Gathercast: An efficient multi-point to point aggregation mechanism in IP networks

B. R. Badrinath Pradeep Sudame

Department of Computer Science
Rutgers University
Piscataway, NJ 08855
{badri,sudame}@cs.rutgers.edu

July 29, 1998

Abstract

IP multicast is an efficient point to multi-point protocol. However, a number of scenarios exist where the reverse mechanism of multi-point to point protocol is needed. *Gathercast* is a mechanism by which packet flows from a large number of nodes can be gathered as the packets are routed towards the *gatherer*. Gathercast uses transformer tunnels [1] at routers to provide a mechanism for aggregating packets directed towards a node. Increased goodput for applications that need flow aggregation can be obtained by using any of a number of transformation functions such as recombination of small packets, compressing data, rate control, replication removal, and so on, along a distribution tree. Gathercast works well in conjunction with the current IP multicast model. Simulations results for gathercast show that using aggregators along a gather tree improves the network utilization.

1 Introduction

The number of end-user devices reachable by the Internet is increasing at a tremendous rate. Scalable mechanisms for not only distributing data but also for gathering data from such devices are needed. IP Multicast [2] provides an efficient mechanism for point to multi-point packet distribution such as multimedia communication and group communication [3]. In this paper, we propose the concept of *gathercast*, an efficient mechanism for large scale gathering of data from a large set of nodes. Applications that need such a mechanism include sensor networks, smart spaces, responsive environments [4], situation awareness (where soldiers periodically send update messages about their location), remote utility metering, web servers [5], cache state from cache servers, gathering web (Nielsen) ratings to determine clicking habits of users, and monitoring the status of devices at home or office [6]. In all these situations, we have the problem of a large number of nodes sending small updates to another node (*a gatherer*) that gathers or monitors information in such networks. These updates may be periodic, spontaneous, or in response to a query from the gatherer.

Even in today's networks, in the context of web cache updates, there is evidence of *small* packet traffic originating from a set of nodes to another node [7]. Also, the packet trace at MIT [8] shows that *replicated* packets (packets with identical payloads) form a significant fraction of the total number of packets in a network. Thus, a scalable mechanism for gathering information sent by a large number of entities, without overwhelming the receiver and the network, is needed. Just as in IP multicast where packets are copied as needed, in gathercast we need mechanisms to reduce the packet traffic towards a particular node by eliminating redundancy and by combining small packets.

For this purpose, we propose the concept of *gathercast*, a new protocol for efficient multi-point to point aggregation. IP multicast is an efficient point to multi-point protocol for the problem of how to reach a number of receivers efficiently (increasing goodput) without having to use multiple unicasts. Gathercast, on the other hand, addresses the reverse problem: how to gather information from a large number of nodes with the objective of increasing goodput.

We have already developed the concept of transformer tunnels [1]. A transformer tunnel can be programmed to optimize a flow for a given link. In gathercast, we propose to use transformer tunnels in the context of IP multicast where transformer tunnels are established over paths leading to the gatherer. We call the corresponding tree a *gathercast tree* or simply a *gather tree*. We attach various transformation functions such as *reassembly, compression, rate control,* and *replication removal* to transformer tunnels. The reassembly function combines small packets to form a larger packet. This reduces the number of packets within the network and also reduces load at the receiver by reducing the number of interrupts generated. The replication-removal function [8] reduces redundancy in the data packets flowing over a link by sending identical payloads only once across the link.

This paper is organized as follows. Section 3 describes what aggregation functions are meaningful in the context of gathercast. Section 4 gives the details of the gathercast protocol. Section 5 describes how gather trees can be used in the context of multicast trees. Section 6 explains how we evaluated the gathercast protocol. The paper concludes with a description of possible extensions of this work.

2 Related Work

There are mechanisms that require network participation for increasing the goodput. For example, many reliable multicast schemes [9, 10, 11, 12] propose suppressing NACKs to solve the problem of NACK implosion. In such schemes, if a node within the network receives identical NACKs from multiple nodes, it forwards just one NACK to the source.

The Cu-SeeMe project at the Cornell University [13] advocates the concept of reflectors which are placed throughout the network. Loss reports are sent back up the distribution tree via the reflectors. Reflectors can also collapse loss reports.

Gathercast provides a generic mechanism for aggregating data from a number of sources, and allows applications to use either multicast or unicast as the underlying transport mechanism.

Active networks [14] suggest use of internal nodes for combining information from multiple nodes (*fusion*). Gathercast is a special case of such active networks where programmability of the internal nodes is achieved using transformer tunnels.

