1988

FLOWS: Performance Guarantees in Best Effort Delivery Systems

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Rajendra Yavatkar

Report Number:
88-791

FLOWS: Performance Guarantees in Best Effort Delivery Systems[1]

Douglas E. Comer
Rajendra Yavatkar
Purdue University
West Lafayette, IN 47907

CSD-TR-791
July 1, 1988

Abstract

A best effort delivery system provides a connectionless, datagram service in which the packets may be delayed or dropped in the presence of congestion. The underlying network does not make any guarantees about performance to an application. Performance needs of various applications differ greatly and applications such as packet voice and video have strict performance requirements.

Existing network architectures and protocols do not make provisions for applications to specify their performance needs, and communication architectures do not have mechanisms that satisfy and guarantee performance. We introduce a network level communication abstraction called *flow*. A flow is a communication channel that has specific performance characteristics associated with its traffic. The underlying delivery system guarantees to meet performance requirements of a flow once it accepts a flow request. Upper layers use flows to implement transport level protocols and high performance applications.

This paper describes the concept of flows in detail and presents algorithms that implement flows in a high speed packet switched network under development at Purdue University.

[1]To appear in the proceedings of IEEE INFOCOM '89.

FLOWS: Performance Guarantees in Best Effort Delivery Systems *†

Douglas Comer and Rajendra Yavatkar

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

A best effort delivery system provides a connectionless, datagram service in which the packets may be delayed or dropped in the presence of congestion. The underlying network does not make any guarantees about performance to an application. Performance needs of various applications differ greatly and applications such as packet voice and video have strict performance requirements.

Existing network architectures and protocols do not make provisions for applications to specify their performance needs, and communication architectures do not have mechanisms that satisfy and guarantee performance. We introduce a network level communication abstraction called *flow*. A flow is a communication channel that has specific performance characteristics associated with its traffic. The underlying delivery system guarantees to meet performance requirements of a flow once it accepts a flow request. Upper layers use flows to implement transport level protocols and high performance applications.

This paper describes the concept of flows in detail and presents algorithms that implement flows in a high speed packet switched network under development at Purdue University.

1 Introduction

The advent of high-speed fiber optic technology and the emergence of performance intensive distributed applications have provided major impetus to research in computer communications. First, commercial availability of very high speed communications technologsie has led to the development of innovative packet switch architectures such as Fast Packet Switches and photonic switches[26,12,9]. Second, the use of networks is no longer limited to applications of remote login, file transfer, and electronic mail. New applications such as real-time voice, multimedia conferencing, and image networking have performance requirements that span a wide range. For example, packet voice demands low delay with strict limits on delay variance, image data transfer requires high throughput; and some applications require both low delay and high throughput.

Unfortunately, communication protocol and transport architectures used in conventional packet switched networks have not kept pace with these advances and, therefore, cannot cope with the demands made by both the underlying hardware and higher level applications. The challenge in designing future networks is to design protocols and algorithms that will offer transmission services that match user's delay, throughput, and reliability requirements.

As part of the Multiswitch project [8] at Purdue university, we are exploring a new network architecture to provide high performance communications in a wide area network. Our project is building a new, multiprocessor-based packet switch architecture as the cornerstone of a network with a gigabit per second capacity.

Traditionally, packet switched networks offer a connectionless, datagram service. To the upper layers, the lower layer offers a *best effort delivery* abstraction. A best-effort delivery system does not guarantee to deliver messages; messages may be duplicated, lost, or delivered out of order. Each packet switch has a finite buffer capacity and, in the presence of overloading conditions, it simply discards the packets it cannot handle. Also, datagram networks treat each packet as an independent entity and do not maintain any state information about the traffic generated by individual sources.

Higher level protocols use the datagram mechanism to offer higher-level abstractions such as reliable byte streams [19] and request-reply message transactions

*This research is supported in part by a David Ross fellowship and by Purdue University.

†To appear in the proceedings of IEEE INFOCOM '89.

[4]. Integrating digital voice or video with data in a packet switched network offers many benefits including better use of network resources [27]. However, supporting real–time applications such as packet voice [24,23], multimedia conferencing, and remote instrumentation [14] with a best effort delivery mechanism is difficult because such applications have strict delay constraints. Therefore, arbitrary packet losses and wide fluctuations in delays caused by congestion are unacceptable.

Existing network architectures make no provisions for specifying and satisfying the performance needs of an application. Under the datagram model, higher levels cannot specify the limits on delay, bandwidth, or error rates and the lowest layer has no way of providing such guarantees.

Virtual circuit based networks provide reliable data transmission and reserve buffer space (and/or bandwidth) for individual connection at intermediate nodes. However, reservation of buffer space at the packet switches does not imply guaranteed performance and there is no provision for specifying data rates or limits on delays. Both datagram and virtual circuit based networks are prone to congestion and the performance depends on the amount of traffic in the network.

