Building A Programmable Multiplexing Service Using Concast *

Kenneth L. Calvert, James Griffioen, Amit Sehgal, and Su Wen
Department of Computer Science
University of Kentucky
Lexington, KY 40506
[calvert,griff,asehgal,suwen]@dcs.uky.edu

Abstract

Concast is a scalable "inverse-multicast" network service: messages sent from multiple sources toward the same destination are merged into a single message that is delivered to the destination. The mapping from sent messages to received messages is programmable, so the service can be tailored to the needs of specific applications. However, the service can also be used as a building block for other generic *network services, such as a packet multiplexing service that encapsulates multiple small packets into a single larger packet and then unencapsulates the small packets at their (common) destination. Such a service offers several potential benefits, including reduced packet processing overhead and increased fate-sharing, but must also be carefully designed to avoid problems caused by added packet delays. In this paper we show how concast can serve as the basis for a multiplexing service that can be tailored to the needs of the application. We present simulation results showing that the benefits of our multiplexing service vary with delay. We also show that given certain queue-manipulation capabilities, benefits can be achieved with zero added delay.*

1. Introduction

Recently, the research community has proposed and debated the merits of having the network bundle small packets (traveling in the same direction) into larger units for transport, and then dispersing them again when their paths diverge [1, 2]. The basic idea is to delay small packets slightly as they go through certain routers, in hopes that other small packets heading in the same direction will arrive. After a certain time, any waiting packets are encapsulated in a single large packet and forwarded toward the destination. At the destination, or possibly at intermediate stops along the way, encapsulated packets are extracted and delivered using standard unicast delivery mechanisms. Because messages are packed and unpacked transparently *en route*, senders and receivers do not necessarily know that aggregation is occurring. In other words, the multiplexing/demultiplexing service is application-independent, combining packets from many flows, without any specific information about the application to which the flow belongs.

A multiplexing service that replaces many small packets with fewer, larger packets is interesting in that it offers the potential to improve the network's *overall* health and performance. Because routers generally have a fixed per-packet processing overhead, small packets can consume a disproportionate amount of a network's resources. Replacing several small packets with a single large one should reduce demands on a switch's resources, benefitting all flows going through the switch. A reduced packet count can result in fewer packets drops and higher throughputs for all flows. Moreover, studies have shown that about half the packets in the Internet are 50 bytes or less [8], and about 60% are less than 100 bytes.

Unfortunately, delaying packets in order to combine them can be harmful in several ways. First, many end-to-end protocols depend on accurate Round Trip Time (RTT) estimates for mechanisms such as retransmission timeouts. Artificially increasing the RTT (in either or both directions) can reduce throughput under some conditions. Second, intentionally delaying packets may change the inter-packet gap between two successive packets. Several protocols, including TCP, use the arrival of incoming packets to trigger the transmission of outgoing packets. Perturbing the inter-packet arrival times can affect the burstiness and overall transmission rate of a flow [3]. In other case, inter-

* Effort sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0514. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force Research Laboratory, or the U.S. Government.

packet gaps are used to estimate bottleneck bandwidths [11]. Third, packet reordering can occur if large packets are passed through immediately while small (but temporally related) packets are delayed. For certain protocols this may cause otherwise unneccessary error recovery traffic (e.g., in NACK-based protocols). Fourth, no single fixed delay will be appropriate for all applications. In other words, the delay a packet can tolerate depends on the application to which the packet belongs.

Determining the appropriate amount of delay for aggregation is not the only challenge in setting up a multiplexing service. For example, the (maximum) number of small packets that can be combined into a large packet depends on the path MTU. Unfortunately, this information is typically unavailable at the point of aggregation. Determining which network nodes should perform multiplexing, to which packets, and under what conditions are also difficult problems. Thus, though there appear to be good reasons for deploying a multiplexing service, the problems associated with such a service are potential show-stoppers.

In this paper, we explore the use of the *concast* programmable network service to support generic network services, specifically a multiplexing service. Note that our goal is not necessarily to advocate or demonstrate the need for a multiplexing service; this is being studied by others [2]. Rather, we use the multiplexing service to explore the capabilities and limitations of our programmable network service (concast). Our study shows that the concast service and API is robust and flexible enough to implement other generic programmable network services. We also show that given the ability to examine and manipulate the queue of packets awaiting transmission, it is possible to provide some multiplexing benefits *without increasing delay*.

The existence of useful *generic* concast-based services interesting because such services could be "bundled" with the concast implementation itself, providing more or less immediate benefits and a way to bootstrap deployment of concast. We argue that such benefits are crucial for services like concast—that is, programmable services designed for "aware" applications—to succeed. Without them, a chicken-and-egg problem arises: applications cannot adapt to or depend on the service until it is supported in operating systems, but network vendors and operating system manufacturers are reluctant to implement the service unless there are applications to use it.

The rest of this paper is organized as follows. Section 2 describes past approaches to the problem of small packet aggregation. Section 3 then introduces the concast service, including the merging framework, within which we define our multiplexing service. Section 4 describes our new multiplexing service, including the API for setting service parameters. Section 5 presents simulation results that illustrate the benefits of building a multiplexing service using concast. Section 6 concludes the paper.

