A Flexible Concast-based Grouping Service

Amit Sehgal, Kenneth L. Calvert, James Griffioen
{amit,calvert,griff}@netlab.uky.edu

Laboratory for Advanced Networking, University of Kentucky, Lexington, KY

Abstract. We present a scalable and flexible grouping service based on concast and best-effort single-source multicast. The service assigns participating end systems to specific groups based on application-supplied criteria. Example uses of such a service include peer-to-peer applications that want to group machines that are "near" each other, and reliable multicast services that need to assign receivers to repair groups. Our generic grouping framework relies on concast's many-to-one transport service to efficiently collect and apply the application-specific grouping criteria to the group members' information, and it relies on single-source multicast (i.e., one-to-many communication) to distribute the results to the nodes being grouped. The service can easily be customized to meet the grouping requirements of the application. We present simulation data showing the convergence properties of our grouping service and its effectiveness when applied to the problem of constructing overlay networks.

1 Introduction

Distributed applications often partition their member-set into groups to improve scalability and performance. In some cases this process occurs randomly, while in others it is controlled in order to ensure that nodes having similar characteristics end up in the same group. For example, the scalability of reliable multicast improves when receivers are grouped according to patterns of loss, and retransmissions are sent only to groups in which some member lost the packet [1, 2]. Peer-to-peer applications and overlays benefit from reduced latency when participants are grouped (connected to) others who are topologically "nearby".

Although examples of this kind of grouping abound, it is typically achieved through custom means; we know of no protocol designed to assist in group formation, say by assigning participants that satisfy the same application-supplied criterion to the same group. For small applications, a centralized approach to grouping is a viable solution: a single machine collects information from all participants, assigns them to their respective groups and returns the result to each participant. However, for large applications, the need to avoid implosion and reliably convey results to all participants limits the scalability of a centralized solution. Also, group formation decisions may be influenced by topology considerations. For example, the application may want to group nodes that are topologically "near" one another (for some definition of "near"). Thus, a grouping service must be able to take topology into account in some form.

In this paper we present a customizable *grouping service* that assigns network nodes to specific groups based on application-supplied criteria. The service uses single-source multicast to disseminate results, and concast to collect and process information from the participants. It is scalable because concast reduces the load on the central node and spreads the work of group formation across the network infrastructure. In addition, by processing information as it travels through the network, information about topology can be included in the process. The programmability of our service derives from that of concast, which allows an application to supply a *merge specification* describing how its packets should be combined as they travel through the network. Our grouping service is implemented as a generic merge specification plus an application-supplied data structure and three functions that operate on that data structure.

The remainder of this paper is organized as follows. Section 2 gives an overview of the concast service and Section 3 describes our grouping abstraction in detail. Section 4 demonstrates our service's applicability and applies the grouping abstraction to two problems: grouping receivers in reliable multicast and constructing topologically-efficient overlay networks. Section 5 discusses the simulation results that prove the convergence and scalability of our approach and Section 6 concludes the discussion.

2 Concast

This section provides a necessarily brief overview of the concast service and its programming interface. The interested reader is referred to [3] for a detailed description of concast.

A *concast flow* is uniquely identified by a pair (G, R) where R is the receiver's (unicast) IP address and G is the *concast group identifier*, which represents the set of senders communicating with R. Concast packets are ordinary IP datagrams with R in the destination field and G in the IP options field (in a *Concast ID* option). The IP source address carries the unicast address of the last concast-capable router that processed the packet. Concast-capable routers intercept and process all packets that use the Concast ID option.

getTag(m): A tag extraction function returning a hash or key identifying the message. Message 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 if the current merge state (i.e. information representing messages that have already been processed). The third parameter is a "flow state block" containing information about the concast flow to which m belongs

done(s): The forwarding predicate that checks s, the current merge state, and decides whether a message should be constructed (by calling buildMsg) and forwarded to the receiver.

buildMsg(s): The message construction function, which takes the current message state s, and returns the payload to be forwarded toward the receiver.