Some of the existing network layer protocols such as IP [18], ISO [22], and SNA [15] define a *type of service* (TOS) selection that allows higher layers to specify the quality of service desired. However, the TOS specifications are in qualitative terms and, therefore, are not suitable for applications that demand quantitative performance guarantees. Also, network layer protocols provide no guarantees for supporting a TOS specification or achieving the performance desired.

To support high performance applications in a best effort delivery system, we provide an abstraction called a *flow*[1] at the network level. A flow is a communication channel between a source and a destination and has specific performance characteristics associated with its traffic. Upper layers treat a flow as an end–to–end abstraction and specify its performance characteristics in quantitative terms when they request a flow. At that time, the underlying delivery system verifies that it can guarantee the performance needs under any condition barring a *flow failure*. A flow is implemented using the best effort delivery and does not imply absolutely reliable delivery. Messages in a flow may occasionally be lost, but an application may specify upper bounds on acceptable error rates for a flow.

Another motivation for introducing flows is that we are investigating a resource reservation model for packet switched networks. Unlike virtual circuit networks where the emphasis is on error control and flow control based on buffer space reservation, we are interested in effectively managing the networks resources (that include processing power at a packet switch, capacity of communication lines, and buffer space at switches). Van Jacobsen's work indicates that window–based flow control leads to wide fluctuations in network delays and hence the performance. Our goal is to explore the impact of a resource reservation scheme that uses rate–based control within a network.

The remainder of this paper discusses the design and implementation of flows. Section 2 describes the concept of flows in detail. Section 3 discusses the implementation of flows. Section 4 gives examples of uses of flows and section 5 describes possible extensions and alternatives to the present design. Section 6 contains a description of related work and section 7 provides a summary of our work.

2 Flows

For the purpose of defining and discussing flows, we assume a datagram network model with two layers, called *link* and *network* in the OSI reference model [22]. A flow represents a simplex, end-to–end communication channel between two network level entities, a *sender* and a *receiver*. A sender sets up a flow that has specific performance characteristics associated with it. A link level *flow arbitrator* guarantees, barring a flow failure, to deliver packets in the flow with performance bounds specified at the time of flow creation.

The flow arbitrator reserves appropriate amount of network resources corresponding to the performance desired, but packet delivery is still best effort. Because we are interested in guaranteeing performance and reserving network resources for a flow, it may seem appropriate to use a connection oriented scheme such as virtual circuit for flow implementation. However, a virtual circuit–based scheme uses an elaborate hop–to–hop or entry–to–exit error and flow control. Such control is not necessary for all the applications and the processing overhead associated with error/flow control is not desirable for some applications. For example, considering the strict delay constraints that exist on the delivery of each packet in real–time applications, connectionless transport is more suitable than connection oriented transport because connection oriented transport implies higher delay and more processing time at each intermediate node. Furthermore, the major benefits of a virtual circuit

[1] The term *flow* was inspired by David Clark of MIT [5]

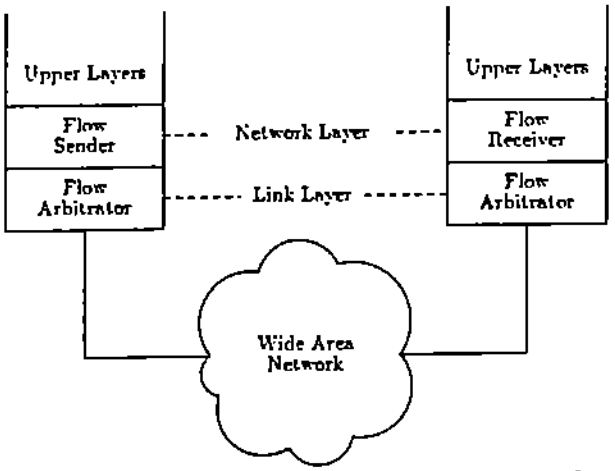

Figure 1: An abstract view of components of a flow

necessary to real-time traffic because retransmissions will probably cause violations of delay constraints.

Sender and receiver entities have host-specific network level identifiers and each flow also has a network-wide unique flow identifier called *flowid*. The flow arbitrator at the sender's side marks all the packets in a flow with its flow identifier. Figure 1 shows the model of flow at the network layer. All the details of creating and managing a flow as well as routing the flow traffic are handled at the link layer and are hidden from the sender and the receiver. When a flow fails, the flow arbitrator informs the sender and the receiver about the failure.