3 Aggregators

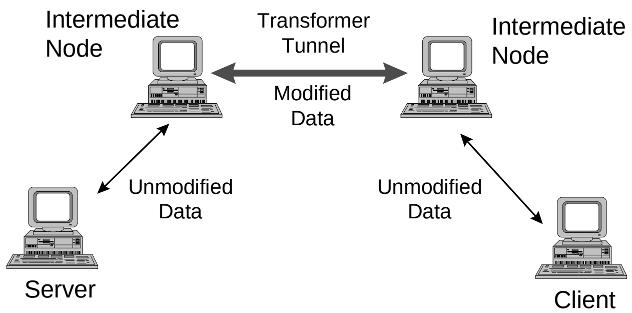

Figure 1: Transformer tunnel

Transformer tunnels provide a uniform mechanism to transform packet flows in a network. Packet flow over a network segment is modified by placing a transformer tunnel between two end nodes of the segment. Packets entering the tunnel are modified — either by changing the content, or by changing the way they are transmitted — to fine tune the flow according to the segment properties.

Original packets are restored at the other end of the tunnel. Figure 1 shows how packets undergo transformations at the end points of the tunnel.

The transformations that are performed depend on the transformation functions attached to the tunnel. In the transformer tunnel mechanism, reverse transformations are performed at the destination end of the tunnel. However, for gathercast, we need to perform the reverse transformations (such as regenerating the packets from the reassembled packets, decompressing packets, and so on) only at the gatherer. Therefore, we need to modify the reverse transformation functions. These modified functions perform a reverse transformation only if the destination of the transformed packet is same as the original destination of the packet. (Information about original destination of the packet can be found out from the metadata in the transformed packets that is also sent along with the packet.) Thus, the reverse transformations will be the identity function (that returns the packet without any changes) at the intermediate nodes, and the actual reverse function at the final destination.

In the context of gathercast, we call the transformation functions *aggregators*. This sections describes various aggregators that can be used to increase the goodput of gathercast. Table 1 gives some examples of of useful aggregators. We have already implemented the compression and the reassembly transformation function. Other functions are yet to be implemented in the transformer tunnel framework.

Aggregator	Where to place	Impact
Compression	Close to the senders	reduces number of bytes for compressible (large) packets
Reassembly	Throughout the network	reduces number of packets for small packets
Replication removal	some intermediate nodes	saves bandwidth if payloads are replicated
Rate control	Close to the receiver	Reduction in peak rate at the receiver

Table 1: Aggregators

3.1 Compression

On slow links, the time for transmitting packets is large and compressing packets before transmission improves performance. Header compression techniques [15, 16] have been proposed to improve performance for such links. On very slow links, compressing the data portion leads to even more gains [17, 18]. Compressing data is not useful on fast links, because the compression and decompression overheads offset the savings obtained by sending fewer bits. On slow links, however, a fast compression function (where the time per byte for compression and decompression multiplied by the bandwidth is less than the fraction of bytes saved) leads to improved performance.

We have used a simple compression function provided by the minilzo library [19]. If the packet is incompressible, we send the original packet without any modifications. The LZO compression method is fast enough to be useful even on a 2-Mbps WaveLAN. For wide-area networks (where gathercast will typically be used), compression will lead to more gains.

Once a packet is compressed, it cannot be compressed further. Therefore, we should compress packets at the senders (by using a transformer tunnel at the senders). The packets can be decompressed at the receiver. All the intermediate links will therefore see a reduction in number of bytes transmitted over them.

3.2 Reassembly function

The reassembly function uses a mechanism similar to TCP delayed ACKs. Every small packet is delayed by a small amount of time. If another small packet arrives in this interval, both the packets are combined. Otherwise, the packet is sent as is. This reassembly mechanism is different from IP reassembly (performed after IP fragmentation). A reassembly function combines small packets, along with their IP headers, to get a larger packet. The mobile host regenerates all the original packets when it receives such a reassembled packet. Thus, this mechanism can be used even for protocols that honor message boundaries (for example, UDP).

We use the reassembly algorithm shown in Figure 2. A reassembly tunnel maintains a single packet buffer for every flow that needs this adaptation. The maximum time for which packets are delayed is a parameter that can be set by applications. The applications also have to decide the maximum delay it can tolerate. A larger delay value increases the probability that two small packets will be combined to create a larger packet. It is possible to extend the strategy to combine more than two packets if more delay is tolerable. To simplify the implementation we combine at most two packets at a time.