Fig. 1. Merge Specification Methods

```
ProcessDatagram (Receiver R, Group G,
                        IPDatagram m) {
    FlowStateBlock fsb;
    DECTag t;
    MergeStateBlock s;

    fsb = LOOKUP_FLOW(R,G);
    if (fsb ≠ ⊥) {
        t = fsb.getTag(m);
        s = GET_MERGE_STATE(fsb, t);
        s = fsb.merge(s,m,fsb);
        if (fsb.done(s)) {
            (s,m) = fsb.buildMsg(s);
            FORWARD_DG(fsb, s, m);
        }
        PUT_MERGE_STATE (fsb, s, t);
    }
}
```

Fig. 2. The Concast Framework

The packets delivered to R are a function of the packets sent by the members of G; the concast abstraction allows applications to customize the mapping from sent messages to delivered message(s), which is carried out hop-by-hop by the concast-capable routers along the path from senders to R. This mapping is called the *merge specification*; it controls (1) the relationship between the payloads of sent and received datagrams, (2) the conditions of message forwarding, and (3) packet identification (i.e. which packets are merged together). The merge specification is defined in terms of four methods (see Figure 1), which are invoked from a generic packet-processing loop as packets are processed hop-by-hop (see Figure 2).

The concast framework allows users to supply the definitions of these functions using a mobile-code-language. The merge specification functions are injected into the network by the receiver and pulled down into the network to the appropriate nodes by the *Concast Signalling Protocol* (CSP) when senders join the group.

Each concast-enabled router maintains the following information for each flow (G, R) in a *flow state block*, or FSB.

***Merge Specification*:** the definitions of the **getTag**, **merge**, **done**, and **buildMsg** functions.

***Per-message State List*:** A list of in-progress "merge states" indexed by message tags.

***Upstream Neighbor List* (UNL):** Each item in the UNL represents a concast-capable node or a local sender (i.e. an application on the same node) for which the current node is the next concast-capable hop on the way to R. The node processes concast messages sent by the members of this list. Every member of this list is responsible for refreshing its state to avoid being purged from the list.

Incoming packets are classified into flows based on (G, R). If no FSB is found for a packet, it is discarded. Otherwise the **getTag** function is obtained from the FSB and applied to the packet to obtain a tag that identifies the equivalence class of packets to which the packet belongs. The **merge** function is invoked on the current merge state for the tag (i.e., the merged state from all messages already received) to compute the new merge state. The **done** predicate then determines whether the merge operation is complete. If so, **buildMsg** is invoked to construct an outgoing message from the merged state that is forwarded toward the receiver.

CSP establishes concast-related state in network nodes. The protocol works by "pulling" the merge spec from the receiver towards senders as they join the group, installing concast state along the path from the sender to the receiver. All CSP message are sent as regular IP unicast messages with CSP identified in the protocol field, (G, R) in the Concast ID option field, and the IP Router Alert Option [4] to stimulate hop-by-hop processing.

3 Grouping Abstraction

Our goal is a scalable service that can be used as a building block by applications that need to partition their members into groups. An easy solution is to support a predefined set of application-independent grouping policies. For example: network hop distance can be used as the group formation criteria where participants placed are placed in the same group if they are within a certain hop-distance of each other. However, predefined criteria will not suffice for many applications. Instead these applications want to form groups based on participant-supplied criteria. Our proposed application-specific grouping framework makes the following assumptions:

– All participants provide input (data) values (of a fixed, application-determined type) to the grouping service; this information is used to map participants to groups.
– There exists a mapping from the input values to a merged value of the same type that represents the group. (i.e. the values of the participants should be *merge-able* in some way other than simple concatenation.)
– All members of the application are multicast- and concast-capable.

Figure 3(a) illustrates the abstraction of the grouping service. One participant of the application initiates the service by providing the *grouping criteria*. The framework applies this grouping criteria to all participants' data in a distributed manner. Every participant m of the application provides a value v_m and an identifier id_m as input to the grouping framework. The structure and content of v and id are defined by the application and interpreted *only* by the grouping criteria.