2. Related Work

The idea of combining packet flows together has been proposed and studied in various application-specific contexts. Examples include reliable multicast (group feedback) applications that combine messages in order to avoid implosion [9, 12, 14, 21], multiplexed persistent TCP connections [13] that avoid connection setup, teardown, and slowstart, and consolidated control information used in fate-shared congestion control approaches [4, 18].

A more general application independent service, called *gathercast* [2], has been proposed as a way of increasing efficiency by aggregating small datagrams (e.g., TCP acks) into a larger packet and transporting them together for some or all of the way toward their destination. At gathercast-enabled routers, small packets are delayed briefly waiting for other small packets going in the same direction. When an MTU's worth of small packets has been accumulated, or when a certain time has expired, any accumulated packets are packed into a single gathercast packet and forwarded. At the destination, the gathercast packet is broken down and the original packets are delivered to the receiving applications. Preliminary results suggest a reduction in packet drops resulting in improved throughput [2].

Gathercast deals with the delay issues described in Section 1 by placing an upper bound on the maximum delay a packet can experience. The bound is enforced using a predefined timeout value that limits how long packets will be delayed at a node. To prevent delay from accumulating, a packet can be buffered (delayed) at most once. Thus a packet has at most one opportunity to be combined with other packets. A second parameter defines the maximum number of packets that can be aggregated. If this maximum is reached before the timeout occurs, the packet is forwarded, and experiences less delay. Unfortunately it is not clear from [2] how to select a timeout value such that the delay does not adversely affect protocol performance or correctness. Moreover, all sources that transmit packets across a gathercast link (transformer tunnel [16]) are subject to the same delay. Thus the delay is not based on the rate at which sources (i.e., applications) generate packets or an application's ability to tolerate the (fixed) delay. To avoid the ack compression problem caused by removing inter-packet gaps, the gathercast algorithm does not combine two packets from the same source, thus potentially preventing non-ack traffic from being combined (e.g., small TCP data segments). To maintain delivery order, gathercast immediately flushes delayed packets when any large packet arrives, even if ordered delivery is not important to the application. Also, the maximum packet size is set by the receiver and is the same for every node in the graph, regardless of path MTUs.

In short, existing approaches fail to solve key problems related to application control over the delay and placement/instantiation of the service. In the next sections we show how these problems can be solved using the concast service.

3. The Concast Service

Recently we described a new programmable network service called *concast* [5], which is essentially the inverse of multicast. In defining the concast service, we were guided by the goal of maintaining symmetry with IP multicast where it was reasonable to do so. Like multicast, concast is a best-effort service, and does not provide any guarantees. Multicast uses a single group ID (multicast address) to represent a group of receivers; concast uses a single group ID to represent a group of senders. In multicast, the sender does not know who the receivers are; in concast the receiver does not know who the senders are. In multicast, the network *duplicates* packets at branch points enroute to the group of receivers. In concast, the packets transmitted by a group of senders are *merged* at confluence points in the network; thus each packet delivered to a concast receiver may contain information from packets transmitted by several different group members.

The basic unit of concast service is the "flow", identified by an (R, G) pair, where R is the (unicast) IP address of the receiver and G is the concast group identifier (representing the set of senders). Concast datagrams are identified by a "Concast ID option" in the IP header. The IP destination address R, together with the group ID G carried in the Concast ID option, uniquely identify the concast flow.[1]

The concast receiver is responsible for providing the network with a *merge specification* that defines the mapping from sent messages to the delivered message. The merge specification defines the computations that must be carried out to merge packets belonging to the same (R, G) flow, including a definition of what packets are eligible for merging. In multicast, the group of receivers use a signalling protocol to join the group. Similarly, concast senders execute the *Concast Signalling Protocol (CSP)* [5] to join the group as a sender. The CSP protocol attaches the sender to the concast tree and also causes the merge specification to be pulled from the receiver (or, more often, the nearest router on the concast tree) toward the sender.

As concast datagrams travel through the network, they are processed hop-by-hop. Concast-oblivious routers do not recognize the Concast ID option and simply forward the datagram toward R as usual. This means the service can be incrementally deployed and is useful even when partially deployed. At each concast-capable node, concast packets (flagged by the Concast ID option) are diverted for merge processing, the flow ID, (R, G), is extracted from the packet, the merge specification for flow (R, G) is retrieved, and the flow-specific processing is applied according to the merge specification. In the next section we discuss the details of the hop-by-hop processing and merge specification.

3.1. Merge Specifications

The hop-by-hop merge processing that occurs at concast-capable nodes has a fixed part, which is common to all concast flows, and a variable part, which can be different for each flow. The merge specification defines the variable part, and consists of a definition of the *merge state block*, which stores the state of each ongoing merge computation, and four functions described below. The fixed part of the merge specification comprises a flow state block, which contains global information (including the merge specification) associated with the flow, and the per-packet processing algorithm depicted in Figure 1.

***getTag*(m):** a *tag extraction* function returning a key identifying the datagram equivalence class (DEC) to which the given message belongs. Messages m and m' are eligible for merging iff *getTag*(m) = *getTag*(m').