As small packets are delayed for some time in the hope that they can be combined with other small packets, they see increased latency. In the best case, when every two packets are combined, every alternate packet sees an increase in the latency equal to the inter-arrival time. In situations where no packets can be combined because the combined packets are larger than the link MTU, the packets experience the latency equal to the packet inter-arrival time. In case when the timeout

```
Initialization code
    delayed_packet = $null$;

reassemble (p) {
    if (p is large) {
        send delayed_packet; send p;
        delayed_packet = $null$;
        return PACKET_SENT;
    }
    if (delayed_packet exists) {
        if (p and delayed_packet can be combined) {
            combine and send;
            delayed_packet = $null$;
            return PACKET_SENT;
        } else {
            send delayed_packet;
            delayed_packet = p;
            set timer; return PACKET_SENT;
        }
    } else {
        delayed_packet = p;
    }
}

Timer expires:
    send delayed_packet; delayed_packet = $null$;
```

Figure 2: Reassembly algorithm

is smaller than the packet inter-arrival time, every packet sees a latency equal to the timeout. Large packets do not see any increase in the latency as they are never buffered by the reassembly function. Therefore, the reassembly function increases latency experienced by small packets, but allows a more efficient use of the bandwidth by potentially reducing the number of small packets flowing in the network. It also improves the gatherer performance by reducing the number of interrupts the gatherer receives because of incoming packets.

3.3 Replication removal

The MIT trace [8] shows that in today's networks, there is a significant number of replicated packets. Since all the replicated packets carry identical payloads (and hence have the same md5 fingerprint), payload needs to be sent only with the first packet. For every new packet with the same md5 fingerprint, only the fingerprint needs to be sent. The receiver can recover the original packet by reinstalling the payload from its cache. Since, such a mechanism needs a large amount of memory in the nodes, it should probably be performed only at a few nodes that are not heavily loaded. If such a function is placed near the leaves of the tree, there may not be a large number of replicated packets. Therefore, we need to find out nodes near the gatherer that are not be heavily loaded. A potential choice is the next-hop routers for the gatherer.

3.4 Rate control

In a gather tree, every sender may send data at regular intervals. For example, devices that monitor external conditions may be reporting data every five minutes. However, at the receiver, the packets may arrive in bursts, because a large number of devices may send data at approximately the same time. Therefore, even when the leaves generate packets at a regular intervals, the gatherer may see bursts of incoming packets.

Bursts are not a great problem for the intermediate routers as the only processing they do is to forward the packets. For the gatherer, however, it may pose a serious problem as all these packets may have to be processed. This may increase the peak load on the gatherer. A rate control transformation function can buffer packets close to the gatherer, and can pace them at regular intervals thereby reducing the peak rate seen by gatherer.

4 Gathercast

Gathercast uses transformation functions in a hierarchical fashion so that packets are aggregated at various levels of the gather tree. Figure 3 shows how such a hierarchical application of the reassembly function can reduce the number of packets received by the receiver. Node 6 in the figure combines three packets, one from each of its children. Node 4 combines two packets from one child and one packet from another child. Depending on the applications that may use gathercast, we may have to decide the exact way of combining packets. Packets can be combined at various levels of the tree. In rest of the paper, we explain the gathercast protocol by use of reassembly transformation function as the only aggregator being used at the intermediate nodes.

- **Combining close to the gatherer only (NEAR_ROOT):** If a reassembly function is placed very close to the root of the tree, (for example in a core based tree [20] we can have a

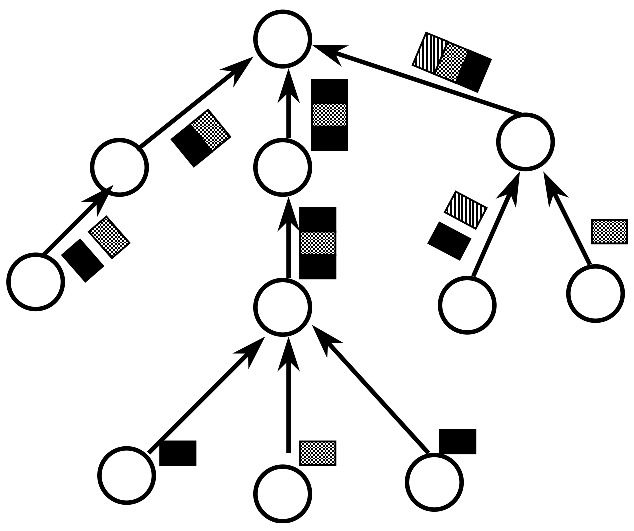

Figure 3: Gather Tree

transformer tunnel from the rendezvous point to the receiver), it will achieve reassembly with low latency. Unfortunately, this means that rest of the network will not see any advantages of reassembly. Packets will be combined only on the link directly attached to the root of the tree. Thus, such a strategy keeps the latency reasonably low but does not provide a great improvement in goodput, especially if the tree if very skinny with a number of hops to the leaves.