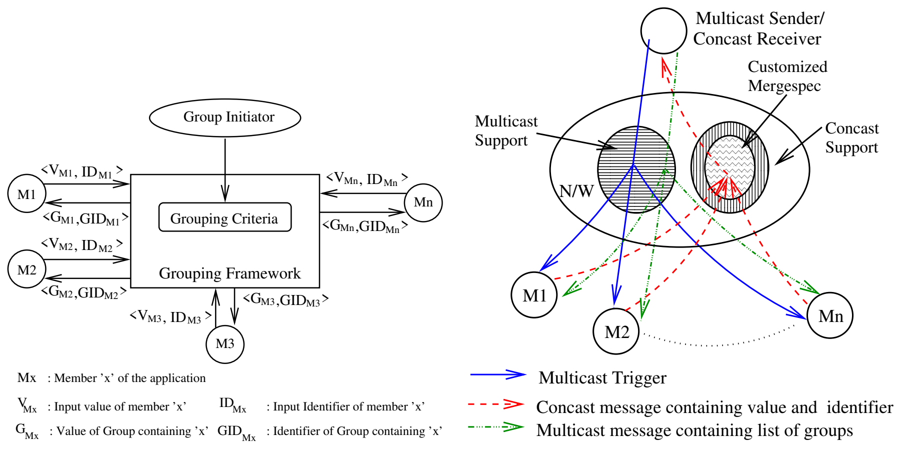
The framework, using the grouping criteria, compares and processes input values of all participants and assigns them to groups. The values of members that should be in the same group are mapped to a single group value v_g that characterizes the members of the group. The framework nondeterministically chooses one of the group members' identifier as the group identifier id_g. Thus if m', m'' and m''' constitute the first group, then $\{v_{m'}, v_{m''}, v_{m'''}\} \rightarrow v_{g1}$ and $id_{g1} \in \{id_{m'}, id_{m''}, id_{m'''}\}$. Once completed, the grouping result is propagated back to all participants so that each can learn its own group value and the identifier associated with it. Thus from a participant's perspective, the grouping service takes an application-specific value and an identifier as input and in return provides the group value and the group identifier of the participant.

The grouping service provides one other configurable aspect: applications can choose different levels of reliability for the service. The strongest form of reliability requires the service to process the values of *every* participant before returning the grouping results. Another form collects input values of members over a specified interval of time and then propagates the results back, independent of the number of participants that have been grouped. In the following subsections we describe the implementation of our grouping service.

3.1 Grouping Algorithm

We use concast and single-source multicast to implement a scalable solution to the grouping problem. Figure 3(b) depicts the messages sent during the operation of the service. These messages are discussed in more detail in the following overview of the grouping algorithm:

1. One of the participants initiates the grouping service and provides the application-specific grouping criteria in the form of a merge specification to be used by concast. The initiator node functions as a concast receiver and a multicast sender in the grouping algorithm. As described in Section 2, the concast service deploys the mergespec (i.e. grouping criteria) in the network along the path from the members towards the initiator (concast receiver).



(a) Abstraction of the Grouping Service (b) Messages used during Grouping

Fig. 3. The Grouping Service

2. The initiator multicasts a message that triggers all participants to respond.
3. Each member m responds to the trigger with a concast message containing its application-specific value v_m and identifier id_m.
4. Concast applies the grouping criteria to the concast messages. Each resulting group contains the merged value v_g and a identifier id_g. A single message containing a (value, identifier) pair for each of the resulting groups is delivered to the initiator (concast receiver).
5. The initiator multicasts the list of groups back to all the participants.
6. Once a participant receives the entire list it identifies its group. We discuss the process of identification below.
7. The value and the identifier associated with the participant's group are ultimately returned to the application on each node.

Once a participant receives the list of $\langle v_g, id_g \rangle$ pairs, it needs to determine which group it belongs to. An easy solution is to have the grouping framework attach the unique node identifier[1] nid of the member to the $\langle v, id \rangle$ pair input by it. When two values are classified under the same group, their corresponding nid's can be concatenated to the $nidlist$ of the group. Thus the concast receiver would get a message containing a list of $\langle v_g, id_g \rangle$ pairs and a $nidlist_g$ associated with each pair. This message is multicast back to the participants who identify their $\langle v_g, id_g \rangle$ pair by searching for their nid in the $nidlist_g$. However, this approach does not scale well, as the size of the $nidlist$ grows with the number of participants.

To achieve scalability, we propose a solution using Bloom filters. A Bloom filter [5] is a method for scalably representing a set of elements and then supporting membership queries for the elements. To summarize, a Bloom filter is a bitmap of l bits and k independent hash functions. For each output of the hash function (applied to some value e)

[1] This is not the same as the application-level identifier id.

the corresponding bit in the Bloom Filter bitmap is set. Thus bits $h_1(e), \ldots, h_k(e)$ are set in the bitmap. An element x is assumed to be present if all the bits $h_1(x), \ldots, h_k(x)$ are set in the Bloom Filter. It is possible that a lookup finds an element, even though it was not inserted. Such a misidentification is termed a *false positive*.