We designed the semantics of flows to provide maximum flexibility in their use at the upper layers and, therefore, we do not restrict the operations on flows to be strictly synchronous. To allow for synchronous or asynchronous behavior, we assume that a receiver may either be an active entity such as a process or a passive entity such as a port where packets are queued. In the following, we describe a set of flow operations available at the network level:

flowid ← setup(destination, parameters) The flow arbitrator checks to verify that it can select a path to the destination that satisfies the specified parameters, sets up a flow path, and returns a flow identifier.

send(flowid, message) Sender requests delivery of the message to the destination of a flow. This is an asynchronous operation in which the operation appears to complete before the transmission completes. Thus, the sender does receive neither any acknowledgement that the message is delivered, nor a notification if the message is lost.

terminate(flowid) Sender requests the termination of a flow when the flow is no longer needed,

failed(flowid) A flow arbitrator informs the sender when one of its flows fails. The arbitrator may inform a sender of a failure either through an explicit upcall [6] (i.e., by calling a procedure that the sender specifies) or by returning an error when a sender attempts a flow operation later.

Message ← receive(flowid) Sender receives a message from the flow arbitrator.

At the time of flow creation, a sender may specify some or all of the following parameters:

Delay bounds Delay for a message is measured as the interval between the time at which a sender passes the messages to its flow arbitrator and the time at which the message is delivered to the receiver. At the time of flow creation, a sender may specify an upper bound on the delay any individual message (of a given size) in the flow may experience without breaking the flow and may also specify the average delay expected by the flow traffic. When a flow arbitrator successfully sets up a flow, it guarantees those bounds for the flow traffic except in the case of a flow failure.

Throughput A sender may also specify the desired average throughput (in bytes/sec) for the flow. The value of this parameter may be specified in statistical terms, such as an expected value with an upper bound specifying the burstiness, or in deterministic terms.

Error rate Some applications are sensitive to the rate of packet losses and, therefore, a sender may specify the amount of packet loss it can tolerate. Because a flow arbitrator reserves the network resources along the flow path, the packet losses are mainly due to transmission errors. Therefore, we express the error rate as an average bit error rate. At the time of flow creation, a sender may specify an upper bound on the error rate that it can tolerate.

3 Implementation

This section describes an implementation of flows in a best effort delivery system. Our implementation scheme is based on the underlying Multiswitch network architecture [28]. The following principles dictated our choice of algorithms for implementation:

Simplicity The concept of flows is novel and we do not have enough experience with the performance-driven applications that will use them. Therefore, we want our implementation to be simple and easy to implement. As a result, whenever possible, we have discarded complex schemes in favor of simpler ones.

Efficiency Finding an optimal route is a difficult problem. Because we want to allow multiple flows between identical endpoints to use different paths depending on the flow specifications, we are interested in finding multiple paths to a destination. Given more than one metric to choose a path, computation of an optimal path that satisfies all the metrics is an expensive operation that does not scale well given the combinatorics involved. Therefore, whenever possible, we have chosen to use an efficient (computationally less expensive) algorithm or heuristics that may not always lead to an optimal path to a destination, but will find a path that meets the flow specifications.

Stability Currently, there are a variety of routing algorithms for long haul networks in use or proposed in the literature. Many of the adaptive, distributed algorithms that adapt to the network state dynamically suffer from the problem of instability as changes in network conditions cause the computed *best* paths to oscillate between two or more alternatives. Our goal is to use a routing algorithm that will lead to stable paths for a flow. Therefore, we chose a stable and more *static* scheme for routing a flow through the network.

3.1 Underlying Network Architecture

This section describes the Multiswitch network architecture under development at Purdue. We use this architecture to describe an implementation scheme for flows., but the concept of flows is equally applicable to other architectures that use circuit/path switching or to non store-and-forward switches like crossbar and Batcher–Banyan packet switches.

The Multiswitch network architecture consists of a set of store–and–forward packet switches interconnected using point–to–point links in any arbitrary configuration. Each packet switch or node in the network has enough processing capacity to keep all the network interfaces busy. In other words, a packet switch is not a bottleneck and a switch can route an incoming packet to an outgoing link with minimal switching delay. The network uses a Shortest Path First (SPF) routing algorithm similar to the one currently used in the ARPANET [16]. This algorithm uses the delay as a distance metric and the shortest path to a destination has minimum delay for packet delivery. Each node in the network knows the entire network topology, and nodes periodically exchange link status update information to keep the topological information up to date.

Each link status update contains the measured time delay for that link. The network treats the packets with control information such as link status updates separately from the packets that carry data. Before a packet enters the network, the source node assigns a priority level to that packet. The control packets have the highest priority and the data packets may have one of four additional priority levels. At each node, there is a separate FIFO queue for each level of priority. A node always handles a packet with higher priority first and queues up packets with lower priority until they can be processed. There is a finite bound on the length of the queue for each level of priority and a node discards packets when a queue gets full. Whenever packets are discarded, the congested node informs the source of the packets which in turn reduces the traffic it sends through that node.