***merge*(s, m, f):** the function that combines messages together. The first parameter is the current *merge state block*, which summarizes messages that have already been processed. The second parameter is the incoming message, which will be merged into the saved state s. The third parameter is the flow state block for the concast flow to which m belongs. Returns an updated message state block.

***done*(s):** the *forwarding predicate* that checks whether a message should be constructed (by calling *buildMsg*) and forwarded to the receiver.

***buildMsg*(s,f):** the *message construction* function, which takes the current merge state and flow state, and returns the payload to be forwarded toward the receiver, along with a possibly-updated message state.

When a node recognizes a concast packet during forwarding, it retrieves any state associated with the given (R, G) flow. If flow state exists, the *Datagram Equivalence Class* (DEC) of that message is computed using the supplied *getTag*() function, and the associated message state block is retrieved. A new message state block is computed from the old state and the incoming message, and the result is stored back with the tag. If the *done* predicate indicates that merging is finished, the *buildMsg* function is called to generate a payload (along with a new message state); an IP datagram containing that payload and the appropriate addresses is then forwarded toward the receiver.

[1] An earlier version of the concast service [5] carried G in the IP source address which is more symmetric with multicast. The Concast ID option is more robust to filtering and anti-spoofing source checks applied by some routers.

```
ProcessDatagram(IPAddr R, ConcastGroupID G,
                ConcastDatagram m) {
  FlowStateBlock fsb;
  DECTag t;
  MergeStateBlock s;

  fsb = lookUpFlow(R,G);
  if (fsb != NULL) {
    t = fsb.getTag(m);
    s = findMergeState(fsb,t);
    s = fsb.merge(s,m,fsb);
    if (fsb.done(s)) {
      IPPayload merged;
      (s,merged) = buildDatagram(fsb,s);
    forwardConcastDG(R,G,merged);
    }
    saveMergeState(fsb,s,t);
  }
}
```

Figure 1. Network per-packet processing.

By supplying different instantiations of the four functions above, the concast service can be customized for different applications or classes of applications. These functions can be supplied using an encoding designed for that purpose (e.g., a mobile-code language [7, 20, 15]). For example, our prototype implementation [6] supports merge specifications coded in restricted forms of Java and Tcl.

3.2 Merge Processing Context

The operations available for use by the four functions making up the variable part of the merge function determine the capabilities of the particular concast service. In order to implement a generic multiplexing function, certain capabilities are needed. For example, the *done*() function needs to be able to set up timeouts to arrange for later processing in case additional packets do not arrive.

In our previous applications of the concast framework [5, 6], merge processing was entirely self-contained and did not require any extraordinary access to other processing at a node. For the multiplex service, however, it is very useful to allow the merge functions of a flow directed toward receiver R to access the queue of all datagrams awaiting transmission toward R. This situation is depicted in Figure 2. Upon arrival, packets (both concast and non-concast) are queued for transmission toward R. When they reach a point near the front of the queue they undergo concast processing. The concast merge function is able to inspect and possibly modify (in a restricted manner) the queue of packets waiting *behind* the concast packet being processed. To maximize the portion of the queue that is accessible to the merge function in this way, the post-merge-processing part of the queue has just enough capacity to keep the channel from becoming idle during merge processing.

In this context, we assume the *done*() and *buildMsg*() functions can invoke operations that examine and manipulate these packet queues. Specifically, *done*() can check whether any other packets belonging to its DEC are in the queue. If so, *buildMsg*() can remove those packets from the queue for immediate merging and forwarding. As we shall see, the ability of these *done* function to access the queue for additional packets that can be merged permits multiplexing without adding delay.

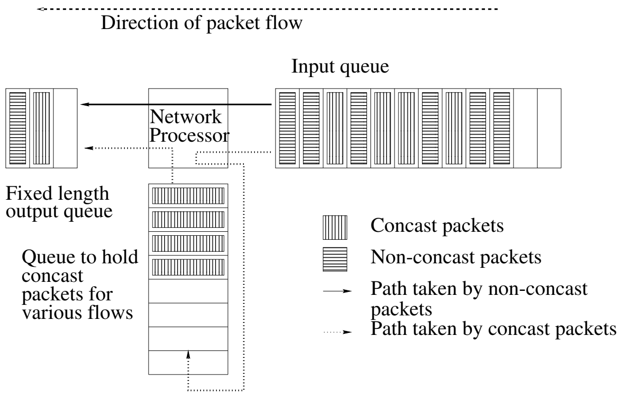

Figure 2. Merge Processing Context

3.3. Concast Programming Interface

The concast module exports an interface supporting three operations: (1) associating a sending endpoint with a particular concast flow (receiver, concast group pair); (2) installing a merge specification for a given receiver and concast group (R, G); and (3) processing a concast message as described in Figure 1.

join_concast_flow(endpoint, concast_addr, receiver_addr) arranges for packets originating from the the given endpoint, destined for the given receiver, to be marked with Router Alert and given the concast group address in the source field. Triggers the signaling process to retrieve the Merge Specification from the specified receiver.

install_merge_spec(flow_id, merge_spec) sets up state for the specified flow; `merge_spec` is a data structure that contains the definitions of the functions given in the previous section in some platform-independent form (e.g. Java bytecodes).