At node m, the grouping framework hashes the unique node identifier nid_m into an empty Bloom filter bf_m. The grouping framework couples bf_m with the $\langle v_m, id_m \rangle$ pair and forwards them as part of the concast data. At the concast merging nodes, when two v_m's belonging to the same group are merged, the corresponding bf_m are combined by bitwise OR-ing them together. Thus, the amount of grouping information returned to participants depends on the number of groups and not on the number of participants.

The concast receiver finally receives a message containing a list of $\langle v_g, id_g \rangle$ pairs and their corresponding bf_g's. This message is multicast back to the members, each of which performs a membership query for its own nid on each group's Bloom filter. This query must succeed for at least one group. If it succeeds on exactly one Bloom filter the member has uniquely identified the group to which it belongs. However, membership queries may succeed on multiple Bloom filters due to false positives. To overcome this problem, additional iterations of the grouping algorithm are carried out, with one difference from the steps above: members that have already uniquely identified their groups do not hash their nid in the Bloom filter in any of the subsequent iterations of the algorithm. This decreases the probability of false positives and helps the grouping algorithm converge faster. The grouping algorithm iterates until all members have uniquely identified their group, i.e. the concast receiver receives a message containing only empty Bloom filters.

3.2 Implementing the Grouping Algorithm

```
merge (MergeStateBlock msb, Packet p,
            FlowStateBlock fsb) {
  for each record r in p
      process_value (r.val);

  if (msb is NULL)
      msb ← create new merge state block
      msb.seq ← p.seq
      msb.neighbor_total ← fsb.unl_total
      msb.neighbor_count ← 0
      msb.timer ← NOW + δ

  if (p.senderid ∈
            msb.neighbors_heard_from)
      return (msb)

  for each record r in p
      for each record s in msb
          if (can_combine (r.val, s.val))
              s.val ← combine (r.val, s.val)
              s.senderid ← choose (r.senderid,
                                   s.senderid)
              s.bf ← r.bf || s.bf
              delete r from p
              break

  for each record r in p
      add r to msb

  add (p.senderid to
            msb.neighbors_heard_from)
  msb.neighbor_count ←
            msb.neighbor_count + 1
  return (msb)
}

done_allnode (MergeStateBlock msb) {
  return (msb.neighbor_count ==
            msb.neighbor_total)
}

done_timeout (MergeStateBlock msb) {
  return ((msb.neighbor_count ==
            msb.neighbor_total) ||
        (NOW ≥ msb.timer))
}
```

Fig. 4. Grouping Mergespec

Figure 4 describes the merge specification for implementing the grouping service. The initiator supplies three routines to the grouping service; ***process_value*** to preprocess the values of new incoming packets, ***can_combine*** to determine if two values are mergable and ***combine*** to merge two values. These three functions are used by the merge function (lines 1-25) at every concast node to achieve grouping. At the member node, the concast packet sent by the grouping framework carries the value v, the identifier id and the Bloom filter bf as data.

When a packet arrives at an intermediate concast node the merge function preprocesses it using ***process_value***. If the merge state for the DEC corresponding to the incoming packet does not exist then it is created and all variables accordingly initialized. Next, if the incoming packet is a duplicate packet then the mergespec skips its processing. Otherwise ***can_combine*** determines if the records in the packet are mergable with the records in the merge state. If found to be mergable ***combine*** is used to merge the values in the records. **choose** non-deterministically picks either of the two id's corresponding to the two merged values. Their corresponding Bloom filter's are merged using a simple *OR* operation. A merged record is deleted from the packet and in the end all unmerged records left in the packet are added to the merge state. The sender if of the processed packet is noted to avoid duplicate processing.

The mergespec also describes two kinds of **done** functions: **done_allnode** (lines 26-28) and **done_timeout** (lines 29-32). The two functions impose different conditions on packet forwarding at the concast node and as a result provide two different levels of grouping service. Either of the functions can be selected for use by the application at initiation time. Note that the application only needs to specify which of the two functions should be used and not provide the functions themselves.