3.2 Information Exchange

We assume that most of the traffic through the network will consist of datagrams that are not part of any flow and, thus, most of the traffic has no performance characteristics associated with it. For such traffic, we choose the shortest path based on an independent criterion. The routing algorithm used in the Multiswitch network is similar to the current ARPANET routing algorithm [16][2]. In the ARPANET algorithm, each node in the network measures the delay on each of its links. If the delay on the link exceeds a predetermined threshold, the node broadcasts an update on the link status to all the other nodes in the network. The threshold is time–driven and is reduced every 10 seconds, so that, even in the absence of change, a node broadcasts an update approximately every minute. When a node receives an update that may change its routing tree, it updates its routing table. Each node computes its routing table using a *shortest path first* algorithm [1]. The network topology information available at each node is consistent throughout the network and, therefore, the paths computed independently by each node are consistent. For each

[2]Our algorithm differs from the ARPANET algorithm in some respect, but we omit the details of our algorithm in this paper because they do not change our scheme for implementing the flows

destination, a node computes and maintains the current shortest path, that is, the path with minimum delay.

To simplify computation of paths for flows, we include the following additional information in each link status update message:

- average delay measured for each level of priority except that for the highest priority level.
- average bit–error rate measured. The bit error rate includes only the packet losses due to transmission errors.
- available link capacity[3] for allocation of flows in bytes/sec.

The link status update messages are short and do not cause excess overhead because each node generates such a message every minute. Whenever failure of a node or a link is detected, an update message is generated and broadcast immediately throughout the network. Because the control messages have highest priority, the link status messages propagate quickly throughout the network.

3.3 Flow setup

Each node has a flow arbitrator responsible for setting up and managing a flow. Peer flow arbitrators at all nodes cooperate with each other in implementing flows. A flow arbitrator implements the flows using the following strategy:

1. A part of each link's capacity is reserved for carrying the flow traffic and we call it the *flow reservoir*. A node makes part or all of this capacity available to a flow by reserving it for the duration of a flow's lifetime.
2. A flow arbitrator selects and reserves a path to the destination in two steps. First, using the currently available traffic information about the current paths, it selects a path to the destination that meets the delay, error rate and throughput specifications of the flow. Then the flow arbitrator tries to install that path by requesting all the nodes in the path to reserve resources for that flow. Once the resources are reserved, the path selection is complete. However, if two or more sources attempt to reserve resources on the same path, the reservation request may fail and, in that case, the arbitrator tries alternate paths until it succeeds.
3. Once a path is selected, it remains in force for the lifetime of the flow barring any link/node failures. All the traffic in the flow is routed on the same path.
4. At the time of flow creation, all the *courier* nodes (nodes other than the source and destination of the flow) in the selected path reserve an amount of link capacity equal to the throughput specification from the link's flow reservoir. To achieve the desired delay bound, the flow traffic is assigned higher priority than the non–flow traffic and we refer to this priority as the *flow priority*. Reserving the link capacity and using the higher priority minimizes queuing and switching delay at each node in the path.
5. The reservoir capacity is divided among the requesting flows so that traffic in a flow does not experience additional delays in the presence of traffic from other flows. Unlike virtual circuit networks that exclusively reserve the channel capacity for a connection, the link capacity reserved for a flow is available to the ordinary, non–flow, datagram traffic when a flow is not transmitting. Thus, we achieve significant savings in channel capacity by multiplexing lower priority traffic with the flow traffic over the same links.

In the following, we discuss the selection and installation of flow paths in detail.

3.3.1 Terminology

To simplify our discussion, we will use the following terminology. As part of network topology, each node maintains the following information for each link that it extracts from the link status update messages:

- average delay δ over the link for traffic with flow priority,
- remaining reservoir capacity ω available for flow allocation, and
- average bit error rate ψ.

A path from a source to a destination consists of a set of n links and has the following properties:

[3] A node computes the total capacity of a link using its link speed. For example, with synchronous hardware, link speed in bits per second (bps) divided by 8 gives the link capacity in bytes per second. In calculating link capacity, we do not consider the switching time because we assume that there is enough processing power at each node to saturate all its interfaces with data.

- Total amount of expected delay over the path for flow traffic. Computed as

$$\Delta = \sum_{i=1}^{n} \delta_i$$

- Amount of capacity available for flow allocation on the path,

$$\Omega = min\{\omega_i, i = 1, \ldots n\}$$

- Expected bit error rate of the path

$$\Psi = 1 - \prod_{i=1}^{n}(1 - \psi_i)$$

Because a sender need not specify all the parameters for a flow, the flow arbitrator at each node maintains the default values of these parameters for the entire path. They are D (default delay), W (default throughput), and E (default error rate).