- **Combining close to the leaves only (NEAR_LEAVES):** We can also perform reassembly on the routers that are one hop away from the leaves (the senders). With such a strategy, single link subtrees may not be able to combine packets. For balanced trees, this strategy provides a great improvement in goodput, especially when the packet generation rate at the leaves is high.

- **Combining in the middle of the tree (HALF_WAY):** A naive strategy of combining packets in the middle of the tree is a compromise between the two strategies mentioned above. Moving away from the receiver increases the number of small packet seen by the combining node; whereas moving away from the root reduces number of packets seen by the nodes between the combining nodes and the root.

- **Combining throughout the network (ALL_NODES) :** In another strategy, we can have a transformer tunnel at every router. Such a tunnel can be established for a given gatherer and a group. Such a collection of tunnels is very effective in combining small packets greedily *i.e.*

small packets are combined as soon as possible. However, a packet will encounter a number of transformer tunnels along its way to the destination, and delays introduced by each tunnel will add up. Therefore, in the worst case, there may be a large latency for packets.

- **Combining along a gathercast tree (DYNAMIC):** If we can get rid of the tunnels (from the naive strategy described above) that add to the latency without providing any useful transformations, we can reduce the latency considerably. We can use a simple strategy to drop such useless tunnels. We establish the tunnels all over the network. Later, tunnels that cannot perform reassembly for k consecutive packets (k is another parameter that can be set by applications) are dropped. The root can periodically (with a longer period) reinstall all the tunnels — maybe when the topology changes significantly. Larger period for reinstalling the tunnels implies longer time for adjusting to the changes in the network; whereas smaller period implies more overheads, because it takes k packets to drop the useless tunnels and these k packets see a marked increase in the latency.

  Once the useless tunnels are dropped, the gather tree will have tunnels only at those places where they are useful.

	Unicast Routing	Reverse-multicast Routing
Tunnels establishment Protocol	Complex, needs more messages	Simple
Routing	Simple and efficient	Inefficient, requires encapsulation
Deployment	Requires intermediate routers to intercept packets	Requires changes to mrouted

Table 2: Two alternative routing strategies for gathercast

5 Multicast Tree and Gathercast Tree

When the leaves send unicast packets to the root of the gathercast tree, the packets follow unicast routes, which are not necessarily same as multicast routes. Therefore, there are two alternatives: first, to establish tunnels along the multicast tree and force the packets to follow the reverse-multicast routes; and second, to let the packets follow the unicast routes and establish tunnels on the routers along those routes. This section describes both these alternatives in detail. Table 2 gives a brief comparison of these two alternatives. In both the alternatives, we assume that the gatherer decides when to establish the tunnels and controls the tunnels establishment procedure. Table 2 compares the the two alternative approaches.

5.1 Gathercast tree independent of the multicast tree (Unicast routing)

In this scheme, the senders send unicast packets to the receiver. Therefore, routing is optimal as long as the unicast routing tables are optimal. Moreover, no encapsulation is needed for routing. However, the routes from the senders (leaves) to the gatherer need not overlap. In such cases, the transformer tunnels may not be of any help at all. Also, this scheme requires that intermediate routers check packets contents (such as UDP port number as used by RSVP tunnels [21]) which leads to excessive overheads at all the routers.

5.1.1 Tunnel establishment

To establish tunnels along the network, the gatherer sends a multicast message to the leaves. The message contains a unique id and all the parameters required to establish the tunnels such as timeouts and time to live values. The message is sent with the protocol IPPROTO_XFORM (the same protocol that is used by transformer tunnels for transformed packets). This is achieved by establishing a transformer tunnel at the gatherer with source as the gatherer and the point-to-point address of the tunnel as the multicast address of the group. The message contains *metadata* indicating that the receivers should just bounce the packet back to the sender. The reverse function works as follows: at intermediate routers it just ignores the packet; at leaves, it bounces the packet back to the sender. This is required to ensure that these messages follow the same unicast routes (from leaves to the root of the gather tree) as used by the data packets. While bouncing the packet, the receivers use IP+UDP encapsulation similar to that used by RSVP tunnels [21]. All the routers along the path then intercept this packet. The routers that do not support transformer tunnels just forward the packet. Routers that support transformer tunnels check if the packet with the same id was received recently. If it has received such a packet, it just drops the packet (so that duplicate request are not propagated up the tree), otherwise it establishes a tunnel with itself as the source end and the root of the tree as the destination end. (This ensures that all reverse transformations of regenerating original packets from the reassembled packets, and generating the replicated packets from the fingerprints are performed only at the receiver.) The intermediate routers then forwards the packet up the stream.