One of the desired conditions of grouping is *complete reliability*, i.e. data from all participants must be processed to produce the final "merged" packet before delivering it to the concast receiver. This is accomplished by the **done_allnode** function which waits until it has received and processed a concast packet from all its upstream neighbors. But concast uses UDP for communication and packet losses during communication are possible. If the nodes wait indefinitely for the packets to arrive it may lead to deadlock. Thus if the concast receiver does not receive the merged concast packet, it times out and retransmits the trigger which causes the members to retransmit their $\langle v, id \rangle$ pair. Since the trigger is intended to complete the aborted process, it uses the same sequence number as the previous trigger. This redundancy ensures reliability.

The other desired functionality is that of *time bounded delivery*. This is accomplished by the **done_timeout** function which waits for a fixed duration of time after the arrival of the first packet and then forwards the merged packet towards the receiver. Thus members that fail to send their $\langle v, id \rangle$ pair within the fixed duration of time are excluded from the grouping process. This assures a time bounded response from the members once the trigger has been sent. In this case packet losses amount to excluding the member from the grouping algorithm.

The grouping process can be carried out periodically to incorporate new incoming members. However this can work well if the application has a low rate of participants leaving and joining and they are insensitive to the join delay. We are currently investigating techniques to make the grouping service adapt to more dynamic applications.

4 Applications

In this section we illustrate the use of our grouping service for two applications: reliable multicast and overlay networks. These applications use significantly different grouping criteria, demonstrating the flexibility of our grouping service.

4.1 Reliable Multicast

A common technique used in reliable multicast protocols—both router-based and end-system-based—is to assign receivers that need the same retransmission to the same channels (i.e. multicast groups) and thereby reduce redundant retransmissions.

A common technique is to use *lossprints* to record the retransmissions a machine requests [6–8]. Receivers with similar lossprints are likely to be "behind" the same set of lossy links with respect to the sender, and therefore can expect to request the same retransmissions in the future. Each multicast receiver keeps track of losses on the main multicast channel and generates a lossprint, which can be compared to the lossprints of other receivers to determine which receivers to combine into a retransmission group.

In end-system-based approaches, receiver lossprints must either be multicast to the entire group (thus creating additional load on the network in the form of bandwidth and routing state), or sent unicast to a central collection point (thus creating the risk of implosion). In our concast-based approach, receivers send their lossprints (in response to a trigger) toward the concast receiver. The lossprints are grouped, in the manner described below, hop by hop as they travel toward the concast receiver thereby reducing implosion and bandwidth usage. The concast receiver receives a single message containing a list of lossprints, each representing a group of receivers.

If losses occur *only* on bottleneck links lossprints accurately partition members into retransmission groups. However in practice, losses will occasionally occur on other links and the grouping algorithm must allow for a certain amount of "noise" in lossprints. Hence we combine two lossprints into the same group if they are within a threshold Hamming distance of their intersection (i.e. the bits that are set in both bitmaps). Each group is represented by the intersection of the lossprints of all of its members; this has the beneficial side effect of removing noise from the group lossprint as it moves up the tree.

Figure 5 illustrates the application supplied functions ***process_value***, ***can_combine***, and ***combine*** for this application. The application-specific data consists of a lossprint bitmap representing a fixed number of packets. The ***process_value*** function does nothing, as no pre-processing of the lossprints is required. The ***can_combine*** function compares two lossprints by first computing their intersection—a new lossprint that contains only the bits common to both lossprints. It then computes the Hamming distance (HD) between each lossprint and the intersection lossprint. The lossprints are considered combinable if both Hamming distances are less than ε. Finally, the ***combine*** function returns the intersection lossprint as the new group (lossprint).