3.3.2 Precomputation of Flow Paths

There are several possible approaches for finding alternate paths through the network and [25], [20] describe algorithms for finding *k shortest paths*. Our goal is to find and exploit paths with various performance characteristics. For example, sometimes path delay has less importance compared to the bandwidth requirements whereas sometimes both delay and throughput requirements of an application are satisfiable using a path that provides neither shortest delay nor maximum bandwidth. Therefore, our implementation need not be restricted to a particular method of finding alternate paths through the network. In the following, we describe an approach adopted in our implementation for simplicity.

Each flow arbitrator computes in advance the following paths to each destination:

1. As we mentioned earlier, each node uses the Shortest Path First algorithm to compute a path to each destination and we will refer to it as P_{spf}. The flow arbitrator maintains the values of its parameters, namely, Δ_{spf}, Ω_{spf}, and Ψ_{spf}.

2. For each destination, the flow arbitrator computes a path P_{fd} (a path with default delay bound) with Δ_{fd}, Ω_{fd}, and Ψ_{fd} such that $\Delta_{fd} \leq D$. The path P_{fd} is expected to be different from P_{spf}, but need not be completely disjoint from P_{spf}. In other words, this path is suitable for flows that do not specify the delay requirement and whose throughput specification is within Ω_{fd}.

3. Also, for each destination, each flow arbitrator computes a path P_{fw} (path with default throughput specification) with parameters Δ_{fw}, Ω_{fw}, and Ψ_{fw}, such that $\Omega_{fw} \geq W$. Such a path is suitable for those flows that do not specify any throughput requirement and preferably should be different from P_{spf} and P_{fd}, but need not be completely disjoint from either of them.

Path P_{fd} is computed as follows:

We want the path P_{fd} to be an alternative to the path P_{spf} with minimum delay among remaining possible paths to a destination. A set of such paths to all the destinations is computed using the routing tree computed by the SPF algorithm. Consider any link in the SPF routing tree. Such a link lies on the shortest path to a set of nodes and removal of the link leads to new paths to those nodes. At each step, our algorithm ignores one link from the SPF tree and computes new shortest paths to the set of nodes affected by the removal. The algorithm keeps track of shortest path to each destination found so far and replaces it with a new path if a shorter path is found at any step. This procedure is repeated for all the links in the SPF tree. At the end, we have a set of paths to all the destinations with minimum delay ignoring the shortest paths computed using the SPF algorithm.

Path P_{fw} is computed as follows:

A set of paths (P_{fw}) to all the destinations is found by performing a breadth first search through the network with links at each node ordered in non-decreasing order of remaining capacity for allocation of flows. At each hop, we select a link with maximum available flow reservoir (if more than one link with identical capacity exists, we choose one with lower delay). The resulting path to a destination has maximum throughput available for flow allocation, but the total delay may be larger than that on the P_{spf}. Any path P_{fw} is rejected if it cannot meet the default delay and error rate requirements.

3.3.3 Flow Path Selection

Given a flow specification, a flow arbitrator selects a path as follows:

if only delay δ is specified: It first compares the value of δ against the Δ_{spf} to see whether it can be satisfied. If so, it chooses P_{spf} provided Ω_{spf} and Ψ_{spf} are sufficient to satisfy the default throughput and error rate requirements. If throughput or error rates for P_{spf} are not sufficient, it tries the path P_{fd}. If P_{fd} also fails to meet the requirements, the flow arbitrator rejects the flow allocation request.

if only throughput ω is specified: The flow arbitrator tries to see whether the path P_{fw} has enough flow capacity to satisfy the value of ω. If not, no other path can satisfy the request and, therefore, the request is rejected. If a path P_{fw} does not exist, the arbitrator tries to use paths P_{fd} and P_{spf} in that order.

if delay, throughput, error rate are specified: The flow arbitrator tries to see whether any of P_{spf}, P_{fd}, and P_{fw} (in that order) can satisfy all the three parameters, δ, ω, and ψ. If not, it attempts to find a path P using the following algorithm: It performs a depth first search through the network graph starting at the source node. At each hop, it selects a link with minimum delay among those links that have available flow capacity to satisfy ω and whose error rate is within ψ. At the end, the search is successful if the total delay over the entire path is less than δ.