ProcessMessage(Receiver, Group, Datagram) is as defined above. Invoked when IP identifies a concast packet. Calls the IP forwarding routine to forward completed IP datagrams when processing is complete.

Application-level programs invoke the first two methods above via standard API control routines, for example:

IO_Control (`endpoint`, `CCAST_JOIN_FLOW`,
`ccast_fspec_struct *fspec`)
IO_Control (`flow_spec`, `CCAST_SET_MERGE`,
`merge_spec_struct *mergespec`)

However, as we shall see in Section 4.3, they can also be invoked by other protocol modules within the network subsystem.

4. Packet Multiplexing Using Concast

In this section we describe a packet multiplexing service based on the concast framework described above; such a service can be deployed in any concast-enabled network.[2] Unlike the gathercast approach [2], which is transparent to the sending and receiving hosts, our service only combines packets if receiver *and* senders both request the service. In other words, senders can control if and how their packets are multiplexed with other packets. This allows applications to ensure that their own protocol requirements are not violated by the multiplexing service. Moreover, a sender's packets will only be combined with other packets that are destined for the *same receiver*, and only when they are tagged with the same concast group ID. Note that multiple concast groups can be used for aggregation simultaneously; applications can restrict the set of packets with which their packets will be combined by using a group ID specific to that application. More generally, however, use of a well-known concast group ID for generic multiplexing allows for aggregation of messages from any host/application.[3]

The gathercast service described in [2] is based on *transformer tunnels*, a tool for deploying functionality in the network without modification of the network layer service. Transformer tunnels, which determine the paths on which packets are aggregated, have to be administratively deployed and configured using some (unspecified) out-of-band mechanism. Our service, on the other hand, is based on a network-layer service (concast) plus a Multiplex protocol module in the end systems. As such, functionality is *automatically* deployed to (concast-capable) nodes where it is needed. In particular, packets are not delayed at a node for aggregation unless multiple flows heading for the same destination converge at that node, and end systems are unaware of which interior nodes support concast. Thus the concast approach requires modification of network layer at convergence points, but thereafter is automatically deployed; transformer tunnels are administratively deployed, but require no network-layer modifications and can be transparent to applications.

4.1. The Multiplex API

Receivers and senders request multiplexing service via the *multiplex API*. The API consists of a single control method that enables/disables multiplexing on the current flow. The method is shown below in an OS-independent format[4]. The method takes a `flowid` (or endpoint identifier) and five other parameters:

Mplex_Control (`flowid`, `OnOff`, `SorR`, `MplexGrp`,
`MaxLocalDelay`, `MaxTotalDelay`)

OnOff	enable or disable multiplexing on this flow
SorR	specifies whether the caller is a sender or receiver
MplexGrp	the (concast) group ID to use
MaxLocalDelay	maximum time a packet may be delayed for aggregation by any one node
MaxTotalDelay	maximum time the packet may be delayed for aggregation across the whole network

At the sender (`SorR` indicates "S"), turning on multiplexing results in the sender's node joining the specified concast group (if it is not already a member). This results in the merge specification being "pulled down" to each concast-capable node between the sender and the receiver (which is identified with the flow). At the receiver (`SorR` indicates "R"), turning on multiplexing makes the flow specification available for senders.

The `MaxLocalDelay` and `MaxTotalDelay` parameters limit the delay added by the multiplexing service. Because each application has its own delay requirements, delays are specified on a per-flow basis and are only configurable at the sender (delay parameters are ignored at the receiver). The specified values are placed in packets sent by the application, as described in the next section.

Normally the sending and receiving applications will agree on whether multiplexing should be used or not. However, if the sending application does not enable multiplexing while the receiving application does, packets will be sent unicast (instead of concast) but will arrive at the receiver. On the other hand, if the sender enables multiplexing but the receiver does not, one of two things will occur. If some other application on the receiver's machine has enabled multiplexing (with the same group ID), the merge spec will be avialable and packets will be multiplexed anyway. Otherwise, an error will occur during the concast signaling process, and the sender will revert to standard uni-

[2] A concast-enabled network minimally consists of concast-capable end systems (senders and receivers). Internal network nodes may or may not support concast. Those nodes that do not implement concast simply forward packets unmodified.

[3] At the receiver, the request to enable gathercast may be made by the application itself, or a merge spec may be installed administratively for (R, G), where R is the node's IP address and G is the well-known multiplexing concast group ID.

[4] The actual call would be OS-specific such as a `setsockopt()` call in systems that support sockets.

cast. In other words, even if the sender and receiver disagree on the use of multiplexing, packets will be delivered correctly to the receiver.

4.2. The Multiplex Merge Specification

In this section we outline the processing defined by the four functions that make up the multiplex merge specification. The merge state block stores a single multiplex packet, whose format is shown in Figure 3.