5.1.2 Routing using unicast routes

Once the tunnels are established, transformations are performed as if dealing with unicast networks. No special code for routing is required. Gathercast just uses the underlying IP routing. The packets pass through the transformer tunnels at intermediate routers that have established transformer tunnels using the above mechanism.

5.2 Transformer tunnels along the multicast tree (Reverse-multicast routing)

Multicast trees have naturally overlapping routes. Therefore, transformer tunnels along the multicast trees have more opportunities to combine packets. Moreover, the multicast routing daemon, *mrouted*, is still in experimental stages and can therefore be modified. This allows incremental deployment. Unfortunately, such a scheme requires encapsulation for unicast packets (so that they follow the reverse-multicast routes). This leads to encapsulation overheads which may undermine the gains of combining small packets. However, if the reassembly function is smarter (for example, combining two NACKs for a reliable multicast gives a packet much smaller than just putting two packets together), it may still be able to improve the network utilization. Another disadvantage of this scheme is that it may force packets to follow nonoptimal routes if the reverse-multicast routes are nonoptimal.

5.2.1 Tunnel establishment

For establishing the tunnels, the gatherer sends a new IGMP message to the multicast group. *mrouted* (which has to be modified for this) interprets this message and sets up a tunnel (or informs the tunnel manager that such a request has arrived). The source end of the tunnel is the node receiving this packet and the destination end is the previous-hop router, which can be obtained from the IP-multicast routing table. (Such a previous-hop information is also used by tools like *mtrace*).

5.2.2 Routing using reverse-multicast routes

The main problem, however, is to route packets using the reverse-multicast routes. We use transformer tunnels for this purpose also. The transformation function attached to the tunnel encapsulates the packets in a multicast message with TTL value 1 (same as for join messages used by the DVMRP [22] and PGM [12]). The tunnel also adds metadata to the packet indicating a special

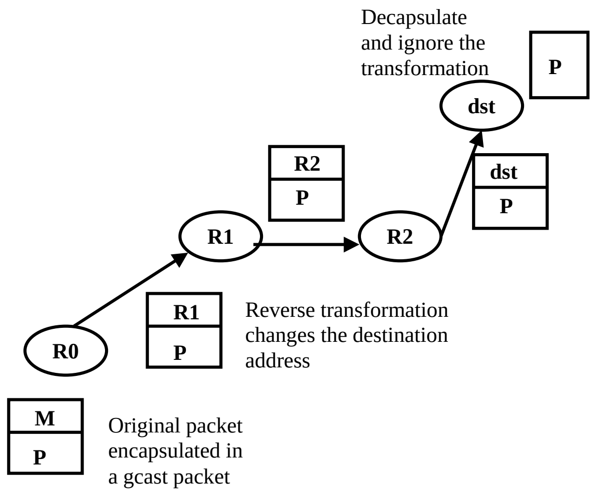

Figure 4: Reverse-multicast routing

reverse transformation function, *multicast-lookup* (that looks at the multicast routing table at the receiver and find out the previous-hop router), should be performed by the receiver of this packet. This message is received by the nearest multicast router. The multicast router then decapsulates the message and applies the multicast-lookup transformation. The multicast-lookup function finds out the previous-hop router (that is the router that would send the messages from the gatherer) from the multicast routing table. It then encapsulates the original message as a transformed packet to the new router and again adds metadata so that the next router will also look into its own multicast routing table to find out the previous-hop router. This process repeats at every router and the packet finally reaches the root. The destination just ignores the reverse transformation function and does not perform any multicast routing table lookup. Figure 4 shows how packets are routed using this protocol.

Unlike other transformation functions such as reassembly, replication-removal, and so on (which are ignored by the routers and interpreted by the final destination), the multicast-lookup function is interpreted by the routers and ignored by the final destination node. The check for final destination is performed by the transformation function itself and no changes to the transformer tunnel code are necessary.

5.3 Signaling

The gathercast protocol introduces some extra signaling packets in the network. These extra packets are used while establishing and refreshing the transformer tunnels. There are various ways in which signaling can be controlled.

- **Minimal signaling:** Each router can decide default parameters and no signaling is required once the tunnels are setup. For example, the reassembly function can decide the timeout based on the rate of incoming packets. This way, the parameters are controlled by the data packets and are transparent to the applications.

