative traffic as routers scanned their tables and sent pings. In the worst case it is imaginable that these pings could cause enough congestion to interfere with other traffic, thus worsening the situation.

There are some ideas as to how to resolve parts of this problem. Routers could incorporate heuristics to block poorly behaved endpoints [18]. Also, mechanisms could

[2] A SYN flood involves sending a huge number of TCP connect requests to a server but not replying with acknowledgements to complete the TCP connection. The server is forced to maintain the state of all connect requests so that it can complete the connection set up, and this attack results in the server being unable to serve incoming connect requests. This specific for of attack is the primary denial of service tactic that has been seen against large servers on the Internet recently.

be used to generalize all traffic to a host who is participating in a very large number of flows, perhaps by sending garbage collection digests instead of large numbers of individual messages. Still, this problem definately warrants further examination.

Chapter 7

Conclusion

As endpoint devices become more powerful and interesting, and network connections between these devices become faster, the benefits of distributed systems become more obvious than ever. It seems clear that an emerging class of distributed system will involve the use of 'heavier' client applications, possibly evolving away from the model of a centralized server completely in some cases. For this class of system, existing communications abstractions, particularly those provided by TCP/IP, are insufficient to effectively provide the necessary services.

This thesis has presented a communications model for distributed systems that addresses these concerns. The model presented here is well suited to providing the type of event-driven structures being used by emerging distributed systems, while remaining in a form that could foreseeably be implemented as a network layer protocol for improved performance.

Having completed this implementation of the flow middleware, we are left with several observations regarding our initial architecture. Primary among these is the insight that the notion of recursive flows, mentioned earlier, is a desireable property and could largely supplant flow banding by providing a more versitile and extensible solution. The

second insight is that flows should support some sort of type descriptor, allowing each flow's content to be described. By providing these two properties, we feel that a network-layer implementation of flows would prove very useful as a universal connective abstraction to provide communication throughout distributed systems.

Bibliography

[1] Gnutella: To the bandwidth barrier and beyond. In *Clip2.com*, http://gnutellahosts.com/gnutella.html, November 2000.

[2] T. Ballardie and C. Trees. Core based trees (CBT). An architecture for scalable inter-domain multicast routing. In *SIGCOM '93*, September 1993.

[3] D. Banks and M. Prudence. A high-performance network architecture for a PA-RISC workstation. *IIEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)*, 11(2):191–202, 1993.

[4] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, 2(1):39–59, February 1984.

[5] A. Carzaniga, E. Di Nitto, D. Rosenblum, and A. Wolf. Issues in supporting event–based architectural styles. In *Proceedings of the 3th International Software Architecture Workshop*, November 1998.

[6] Microsoft Corporation. An introduction to Microsoft .net. 2000.

[7] E. Djikstra. The structure of the the multiprogramming system. 1968.

[8] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Symposium on Operating Systems Principles*, pages 189–202, 1993.

[9] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, December 1995.

[10] Ericsson, Eternal Systems, and et al. Fault tolerant CORBA, joint revised submission. Technical report, OMG TC Document orbos/99-12-19, Object Management Group, Framingham, MA, 1999.

[11] Mike Esler, Jeffrey Hightower, Tom Anderson, and Gaetano Borriello. Next century challenges: Data-centric networking for invisible computing. In *Mobile Computing and Networking*, pages 256–262, 1999.

[12] A.B. Montz et al. Scout: A communications oriented operating system. In *HotOS: Hot Topics in Operating Systems Workshop*, 1995.

[13] Johen Kubiatowicz et al. Oceanstore: An architecture for global-scale persistant storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[14] P. Newman et al. Ipsilon flow management protocol specification for IPv4. In *RFC 1953*, May 1996.

[15] P. Newman et al. Transmission of flow labelled IPv4 on ATM data links. In *RFC 1954*, May 1996.

[16] R. Strom et al. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering*, 1998.

[17] S. Gribble et al. The Ninja architecture for robust internet-scale systems and services, 2000.

[18] P. Ferguson and D. Senie. Defeating denial of service attacks which employ IP source address spoofing. In *RFC 2267*, January 1998.

[19] A. G. Fraser and G. E. Mapp. An interpretation of IPv6 for global service. Technical report, AT&T Research, 2001.

[20] Robert Grimm, Tom Anderson, Brian Bershad, and David Wetherall. A system architecture for pervasive computing. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.

[21] R. Hinden and S. Deering. IP version 6 addressing architecture. In *RFC 2373*, July 1998.

[22] J. Jannotti, D. Gifford, K. Johnson, M. F. Kaashoek, and J. O'Toole Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation.* USENIX, October 2000.

[23] R. Jones. Netperf home page. http://www.netperf.org.

[24] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[25] D. Meyer. Administratively scoped IP multicast. In *RFC 2365*, July 1998.

[26] B. Oki, M. Pfluegl, Alex Siegel, and Dale Skeen. The information bus: An architecture for extensible distributed systems. In *Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*, 1993.

[27] Object Management Group (OMG). The common object request broker architecture and specifications. Revision 2.3. Technical report, OMG Document formal/98-12-01, Object Management Group, Framingham, MA, 1998.

[28] R. Pike, D.L. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proceedings of USENIX*, 1990.

[29] J. Postel. Transmission control protocol. In *RFC 793*, September 1981.

[30] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. In *RFC 3031*, January 2001.

[31] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. In *ACM Trans. on Computer Systems (TOCS)*, November 1984.

[32] D. Thaler, M Handley, and D. Estrin. The internet multicast address allocation architecture. In *RFC 2908*, September 2000.

[33] S. Wilhelmi. Limitations of the proposed fault-tolerance extensions to CORBA. In *Proceedings of the International Conference on Dependable Systems*, 2000.

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