3.3.4 Flow Path Installation

A flow arbitrator exists at each node in the network and all the arbitrators on the path of a flow cooperate in setting up a flow. Once a path is selected, the flow arbitrator sends a *flow setup* request packet with source routing to the destination. The request contains a network wide unique flowid[4] and the amount of capacity that must be reserved. The flow arbitrator at each node on the path of the packet examines the packet and reserves the capacity requested. If a node cannot reserve the capacity, it sends the request back to the source indicating failure, otherwise it reserves the capacity and forwards the packet to the next node on the path. The flow arbitrator at the destination sends back a packet with source route to all the nodes on the flow including the source confirming the successful installation of the flow. All the subsequent traffic for the flow carries the flowid with it. Each courier node maintains a table of valid flowids, along with the identity of the source node and the output link onto which the flow traffic must be forwarded. The information in the table is time-driven and an entry is flushed if the node does not see any traffic for the flow over a long period of time. This timeout mechanism handles the case when a control message during flow setup or termination is lost and fails to reach some of the courier nodes.

When a flow arbitrator receives a *flow terminate* request from the sender, it sends a flow termination request with the flowid to the destination of the flow. Each courier node along the flow path then frees the reserved capacity and flushes the flow entry from its table.

[4]Because each node has a unique identifier, a node can construct a unique flowid by concatenating its nodeid with a locally unique flow identifier

3.3.5 Flow Failures

When a node on the path of a flow detects failure of a link or a node in the path, it sends a *node/link down* message to the source of the flow along with the flowid. The source, in turn, informs the sender of the flow of the failure using the *failed* primitive, and waits for some time greater than the total delay on the flow path before it sends a flow termination message to all the nodes on the path. The flow termination message is routed independent of flow traffic.

3.4 Discussion

Implementation of flows involves two costs, namely, setup cost and cost of precomputing alternate paths. The setup cost is a one–time cost and when amortized over the lifetime of a flow will not be significant because we expect that a flow will be set up to carry traffic over a substantial interval. For example, the bulk data transfers or the real–time applications involving image or voice transfers involve a large amount of data transfers over a long period. Also, at higher levels, a sender may use a flow to multiplex traffic from several sources. Therefore, the setup cost will be less than that incurred in setting up virtual circuits for each end–user. Precomputation cost includes the cost of communicating link status messages and the processing cost. Because the SPF routing algorithm uses the link status messages, the only cost is the overhead of additional information. Given the frequency of updates and the small amount of additional information needed, the communication overhead is insignificant. The newer packet switch designs [29,11] to handle high speed fiber interfaces use multiple processors as does our Multiswitch architecture. These packet switches have enough processing power to devote a processor to precomputation of routes in background without affecting the normal packet switching. Use of priorities in packet handling does not incur significant overhead in packet switching because such link level processing can be accomplished in network interface hardware [13].

Our current design terminates a flow when one or more of the links or nodes in its path have failed. Instead, we could arrange a courier or a source node to dynamically reroute a flow around a failure as long as an alternate path exists to satisfy the specifications. We chose the former approach to keep the design simple and compatible with our "best effort" philosophy.

3.4.1 Comparison with Virtual Circuits

Even though setting up flows involves reservation of bandwidth, flows are different from the virtual circuits.

First, we are interested in reserving two components of network resources (namely, processing power at a switch and channel capacity) to guarantee performance in terms of a certain data rate and bounds on average and maximum delays. A virtual circuit mechanism only reserves buffer space (bandwidth); it does not provide for for specifying and satisfying performance needs of an application. A virtual circuit network is still prone to congestion and may have wide fluctuation in transmission delays, whereas flow traffic has higher priority and is not affected by the presence of datagram traffic.

Second, even though there is an initial cost for setting up a flow similar to the one for a virtual circuit, there is no network layer error or flow control overhead with flows as in the case of each virtual circuit connection.

4 Examples of Uses

At higher levels, flows may be used for either application–to–application or host–to–host level communication.

application–to–application Examples of such applications are digital voice or video transfers, multimedia conferencing, interaction between a user and an application under a network window system, and remote data and image acquisitions. Also, protocols such as VMTP [4] and NETBLT [7] can use a separate flow to carry control information such as acknowledgements and another flow to carry data for an application.

host–to–host Examples of these applications involve kernel–to–kernel communications such as implementations of Remote Procedure Calls (RPC)[3], UNIX interprocess communication (IPC), and SUN Microsystem's Network File System (NFS). A kernel may multiplex multiple IPC communications over a single long term flow. Communication involving NFS client and server is a good example of a flow with low delay and low throughput.

5 Possible Extensions and Design Alternatives

1. **Other Parameters** In addition to the performance characteristics, it should be possible to associate additional optional characteristics with a flow. Examples of such characteristics include reliability (making absolute guarantees for delivery), security, privacy, and integrity. Another useful flow parameter will allow the sender to specify the possible persistence of a flow in quantitative terms. For example, perpetual flows would be useful for kernel–to–kernel communication for applications like SUN NFS [21]. A flow arbitrator may compute in advance alternate paths for a perpetual flow.