- **Medium signaling:** The gatherer can send information about how to set the parameters. This information can be sent to nearby routers when establishing the tunnels. Rest of the routers in the network can use default parameters. In this case, the parameters for nearby hosts are controlled by applications using gathercast; whereas the parameters for rest of the routers are controlled by the data packets.

- **Heavy signaling:** At the other extreme of this spectrum, the gatherer can send parameters to all the routers in the network. The application can therefore control parameters at all the routers in the network.

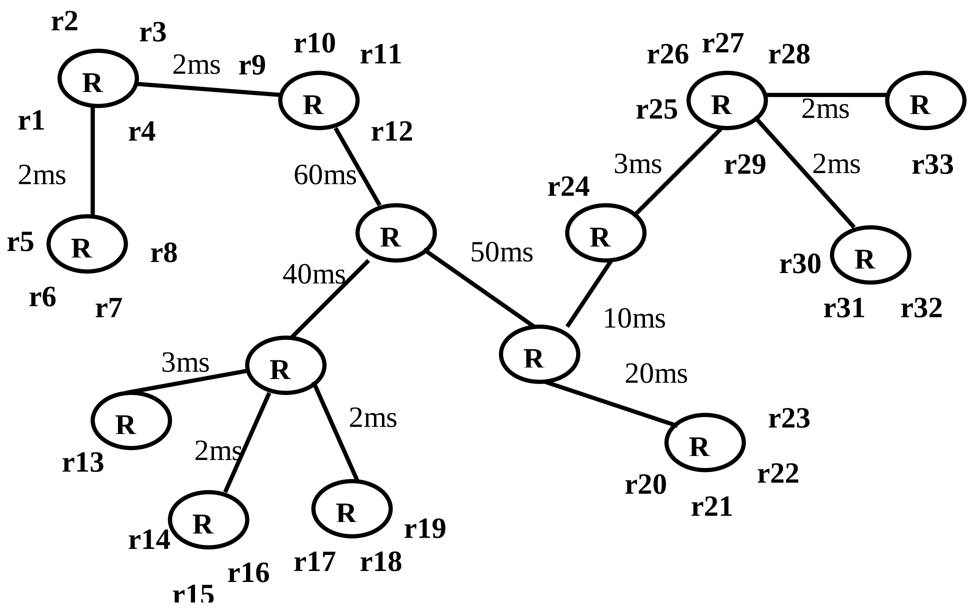

Figure 5: Imaginary WAN topology

6 Evaluation

For evaluating the impact of gathercast, we used three different tree configurations: a complete binary tree with fifteen nodes, a complete binary tree with 127 nodes, and a tree (given in the paper by Papadopoulos et al. [11].) for simulating a WAN topology. Figure 5 shows the WAN topology

Combining heuristics	Avg Delay Sec	Avg Savings pkts/node [Deviation]	Pkts at Root
ALL_NODES	7.5	102 [3.9]	453
NEAR_ROOT	5.7	71 [26.3]	669
NEAR_LEAVES	1.2	17 [15.6]	1052
HALF_WAY	5.7	71 [2.8]	669
DYNAMIC	5.3	77 [9.0]	631

Table 3: Gathercast savings (Uniformly distributed inter-arrival times, 16 node binary tree, 1168 total packets)

Combining heuristics	Avg Delay Sec	Avg Savings pkts/node [Deviation]	Pkts at Root
ALL_NODES	6.6	100 [2.5]	2317
NEAR_ROOT	1.3	102 [49.4]	2194
NEAR_LEAVES	1.8	2 [1.8]	8480
HALF_WAY	4.0	80 [10.1]	3581
DYNAMIC	5.3	86 [5.2]	3202

Table 4: Gathercast savings (Uniformly distributed inter-arrival times, 128 node binary tree, 8595 total packets)

used for simulation. The routers are shown as **R** and leaves are shown as **r** in the figure. For evaluation purposes, we generated packets using two methods. In the first method, packets were generated at every leaf of the tree at an average rate of once every five minutes. Such a distribution represents a sensor network application where the devices send status report periodically. In the second method, we used a packet trace obtained from University of California at Berkeley [5]. This trace represents packets generated by various clients. The trace was collected at web proxies. In both the methods, every node delayed packets by at most 10 seconds. For the DYNAMIC strategy explained in Section 4, we reinstall tunnels every 600 seconds and we drop tunnels at nodes that cannot combine 5 consecutive packets. These values can be sent as parameters to the nodes performing the aggregation function.