4.2 Overlay Networks

Overlay networks offer advantages to applications that need special routing and addressing schemes. In particular, they allow unique routing and addressing schemes to

```
structure RM {
    bitmap lossprint
}

boolean can_combine (structure RM: s1,s2) {
    bitmap new_lossprint

    new_lossprint = s1.lossprint & s2.lossprint

    return ((no_of_bits (new_lossprint ⊕ s1.lossprint) < ε )
    && (no_of_bits (new_lossprint ⊕ s2.lossprint) < ε))
}

void process_value (structure RM: s1) {
    return
}

structure RM combine ( structure RM: s1, s2) {
    structure RM s3
    s3.lossprint ← s1.lossprint & s2.lossprint
    return (s3)
}
```

Fig. 5. Reliable Multicast Grouping Mergespec Functions

```
structure Ovly {
    integer hops
}

void process_value (structure Ovly s1) {
    s1.hops ← s1.hops + 1
}

boolean can_combine (structure Ovly: s1,s2) {
    return ((s1.hops < θ) && (s2.hops < θ))
}

structure Ovly combine (structure Ovly: s1,s2) {
    structure Ovly s3

    s3.hops ← max (s1.hops, s2.hops)
    return (s3)
}
```

Fig. 6. Overlay Mergespec functions

be implemented rapidly, without extra network support beyond standard services. Constructing an overlay requires some means of locating other participants and connecting with them to set up the overlay. In the recent past, peer-to-peer applications like Gnutella, CAN, and Chord [9–11] have proposed some interesting mechanisms for forming and using overlay networks. In this paper, we deal with the specific case of Gnutella.

Gnutella is a protocol for distributed search. Every participant is a client as well as a server. To join the Gnutella network, a new node connects to any existing member of the network and probes its neighbors within a specified overlay-hop radius to discover more members to connect to.

Because the overlay network is established at the application layer, the use of the underlying resources may not be optimum, i.e. multiple connections over the same network link. Since Gnutella forms connections by randomly connecting to a known node and then to other nodes that are within some distance of it, the resulting connections are independent of the underlying topology. To minimize link stress it is desirable to group members by topological vicinity, so that a node's neighbors in the overlay topology also tend to be close to it in the underlying topology. This is especially important in a Gnutella overlay, which relies on expanding ring searches to locate content. If nodes cache content and access patterns are consistent across a topological group, access latency should benefit from the increased efficiency of the overlay connectivity.

Our aim is to construct a topologically aware overlay network. The idea is to form groups by combining nodes within a certain hop-distance (radius) of each other. We also want to identify one node as the group representative. When all members of the group connect to their representative node, they will also discover the identity of the other nodes in the group.

Figure 6 describes the application-specific information used for grouping along with the ***process_value***, ***can_combine***, and ***combine*** functions. The information consists of a single integer: the hop count. The IP address of the member is used as the identifier in this application. The hop count in the value structure carried by a concast packet reflects the number of *concast hops* traversed so far.

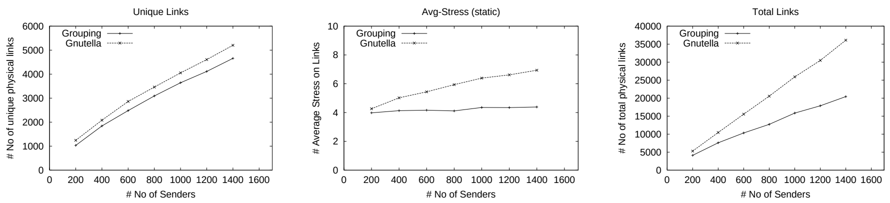
Every member sets the hop count to zero when originating its value, and sets the identifier to its own IP address. At an intermediate concast node, the merge function combines values in the packet with the existing merge state as follows: ***process_value*** increments the hop count. The ***can_combine*** determines two values to be mergeable iff the hop count of both the values is less than the threshold θ. If so, ***combine*** "merges" the two values by selecting the larger of the two hop counts. The grouping framework non-deterministically chooses either of the IP addresses as the identifier(representative node) for the new merged group. Consequently, all members that lie within a radius of θ concast hops of the intermediate node are merged together into the same group and the IP address of a representative node is associated with each group. This enforces the required distance relationship, in concast hops measured towards the concast receiver, among the members of the group. The resulting overlay network is closer to the underlying topology than a randomly connected network.

5 Simulations

5.1 Topology-based Grouping

To demonstrate the topological benefits of our grouping algorithm we simulated the overlay grouping mechanism proposed in Section 4.2 to form an overlay and compare its characteristics with overlays formed using the Gnutella approach (described later). The overlay graphs are generated on a 2376 node GT-ITM transit-stub topology. The member nodes and the concast receiver node are randomly selected from the stub domains. The number of members in the overlay varies from 200 to 1400 in steps of 200, with each member having three or four neighbors. Our grouping algorithm places members within a radius of *three* concast hops in the same group.