2. **Dynamic Flow Management**
   Currently, a flow arbitrator declares a flow failure when any of the links or nodes on the path of the flow fails. Instead, we can use a dynamic scheme under which the flow arbitrators cooperate to reroute the flow traffic around a failure.

3. **Multiple Path Routing**
   For some pairs of source/destinations, multiple paths with similar characteristics may exist. In such a case, a flow arbitrator may use more than one path to a destination to set up a flow and divide the flow traffic among those paths.

4. **Multipoint Flows** Future communication networks will support flexible, multipoint communication needed by a wide class of applications. A broadcast service such as entertainment video or a video lecture requires a *one–to–many* communication, whereas multiperson conferencing requires a general multipoint communication that imposes additional requirements on a packet switched network [27]. Extending the concept of flow to handle multicasts and multipoint communication is a challenging and, as yet, unexplored task.

6 Comparison with Related Work

6.1 Type Of Service Routing with loadsharing (BBN)

In [10], Gardner and others describe a dynamic, multipath routing algorithm. They identify three types of services similar to those specified in the Internet Protocol. A path generation algorithm finds two paths for each type of service for each destination. Path generation finds new paths periodically (every 5 to 15 minutes) in response to information collected through link status update messages, and, thus, traffic pattern changes lead to new paths. A flow allocator at a source node assigns traffic flows to different

paths depending on TOS and current state of the network. When new traffic patterns develop or a source node receives information about the congestion state of paths, it uses this information to adjust the flows of traffic on each path. Gardener et al describe a simulation of their scheme that shows that the multiple path routing algorithm provides better throughput and "reasonable" delays compared to the current single path algorithm used in the ARPANET.

However, the scheme suffers from the following drawbacks:

Instability Because route allocation moves traffic flows among many paths, it may lead to oscillations in load on different links with fluctuations in delays.

Congestion Failure In the dynamic congestion control scheme used, a source node detects the presence of congestion over the path of a flow using the data collected periodically, and then responds by reducing the traffic or by diverting the traffic to less congested paths. Under such a scheme, the perceived state of the network always lags behind the actual state, and this information gap in combination with fluctuations in the flow paths leads to an unstable congestion control algorithm that fails to converge.

Coarse Granularity The loadsharing scheme defines three types of service that characterize the traffic in broad, qualitative terms such as delay sensitive vs. delay insensitive, and low throughput vs. high throughput. A higher level cannot specify performance needs in quantitative terms and there are no predictable bounds on delays and error rates. Such an approach is not suitable for real time applications.

6.2 Real–Time Message Streams

The designers of the DASH project [2] at UC, Berkeley define a new software architecture for network communication. To provide a stream–style communication in long distance networks, the DASH communication system provides an abstraction called *real–time message stream* (RMS). A RMS is a simplex stream that has performance, reliability, and security parameters associated with it, and a RMS abstraction appears at the network and higher levels of the communication architecture. The performance parameters of a RMS include capacity which is an upper limit on amount of data outstanding within a RMS, delay bounds, and other parameters that allow specification of privacy and security characteristics.

The paper [2] does not address the issues of implementing RMS in a packet switched networks. Capacity enforcement is left to the RMS clients and the paper does not address the question of how to support RMS in a datagram network. Specifically, the issues involved in determining whether to grant a new RMS request, and in meeting the delay bounds remain to be investigated.

6.3 Connection–Oriented Transport

[17] describes ongoing work at the Washington University at St. Louis. The paper discusses a proposal for a connection–oriented transport service in a packet switched network. The network makes explicit resource allocation decisions at the time of connection establishment based on the amount of bandwidth needed by a user. The resource specifications may be "degradable" allowing the network to take the resources away from such connections to accommodate new traffic. However, the paper does not provide a clear definition of semantics of connections and schemes for implementing connections in a best effort delivery system.

7 Summary

In this paper, we have proposed a method for providing performance guarantees in best effort delivery systems. We provide a *flow* abstraction as a communication paradigm to the upper layers that can be used by applications that demand specific performance bounds. Communication entities at the network level specify the delay, throughput, and reliability bounds for flows that are guaranteed by the flow arbitrators at the link level. Given the unpredictable nature of packet delivery and performance behavior under a best effort delivery system, our emphasis is on devising schemes that allow predicting and guaranteeing bounds on delays and throughput between end points of communication.

Presently, we are building a prototype network that will act as a testbed for our algorithms and further experimentation. An interesting benefit of our scheme for implementing flows is its impact on congestion avoidance. Because we gather information about the amount of capacity available at each node and the amount of traffic load on each link, it is possible to use that information to implement a long term congestion avoidance policy that controls the amount of traffic entering the network and accepts new traffic only if the intermediate nodes along the path to the destination are not overloaded. We plan to investigate this issue more closely. Another issue that mer-

its investigation is how to extend our scheme to an internet consisting of heterogeneous networks. [17] provides a framework for work in this direction.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison–Wesley Publishing Company, 1974.