We collected the statistics about the number of packets that were combined, and the delay seen by the packets. The data was gathered for one simulated hour in every case. For each tree, we also studied five heuristics for combining packets: combine packets at all the nodes (ALL_NODES), combine packets at one level above the leaves (NEAR_LEAVES), combine near the root of the tree (NEAR_ROOT), combine at nodes at depth equal to half the depth of the tree (HALF_WAY), and combine using the dynamic scheme suggested in Section 4 (DYNAMIC).

Tables 3, 4, 5, 6, 7, and 8 show the results we obtained. A smaller value for packets at the

Combining heuristics	Avg Delay Sec	Avg Savings pkts/node [Deviation]	Pkts at Root
ALL_NODES	2.9	197 [41.0]	1742
NEAR_ROOT	6.4	153 [147.3]	2310
NEAR_LEAVES	5.6	11 [8.1]	4157
HALF_WAY	2.1	117 [62.9]	2774
DYNAMIC	2.6	171 [44.4]	2081

Table 5: Gathercast savings (Uniformly distributed inter-arrival times, WAN topology, 4300 total packets)

Combining heuristics	Avg Delay Sec	Avg Savings per node [Deviation]	Pkts at Root
ALL_NODES	0.4	2148 [613.3]	1963
NEAR_ROOT	0.2	2150 [744.1]	1950
NEAR_LEAVES	0.0	415 [390.8]	14096
HALF_WAY	0.2	2150 [744.1]	1950
DYNAMIC	0.3	1991 [656.8]	3060

Table 6: Gathercast savings (Packet trace, 16 nodes binary tree, 17000 total packets)

Combining heuristics	Avg Delay Sec	Avg Savings per node [Deviation]	Pkts at Root
ALL_NODES	0.5	1713 [229.3]	13807
NEAR_ROOT	0.0	1716 [836.2]	13588
NEAR_LEAVES	0.0	19 [19.0]	120525
HALF_WAY	0.2	1713 [379.6]	13805
DYNAMIC	0.2	1700 [229.6]	14580

Table 7: Gathercast savings (Packet trace, 128 node binary tree, 121725 total packets)

Combining heuristics	Avg Delay Sec	Avg Savings per node [Deviation]	Pkts at Root
ALL_NODES	0.4	3517 [778.3]	13213
NEAR_ROOT	0.6	2689 [746.9]	23981
NEAR_LEAVES	0.0	390 [303.4]	53870
HALF_WAY	0.1	2583 [1190.7]	25351
DYNAMIC	0.1	2576 [724.3	25446

Table 8: Gathercast savings (Packet trace, WAN topology, 58935 total packets)

root implies that more packets could be combined within the network and hence the network utilization improved. Average savings are the average number of packets that were combined per node. A higher deviation for savings implies that the savings came from a small number of nodes (for example, when using the NEAR_ROOT strategy). A smaller deviation implies that more nodes participated in the savings and hence a larger part of the network sees the benefits due to a reduction in the number of packets.

We see that combining near the leaves has the least impact on the network utilization. However, when the inter-arrival time for packets becomes smaller than the timeout values used by gathercast, such a strategy will give a big reduction in the number of packets flowing in the network. Combining packets only near the root of the tree reduces the number of packets seen by the root. However, rest of the network still sees a large number of packets. (Deviation is high in this case.) Combining packets at all the nodes of the tree gives a large reduction in the number of packets seen by the root. However, the buffering at every node increases the delays seen by packets. The dynamic scheme achieves a balance between the delay and the network utilization. In the dynamic scheme, the nodes that cannot combine packets drop the reassembly tunnels placed at them. Therefore, the tree has tunnels only where they are useful. This reduces the unnecessary delay seen by the packets. Another interesting point is that the average delay seen by the packets is far smaller than the worst case delay (where every node in the path introduces a delay of 10 seconds).

Work on evaluation of gathercast with different packet generation methods and for different aggregators is in progress.

7 Extensions and Future Work

We have already implemented the reassembly and the compression transformation function in the transformer tunnel framework. We need to implement the rate-control and the replication-removal function in the same framework. We also need to provide application-specific reassembly functions. For example, two NACKs for the same packet can be combined into one NACK. We also need to use the composition of aggregators. Such a composition is allowed by transformer tunnels. We need to study its impact on gathercast.

For the reassembly function, we need to learn the Maximum Transfer Unit (MTU) dynamically. We also need to figure out how to decide the timeouts, and what impact do such timeouts have on the applications that use gathercast. We also need to decide the combination policy and its impact on applications. For example, following are a few choices.