Member identify their group and the corresponding representative node. They make connections to the representative nodes and also discover other nodes in the same group to connect to. A few connections are made to representative members of other groups to achieve global connectivity. In the Gnutella approach connections are made randomly to any member of the overlay.



(a) Unique Links in Overlay

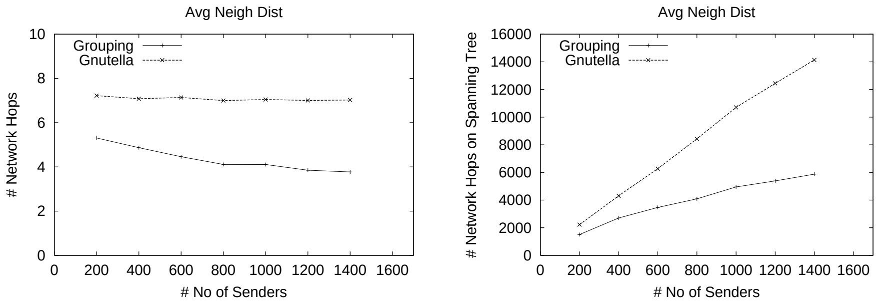
(b) Average Stress of Links in Overlay

(c) Total Links in Overlay

Fig. 7. Network Link Measurements

Connecitons formed using the grouping mechanism traverse fewer underlying network links as comapred to those using Gnutella, and this directly influences the number of links, the stress (defined as the number of connections that pass through the link) on those links, and the average hop distance to neighbors. These are important characteristics that affect latency and congestion issues. Figure 7 plots the network link characteristics of the two overlays and Figure 8 plots the connectivity characteristics of the two overlays.

Figure 7(a) illustrates that the number of unique links in the grouped overlay is marginally less than the unique links in Gnutella. However Figure 7(b) shows that the average stress on the links remains constant for grouped overlays across the number of receivers while the stress on the links in Gnutella increases with the increasing number of receivers. Figure 7(c) synthesizes the information in Figure 7(a) and Figure 7(b) by measuring the total number of links in the overlay and demonstrates the efficient use of links by the grouped overlays as compared to the Gnutella network.



(a) Avg. Neighbor Distance in n/w hops

(b) No of Links in MST

Fig. 8. Overlay Characteristics

Figure 8(a) plots the average number of underlying network hops, taken over all pairs of neighbors, for both overlays. Since Gnutella connections are formed randomly, increase in membership does not affect the average network hop distance to the neighbors. In the grouped case, increase in membership casues the number of members within a group to increase. Thus more neighbors are now available to form connections and this reduces the average hop distance for the grouped overlay.

The average network hop distance to neighbors has a direct influence on the average delay experienced by the nodes when communicating with their neighbors. Figure 8(b) plots the average network hops in the connections that are part of the minimum spanning tree built using the two overlays. Again, we see that the grouped overlay has a more compact network connectivity.

5.2 Notification

Although the grouping algorithm is probabilistic in nature, it exhibits very nice convergence properties. As members learn their respective groups, they refrain from setting bits in the Bloom Filter which greatly reduces the false positive rate (i.e., the number of bits selected by more than one receiver). Even when the number of group members is significantly larger than the size of the bitmap in Bloom Filter (i.e., the false positive rate is extremely high – multiple members per bit), the algorithm converges in only a few iterations. The reason for this is that colliding members remain confused *only if they belong to different groups*. As long as the number of groups remains small compared to the number of members—which is arguably necessary for scalability of any grouping scheme—it is very likely that at least *some* colliding members will end up in the same group and become aware of their group despite false positives. This means that a fixed-size bitmap in the form of a Bloom Filter can be used with a very large member set.

To demonstrate the convergence properties of the grouping algorithm we simulated it on a 1000 node transit-stub network topology generated using the GT-ITM package [12] for the multicast grouping problem. The multicast members were randomly selected within the stub domains in the graph. We simulated our group formation policy described in Section 3 and Section 4.1 to group multicast receivers behind the same "congested" link into the same group. We randomly define L links in the graph as "congested", resulting in at most $L + 1$ possible groups.