[2] D. P. Anderson. A Software Architecture for Network Communication. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 376–383, IEEE Computer Society, June 1988.

[3] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] D. R. Cheriton. VMTP: a transport protocol for the next generation of communication systems. In *SIGCOMM '86 Symposium*, pages 406–415, ACM, August 1986.

[5] D. Clark. Options for Research in Networking. January 1988. Unpublished Note, M.I.T. Laboratory for Computer Science.

[6] D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 171–180, ACM, October 1985.

[7] D. Clark, M. Lambert, and L. Zhang. NETBLT:A High Throughput Transport Protocol. In *SIGCOMM '87 Workshop*, pages 353–359, ACM, August 1987.

[8] D. Comer, J. Steele, and R. Yavatkar. *An Overview of MultiSwitch Project.* Technical Report CSD–TR–753, Computer Science Department, Purdue University, October 1988.

[9] K. Eng. A Photonic Knockout Switch for High–Speed Packet Networks. *IEEE Journal on Selected Areas in Communications*, 6(7), 1988.

[10] M. Gardner, I. Loobeek, and S. Cohn. Type–Of–Service Routing with Loadsharing. In *Globecom*, pages 1244–1250, 1987.

[11] S. Giorcelli, C. Demichelis, G. Giandonato, and R. Melen. Experiements with Fast Packet Switching Techniques in First Generation ISDN Environment. In *Proceedings of International Switching Symposium*, pages B5.4.1–B5.4.7, IEEE, 1987.

[12] Z. Haas. *Packet Switching in Future Fiber–Optic Wide Area Networks.* PhD thesis, Stanford University, May 1988.

[13] H. Kanakia and D. Cheriton. The VMP Network Adaptor Board (NAB): High–Performance Network Communication for Multiprocessors. In *SIGCOMM '88 Symposium*, ACM, August 1988. To appear.

[14] B. Leiner. Network requirements for scientific research. Internet Task Force on Scientific Computing, August 1987. RFC 1017.

[15] J. McFadyen. Systems Network Architecture: An Overview. *IBM Systems Journal*, 15:2–23, 1976.

[16] J. McQuillan, I. Richer, and E. Rosen. The New Routing Algorithm for the ARPANET. *IEEE Transactions on Communications*, COM-28(5):711–719, May 1980.

[17] G. Parulkar and J. Turner. *Towards a Framework for High Speed Communication in a Heterogeneous Networking Environment.* Technical Report WUCS–88–7, Department of Computer Science, Washington University, St. Louis, Missouri 63130, 1988.

[18] J. Postel. Internet Protocol. DARPA Networking Information Center, Request For Comments, September 1981. RFC 791.

[19] J. Postel. Transmission Control Protocol—DARPA Internet program protocol specification. September 1981. RFC 793.

[20] H. Rudin. On Routing and "Delta Routing": A Taxanomy and Performance Comparison of Techniques for Packet–Switched Networks. *IEEE Transactions on Communications*, COM-24(1):43–59, January 1976.

[21] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the SUN Network File System. In *Proceedings of the Summer USENIX Conference*, pages 119–130, USENIX Association, June 1985.

[22] W. Stallings. *Data and Computer Communications.* Macmillan Publishing Company, 1985.

[23] D. Swinehart, L. Stewart, and S. Ornstein. *Adding Voice to an Office Computer Network.* Technical Report CSL–83–8, Xerox Palo Alto Research Center, Palo Alto, California 94304, February 1984.

[24] D. Terry and D. Swinehart. Managing Stored Voice in the Etherphone System. *ACM Transactions on Computer Systems*, 6(1):3–27, February 1988.

[25] D. Topkis. A K Shortest Path Algorithm for Adaptive Routing in Communication Networks. *IEEE Transactions on Communications*, 36(7):855–859, July 1988.

[26] J. Turner. Fast Packet Switching System. U.S. Patent No. 4,494,230, January 1985.

[27] C. Weinstein and J. Forgie. Experience with Speech Communication in Packet Networks. *IEEE Journal on Selected Areas in Communications*, SAC–1(6):963–980, December 1983.

[28] R. Yavatkar. *An Architecture for a High–Speed Packet Switched Network.* Technical Report, Dept. of Computer Science, Purdue University, West Lafayette, IN 47907, May 1988. Proposal for Ph.D. Dissertation.

[29] Y. Yeh, M. Hluchyj, and A. Acampora. The Knockout Switch: A Simple, Modular Architecture for High–Performance Packect Switching. In *Proceedings of International Switching Symposium*, pages B10.2.1–B.10.2.8, IEEE, 1987.