4.2.1 The *merge*() Function

The goal of the multiplex merge function is to aggregate small incoming IP datagrams into a single larger IP datagram. It is applied initially at the sending machine and then at each concast-enabled router on the path from the sender to the receiver. Because aggregation only makes sense for small messages, the *merge*() function immediately forwards IP fragments (from a large IP datagram) without further processing. Large packets whose size exceeds a local threshold are also immediately forwarded. In addition, *merge*() immediately forwards datagrams that have a multicast destination or contain IP options. Finally, *merge*() only aggregates datagrams having the same concast group ID and destination address. If a datagram makes it through this initial series of checks, it is encapsulated in an aggregate datagram for forwarding.

The simplest approach to aggregation is to simply carry the entire packet, including the IP header information, in the multiplex packet. However, the header overhead on small packets is significant and can be substantially reduced by eliminating redundant information and information that can be recomputed. The multiplex *merge*() function therefore drops IP header fields that can be regenerated at the receiver. Only the *source address*, *protocol number*, *TTL (Time-to-live)*, *payload* and *payload length* of each aggregated datagram is carried in the aggregate packet; other information is regenerated at the receiver from the received datagram's header. Consequently, the header-to-payload ratio, which can be significant for small packets, is reduced by a factor of two.

The format of the multiplex packet (`MplexPkt`) is shown in Figure 3. The `EncapsulatedPkt` structure records information from the original IP datagram (generated at the sender) that must be transmitted to the receiver and cannot be reconstructed from the multiplex packet's IP header. At the sender, the unicast address of the sender (`SrcAddr`) and the transport protocol number are placed in the `EncapsulatedPkt` so they can be put in the reconstructed IP datagram ultimately delivered to the receiver's IP layer. Clearly the original payload and payload length must be encapsulated as well.

```
struct MplexPkt {
  struct StdIPhdr iphdr;
  struct MplexHdr mplexhdr;
  struct EncapsulatedPkt pkts[];
}

struct MplexHdr {
  ushort initialTTL;  /* see text */
  ushort MaxTotalDelay;  /* Min of incoming */
  ushort DelayedSoFar;   /* Max of incoming */
  ushort MaxLocalDelay;  /* Min of incoming */
}

struct EncapsulatedPkt {
  struct hdr {
    uint SrcAddr;
    ushort TTL;          /* original DG TTL */
    ushort ProtocolNum;
    ushort PayloadLength;
  } hdr;
  uchar Payload[hdr.PayloadLength];
}
```

Figure 3. Structure of a multiplex packet.

The `MplexHdr` structure records summary information about the encapsulated packets. Delay fields are discussed below. The `initialTTL` field is used to ensure correct handling of time-to-live after packets have been aggregated. Normally, routers decrement the TTL field in the IP header as packet traverses the network. However, because multiplexed packets are encapsulated, the TTL field will not be decremented properly. Consequently, when reconstructing the original packets, the receiver must check that no TTL value would have expired. When an aggregate datagram is constructed, `initialTTL` is set to the maximum of all encapsulated TTL values. The TTL value in the `StdIPhdr` is also set to the same value. Upon arrival at the receiver, the number of hops along the path from the last concast processing point to the receiver is calculated by subtracting the TTL in the header from the `initialTTL`. This path length can then be used to recognize and discard encapsulated datagrams whose TTL values would have reached zero had they been sent unicast. This also makes it possible to decrement the TTL of encapsulated packets at intermediate re-aggregation points.

4.2.2 Delaying Packets

The major drawback to multiplexing is the introduction of additional delays. If added delay is unbounded, it may have a detrimental effect on higher-level protocols. Moreover, longer delays imply more resources required to store packets awaiting aggregation (for a given arrival rate). On the other hand, multiplexing is less useful if the additional delay is so small that there are few packets or no packets to combine.

In our model, the *sender* specifies the maximum additional delay a flow can withstand both at individual nodes and end-to-end. The two values specified via the API, `MaxTotalDelay` and `MaxLocalDelay`, are carried in the `MplexHdr` of each multiplexed packet. In processing a multiplex packet, the concast merge function will delay the packet for the smaller of `maxLocalDelay` or `MaxTotalDelay` $-$ `DelayedSoFar` milliseconds. The amount of time the packet is delayed the node is added to the `DelayedSoFar` field (initial value: zero) before forwarding.[5] Any multiplex-eligible packets that arrive while a packet is being delayed are merged with the delayed packet if they belong to its Datagram Equivalence Class. Multiplexed packets are forwarded if any of the following four conditions are met: (1) `MaxLocalDelay` of any packet is exceeded, (2) `MaxTotalDelay` of any packet is exceeded, (3) the concast router decides not to wait for more packets because its buffer space is running low, or (4) the multiplexed packet reaches maximal size.

As packets move through the network and merge with other packets, the delay values they carry must be adjusted. To ensure that no encapsulated packet exceeds its maximum delay value, the merge function sets `MaxLocalDelay` to the minimum of all the incoming packets being merged together. Similarly, `MaxTotalDelay` and `DelayedSoFar` are taken from the incoming packet with the least remaining time, *i.e.*, for all incoming packets i, select i such that $(MaxTotalDelay_i - DelayedSoFar_i)$ is minimized. More complex merge functions could be envisioned but this approach provides the maximum opportunity for combination without violating the constraints imposed by the high-level protocol.