Figure 9a plots the number of confused receivers against the number of iterations needed to converge for 300 group members, 50 lossy links (i.e. 51 groups) and 3 different Bloom Filter sizes of 128, 256 and 512. The graph highlights the robustness of the grouping algorithm. Even with more than two receivers per bit, the algorithm has a steep convergence rate (i.e. converges quickly).

Figure 9b plots the average number of iterations needed to converge against the number of application members (receivers). In this case we varied the number of receivers from 100 to 800 in increments of 100 for Bloom Filter sizes ranging from 128 to 1024. The number of lossy links (groups) was always 10 percent of the number of receivers. This graph highlights the relative insensitivity of the algorithm to the number of receivers, for a constant receivers/groups ratio.

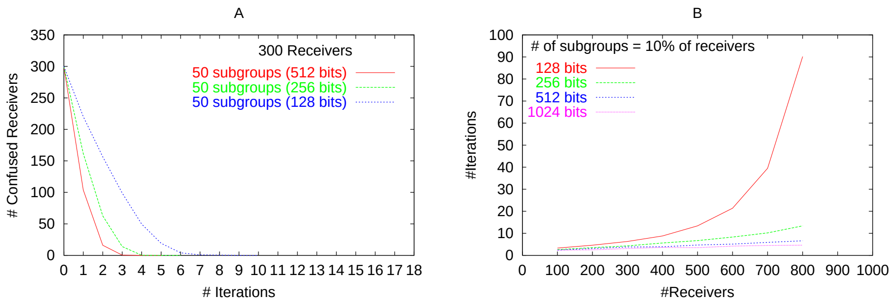


Fig. 9. Convergence rate for (a) different bitmask sizes, and (b) for differing numbers of receivers.

6 Conclusion

We have presented a scaleable and flexible grouping service based on concast and best-effort single-source multicast. The service does not require any router extension other than concast. Applications can make use of the service by supplying three simple functions to our generic group merge specification. These functions manipulate application-supplied data to implement application policy for partitioning members into groups. The functions are easy to specify, and can pose no security threat when used with the concast merging framework inside the network.

Concast makes it possible to collect the grouping information at a single point. Our announcement technique uses Bloom filters to enable each participant to learn its group assignment without explicitly identifying individuals in any message. Our simulation results show good convergence properties for our grouping service, even when the ratio of group size to Bloom filter size is large. Also, by customizing our grouping service to two diverse applications: reliable multicast and overlay networks, we show that the service is flexible and can be adapted to various applications. In the future, we plan to investigate how to apply the grouping framework to other classes of applications.

References

1. 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.
2. 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.
3. Billy C. Mullins Amit Sehgal Kenneth L. Calvert, James Griffioen and Su Wen. Concast : Design and implementaion of an active network service. *IEEE Journal on Selected Areas in Communications (2001)*, pages 19(3):426–437, March 2001.
4. D. Katz. IP Router Alert Option, February 1997. RFC 2113.
5. Burton Bloom. Space time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, pages 13(7):422–426, July 1970.
6. Sylvia Ratnasamy and Steven McCanne. Inference of Multicast Routing Trees and Bottleneck Bandwidths using End-to-end Measurements. In *the Proceedings of the 1999 INFOCOM Conference*, March 1999.
7. Sylvia Ratnasamy and Steven McCanne. Scaling End-to-end Multicast Transports with a Topologically-sensitive Group Formation Protocol. In *the Proceedings of the International Conference on Network Protocols (ICNP '99)*, November 1999.
8. I. Kouvelas, V. Hardman, and J. Crowcroft. Network Adaptive Continuous-Media Applications Through Self Organised Transcoding. In *the Proceedings of the Network and Operating Systems Support for Digital Audio and Video Conference (NOSSDAV 98)*, July 1998.
9. Clip2.com. The gnutella protocol specification ver 0.4, 2000. http://www9.limewire.com/developer/ gnutella_protocol_0.4.pdf.
10. M. Handley R. Karp S. Shenker S. Ratnasamy, P. Francis. A scalable content addressable network. In *in Proceedings of ACM Sigcomm '01 Conference*, August 2001.
11. D. Karger F. Kasshoek I Stoica, R. Morris and H. Balakrishnan. Chord : A scalable peer-to-peer lookup sevice for internet applications. In *in Proceedings of ACM Sigcomm '01 Conference*, August 2001.
12. Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, June 1997.