- Combine K packets (have a common buffer and combine K packets, or $M < K$ packets in the buffer when the timeout occurs).
- Combine one packet per active incoming interface (wait until we have one packet from each of the incoming links or for a timeout, whichever is earlier).
- Combine at a rate proportional to the active incoming interface (A fast incoming link will have more number of packets rather than a slow incoming link).

We also need to decide where to combine packets. We can have an *aggressive* policy where as many packets are combined as possible at lower levels of the gather tree, or a *progressive* policy where two packets are combine at a time along the path of the tree, or *lazy* policy where packets are combined very close to the gatherer only. We need to decide how applications are affected by such policies and whether different applications need different policies.

8 Conclusions

In many applications, such as sensor networks, smart spaces, responsive environments, and so on, a large number of nodes send small packets to a single destination. As opposed to IP multicast, such applications need an efficient mechanism for multi-point to point aggregation. We propose a new protocol, *gathercast*, that allows such efficient aggregation. We also describe various aggregators that can be used in the context of gathercast. We describe how aggregators can be placed at various nodes in the network and how transformer tunnels can be used to perform aggregations. We also suggest various strategies to decide where to place aggregators in the network, and compare the strategies using simulations.

Our trace-driven simulations show that gathercast provides an efficient aggregation mechanism for a class of applications that need to collect data from a number of nodes. The simulation results indicate that gathercast reduces the number of small packets in the network.

We plan to investigate the impact of gathercast on next-generation applications such as smart spaces and sensor networks. We believe that gathercast will allow such applications to use the network efficiently.

References

[1] P. Sudame and B. R. Badrinath, "Transformer tunnels: A framework for providing route-specific adaptations," in *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[2] S. Deering, *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, Dec. 1990. ftp://gregorio.stanford.edu/vmtp-ip/sdthesis[123].ps.Z.

[3] N. L. for Applied Research, "ICP working group." http://ircache.nlanr.net/Cache/ICP/.

[4] S. Elord, G. Hall, R. Costanza, M. Dixon, and J. des Rivieres, "The responsive environment," Tech. Rep. CSL-93-5, Xerox Park, June 1993.

[5] S. Gribble, "UC Berkeley Home IP Web Traces." http://www.cs.berkeley.edu/˜gribble/traces/index.html.

[6] V. Cerf, "Keynote address." Infocom 97, Kobe, Japan, March 1997.

[7] L. Fan, P. Cao, and J. A. anndrei Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol." To appear in SIGCOMM'98.

[8] J. Santos and D. Wetherall, "Increasing effective link bandwidth by suppressing replicated data," in *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[9] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," in *Proceedings of the ACM SIGCOMM*, pp. 342–356, Oct. 1995.

[10] L. wei H. Lehman, S. J. Garland, and D. L. Tennenhouse, "Active reliable multicast," in *Proceedings of the INFOCOM*, 1998.

[11] C. Papadopoulos and G. Varghese, "An error control scheme for large-scale multicast applications," in *Proceedings of the INFOCOM*, 1998.

[12] T. Speakman, D. Farinacci, S. Lin, and A. Tweedly, "PGM Reliable transport protocol specification," 1998. Internet draft, Work in progress.

[13] "CU-SeeMe Project." ftp://gated.cornell.edu/pub/video.

[14] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. Minden, "A survey of active network research," in *IEEE Communications*, Jan. 1997.

[15] M. Degermark and S. Pink, "Soft state header compression for wireless networks," in *Proceedings of the 2nd MOBICOM Conference*, pp. 1–14, Nov. 1996.

[16] V. Jacobson, "RFC 1144: Compressing TCP/IP headers for low speed serial links," 1990.

[17] A. Sacham, R. Monsour, R. Pereiera, and M. Thomas, "IP payload compression protocol (IPComp)," Oct. 1997. Internet draft, Work in progress.

[18] V. Schryver, "RFC1977: PPP BSD compression protocol," Aug. 1996.

[19] M. F. X. J. Oberhumer, "miniLZO – mini version of the LZO real-time data compression library." http://www.infosys.tuwien.ac.at/Staff/lux/marco/lzo.html.

[20] A. Ballardie, "RFC 2201: Core Based Trees (CBT) multicast routing architecture," 1997.

[21] A. Terzis, L. Zhang, and E. Hahne, “Making reservations for aggregate flows: Experiences from an RSVP tunnels implementation.” To appear in IWQoS ’98, Napa Valley, CA.

[22] D. Waitzman, C. Partridge, and S. Deering, “RFC 1075: Distance vector multicast routing protocol,” 1988.