4.2.3 Merging Multiplex Packets

Because packets are originated with the structure shown in Figure 3, we need to describe how the *merge*() function aggregates packets that each contain one or more encapsulated IP datagrams. We use an approach in which a new `MplexHdr` is formed from the incoming `MplexHdrs` by computing new maximum delays and `initialTTL`. The multiplex payloads from the incoming packets (*i.e.*, the encapsulated packets) are then copied directly into the payload of the outgoing packet (without demultiplexing or modifying them). The result is a non-hierarchical structured packet with minimal packet size.

4.2.4 The *getTag*(), *done*(), and *buildMsg*() Functions

The *getTag*() function simply returns a constant: all packets traveling to the same destination should be merged if possible, regardless of content, or the application to which they belong. Thus all (concast) packets matching a given (R, G) pair belong to the same Datagram Equivalence Class (and are eventually merged).

The *done*() function first checks the queue for other multiplex-eligible packets going to the same destination R. If any are present, it includes them in its subsequent calculations. It then checks whether the resulting `MplexPkt` is of maximal size. The maximum packet size depends on the path MTU, which can be obtained from the signalling (CSP) information at each node. Second, if buffer space is running low at the concast router, the *done*() function returns true, so the saved `MplexPkt` will be forwarded and buffer space can be retrieved. Finally it checks to see if any delay bounds have been exceeded. If not, *done*() sets a callback to occur just before the nearest delay bound is reached, and returns false.

If *done*() found any multiplex-eligible packets in the queue, the *buildMsg*() function removes them and adds them to the stored `MplexPkt`, provided there is room. The transit delay for these packets that "jump the queue" is thus reduced. This enables the service to combine packets even when their `MaxLocalDelay` and `MaxTotalDelay` values are zero. As we will see in Section 5, applications that cannot afford to be delayed can set allowed delay to zero and still achieve some of the performance benefits of multiplexing.

4.3. Sender and Receiver Processing

The sender and receiver protocol stacks must be modified in order to transmit and receive multiplex packets. Figure 4 illustrates the protocol stack organization used at both the sender and receiver.

The concast module is responsible for merging incoming and outgoing packets. Outgoing concast flows are passed to the concast module after IP processing for insertion of a concast source address and router alert option as well as merge processing (with other packets originating from the local host). The concast packets are then injected back into the stack for IP routing. Similar layering organizations have been used by other protocols such as IPSEC [19, 10].

The multiplex module is implemented as if it were a new transport-level communication protocol. Incoming multiplex packets first undergo concast processing as described above. After concast processing, the packet is passed to the IP module for processing. The IP module examines the transport protocol in the IP header in the normal way, and, finding the multiplex protocol number, hands the packet to the multiplex module for processing. The multiplex module then demultiplexes the packet to obtain the original IP datagrams. The datagrams are then inserted back into the protocol stack for standard IP processing and are ultimately delivered to the receiving application.

[5] Note that DelayedSoFar only refers to the delay added by multiplexing.

Output processing performs the inverse operation. Packets from multiplexed flows are passed to the multiplex module after IP processing. The packets are then encapsulated in multiplex packets and inserted back into the protocol stack for IP processing which then passes them on to the concast module for merging.

The per-flow-state maintained by the kernel records whether concast and/or multiplexing is enabled on the flow. The concast and multiplex-specific information is entered to the flow state via the API I/O control calls. When multiplexing is required, the user simply invokes the multiplex API. The kernel multiplex module then invokes the concast API to set up the concast group and install the (predefined) multiplex merge spec.

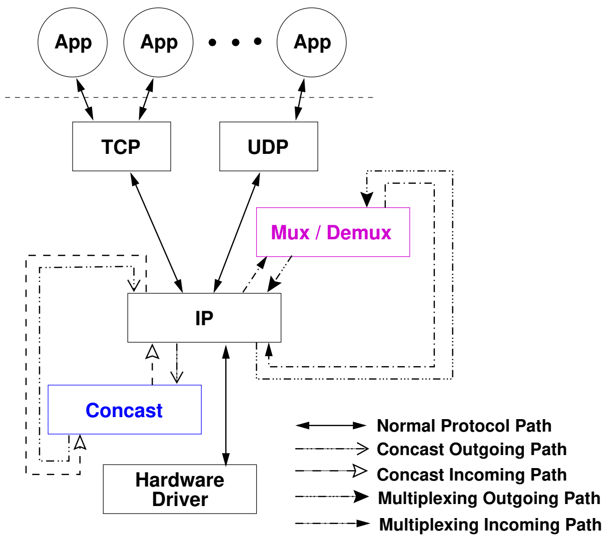

Figure 4. Sender/Receiver protocol stack organization

5. Simulations

The previous sections demonstrated the robustness and flexibility of the concast API via our ability to implement a multiplexing service. To show the performance benefits of a programmable multiplexing service, we simulated our concast-based multiplexing service using the UCB/LBL Network Simulator (NSv2.1) [17]. For the purposes of comparison, we also simulated standard TCP (i.e. without multiplexing). We used the GT-ITM topology generator([22]) to construct a wide-area transit-stub topology of 220 nodes, with a 10 ms link delay between directly connected nodes. There is one "core" transit domain consisting of four nodes. The "core" domain is connected to 24 stub domains each of which has, on average, nine nodes. Links connecting core nodes have a bandwidth of 100 Mbps, while the edge links have a bandwidth of 10 Mbps. We used a queue size of 20 packets for each link.

For the TCP simulations we used the standard TCP/Reno implementation in NS. We simulated our multiplexing service using the queueing mechanism described in Section 3.2 and the merge function described in Section 4.1. Although we could have allowed each flow in the simulation to specify its own delay requirements which could have produced even better throughput, we used the same fixed MaxLocalDelay for all flows to focus on the affect delay has on performance. We then ran simulations with differing MaxLocalDelay values. We did not impose any bounds on MaxTotalDelay. To avoid ACK compression, we implemented the *done* function so that it never merges ACKs from the same source.

5.1 Simulated Workloads

We simulated a web-server environment where a web server simultaneously transmits a 4KB web page to 200 client nodes scattered across the network. Each TCP connection transmits data packets from the server to the clients and TCP ACKs packets in the opposite direction. We ran experiments using two different traffic scenarios. In the first scenario, in addition to the TCP traffic from the web server, a minimal amount of UDP cross-traffic was injected in the system so that router queues were not empty, yet far from overflowing. Given the limited amount of traffic from the clients to the server, TCP ACKs for the web server experience no loss. We refer to this as the *low-loss environment*.

The second simulated environment is identical to the first except that we added cross traffic consisting of 40 TCP flows. The cross traffic competes with the web server traffic for network resources, thereby causing packet loss at network nodes. We refer to this as the *high-loss environment*.

5.2 Result and Analysis

Delayed TCP ACK packets can slow the protocol's response to lost packets. Consequently, TCP performs best when its ACK traffic is not delayed. On the other hand, if the network is congested and the loss rate is high, data or ACK losses force TCP into a congestion control phase that can severly degrade performance. In such congested environments, the benefits of delaying packets (particularly ACK packets) a short amount of time so that they can be combined with other packets, can outweigh the problems caused by the added delay. In other words, there is a tradeoff between combining packets and delaying packets. Large delays mean more combined packets, but too high a delay can degrade performance. Small delays produce fewer combined packets, but avoid the added delay. In other words, a

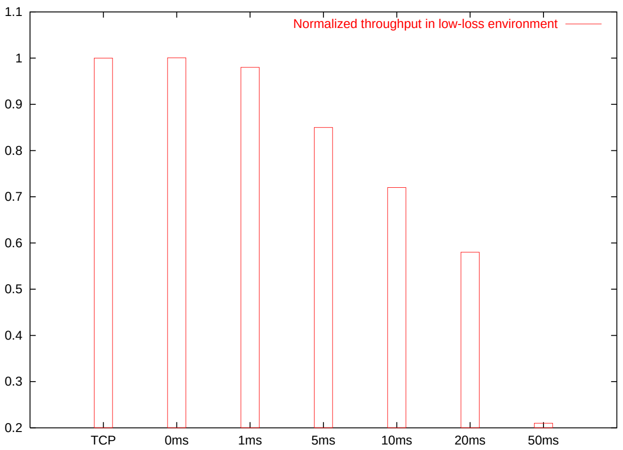

Figure 5. Normalized session throughput of 200 clients retrieving a 4KB page from a web server in the low-loss environment

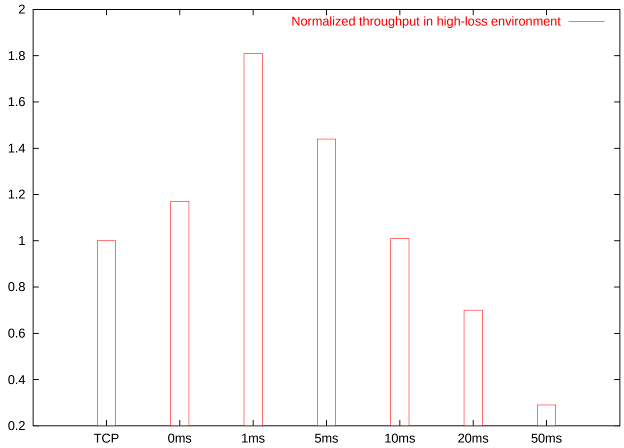

Figure 6. Normalized session throughput of 200 clients retrieving a 4KB page from a web server in the high-loss environment

small amount of delay can be tolerated, and is in fact beneficial, when packet loss rates are high.

Figure 5 shows the aggregate throughput of the 200 web clients retrieving a 4K byte web page in the low-loss environment. Throughputs of concast-supported multiplexing are normalized against standard TCP. We ran experiments with fixed maximum delays of 0ms and 1ms, as well as the 5ms, 10ms, 20ms, and 50ms delays proposed in [2]. In the case of 0ms delays, ACK packets requested a maximum delay of 0, implying that only packets already in the queue are combined (via the queue operations described earlier).

Figure 5 confirms our hypothesis that in a low-loss environment, the additional delay incurred by ACK packets can actually degrade performance rather than enhance it. In particular, a 10 ms delay decreases the throughput by more than 20%. On the other hand, concast-supported multiplexing with no delay shows no reduction in throughput.

Figure 6 shows the performance in a high-loss environment. As expected, the multiplexing approach with small delays works well in such environments, combining ACKs and thereby reducing the packet loss rate. In this case, the benefits of fewer retransmissions far outweigh the added delay. However, for applications that require minimum delay (which is exemplified by the zero delay requests), much of the benefit of multiplexing can be achieved without violating the delay constraints requested by the application.

Table 1 shows the percentage of the total number of ACKs that were combined by each approach. Note that concast-supported multiplexing is able to combine as much as 41% of the ACKs even when using zero delay.

Delay (in ms)	0	1	5	10	20	50
Low-loss	2%	77%	97%	98%	99%	99%
High-loss	41%	86%	97%	99%	99%	99%

Table 1. Percentage of the total ACKs combined for different merging scenarios

6. Conclusions

Services that can be customized through a programming interface are likely to play a key role in future networks, although the forms of programmability that will be important are yet to be determined. In this paper we have shown how to build a generic multiplexing service using a programmable concast service. Our concast-based multiplexing service allows applications to control the delay characteristics of the service, achieving the benefits of multiplexing without violating the delay constraints of the application. The service automatically deploys functionality to exactly those nodes that need it and becomes transparent to the application once it has been invoked. The service can be deployed incrementally in the Internet and offers benefits even when partially deployed.

References

[1] Combining Packets - discussion on the end2end mailing list, December 1998. archived at ftp://ftp.isi.edu/end2end/.

[2] B. Badrinath and P. Sudame. Gathercast: The Design and Implementation of a Programmable Aggregation Mechanism for the Internet, April 1999. (submitted for Publication).

[3] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stem, and R. Katz. TCP Behavior of a Busy Internet Server: Analysis and Improvements. In *Proceedings of the INFOCOM '98 Conference*, pages 252–262, March 1998.

[4] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proceedings of the ACM SIGCOMM'99 Conference*, August 1999.

[5] K. Calvert, J. Griffioen, A. Sehgal, and S. Wen. Concast: Design and implementation of a new network service. In *Proceedings of 1999 International Conference on Network Protocols, Toronto, Ontario*, November 1999.

[6] Ken Calvert, Jim Griffioen, Billy Mullins, Amit Sehgal, and Su Wen. Implementing a Concast Service. In *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, September 1999.

[7] Michael Hicks, Pankaj Kakkar, T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks, 1998.

[8] k. claffy, G. Miller, and K. Thompson. The nature of the beast: Recent measurements from an internet backbone. In *Proceedings of INET '98 Conference, Geneva, Switzerland*, July 1998.

[9] Sneha Kumar Kasera, Supratik Bhattacharyya, Mark Keaton, Diane Kiwior, Jim Kurose, Don Towsley, and Steve Zabele. Scalable Fair Reliable Multicast Using Active Services. *IEEE Network Magazine*, February 2000.

[10] S. Kent and R. Atkinson. Security architecture for the internet protocol, November 1998. Internet Request for Comments 2401.

[11] S. Keshav. Packet-Pair Flow Control. *IEEE/ACM Transactions on Networking*, February 1995.

[12] L. Lehman, S. Garland, and D. Tennenhouse. Active Reliable Multicast. In *Proceedings of the INFOCOM Conference*, March 1998.

[13] H. Nielsen, J. Gettys, A. Baird Smith, E. Prud'hommeaux, H. Lie, and Chris Lilley. Network Performance Effects of HTTP/1. CSS1, and PNG. In *Proceedings of the SIGCOMM '97 Conference*, September 1997.

[14] S. Paul, K. Sabnani, J. Lin, and S. Bhattacharyya. Reliable Multicast Transport Protocol (RMTP). *The IEEE Journal on Selected Areas of Communication*, 1996. (see also the Proceedings of IEEE INFOCOM'96).

[15] B. Schwartz, A. Jackson, W. Strayer, W. Zhou, R. Rockwell, and C. Partridge. Smart Packets for Active Networks. In *1999 IEEE Second Conference on Open Architectures and Network Programming*, pages 90–97, March 1999.

[16] P. Sudame and B. R. Badrinath. Transformer tunnels: A framework for providing route-specific adaptations. In *Proceedings of the USENIX Annual Technical Conference*, pages 191–200, June 1998.

[17] The MASH Research Team. The ns network simulator. `http://www-mash.cs.berkeley.edu/ns`.

[18] J. Touch. TCP Control Block Interdependence, April 1997. RFC 2140.

[19] D.A. Wagner and S.M. Bellovin. A Bump in the Stack Encryptor for MS-DOS Systems. In *The Proceedins of the 1996 Symposium on Network and Distributed Systems Security (SNDSS'96)*, 1996. http://bilbo.isu.edu/sndss/sndss96.html.

[20] D. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH'98*, San Francisco, CA, April 1998.

[21] R. Yavatkar, J. Griffioen, and M. Sudan. A Reliable Dissemination Protocol for Interactive Collaborative Applications. In *The Proceedings of the ACM Multimedia '95 Conference*, pages 333–344, November 1995.

[22] E. Zegura and K. Calvert. Georgia Tech Internet Topology Models. http://www.cc.gatech.edu/projects/gtitm.