GSGC: An Efficient Gossip-Style Garbage Collection Scheme for Scalable Reliable Multicast*

Katherine Guo, Mark Hayden, Robbert van Renesse, Werner Vogels and Kenneth P. Birman

Department of Computer Science

Cornell University

Ithaca NY, 14853

{kguo,hayden,rvr,vogels,ken}@cs.cornell.edu

Abstract

To deliver multicast messages reliably in a group, each member maintains copies of all messages it sends and receives in a buffer for potential local retransmission. The storage of these messages is costly and buffers may grow out of bound. Garbage collection is needed to address this issue. Garbage collection occurs once a process learns that a message in its buffer has been received by every process in the group. The message is declared stable and is released from the process's buffer.

This paper proposes a gossip-style garbage collection scheme called GSGC for scalable reliable multicast protocols. This scheme achieves fault-tolerance and scalability without relying on the underlying multicast protocols. It collects and disseminates information in the multicast group by making each group member periodically gossip information to a random subset of the group.

Extending the global gossip protocol further, this paper also investigates a local gossip scheme that achieves improved scalability and significantly better performance. Simulations conducted in a WAN environment are used to evaluate the performance of both schemes.

Keywords: Message stability, fault-tolerance, scalability, reliable multicast, simulation.

*This work was supported by the ARPA/ONR grant N00014-96-1-1014 and the GTE graduate student grant to Horus research group.

1 Introduction

Multicast communication is an efficient method for disseminating data in a multicast group with a sender and a set of receivers. Many multicast applications require reliable delivery of data to all the receivers. For example, reliable multicast is used in Distributed Interactive Simulations (DIS) for dynamic terrain updates [10]. It is used for dissemination of stock quotes to a large number of clients, and by web servers to send updates of web pages to their proxies.

In scalable reliable multicast protocols [7, 15], it is efficient to use the local repair scheme, that is, for each group member to retransmit messages in response to requests by other members that have detected message losses. For applications that require all messages to be delivered to all correct processes in the group, it is also necessary to buffer all the received messages at every member to handle the case of sender crash and network partition.

On the other hand, the storage of these messages is costly and the buffer space at each member is limited, preventing the protocols to scale to a large group size. A form of garbage collection is needed to address this issue.

In order for members that join the group late to catch up with the rest of the group, a small number of members are designated as the Late-Join Handlers (LJHs). LJHs keep all messages they sent and received in their buffers. The decision of how long the LJHs should keep multicast messages is made by applications instead of the garbage collection mechanism.

Whenever a data message has been received by all the members, only the LJHs should store the message. Other members should discard it, since none of the members in the current multicast group needs retransmission. A message is called *stable* if it is received by all the members of the group. To do this garbage collection, a mechanism is needed to detect which messages are stable. Also a failure detection mechanism is needed to report the current group membership, otherwise a failed member could prevent garbage collection altogether.

We propose an efficient method for garbage collection at the transport level by using *session messages* [6, 7, 10]. It is called the Gossip-Style Garbage Collection (GSGC) service. At minimum cost, the GSGC service offers failure detection and buffer management to existing large scale reliable multicast protocols.

The message stability detection mechanism can also support atomic message ordering. For example, this research was triggered by a problem that a Swiss bank faced when using the Isis group communication system [2]. In their set-up, they had two server machines and about a hundred PC workstations organized in a group. The servers broadcast updates to replicated data maintained at each of the workstations. The updates had to be delivered atomically, in spite of server failures. Therefore, the workstations had to buffer the data until it was known that the data was delivered everywhere. The rate of the updates were sometimes so high, that Isis' stability protocol was not able to keep up, and buffers grew too large. The effect was much exacerbated by the fact that multiple groups were used. Correct ordering between the groups required that switching from sending in one group to another was done only after the messages sent and delivered in the first group had become stable.

These machines were inside a single branch and on a single local area network. One can easily envision multiple branches being linked together, with many hundreds if not thousands of machines. Our interest is in finding a scalable stability detection protocol that fits future requirements.

The paper proceeds as follows: Section 2 describes the system model in which the GSGC scheme is developed and evaluated. Section 3 presents the two integral parts of the GSGC scheme – stability detection and failure detection. Section 4 examines the behavior of the stability detection protocol in various scenarios using simulation. Section 5 discusses extensions to the basic global gossip framework using the concept of local gossip. Finally, Section 6 presents conclusions and directions of future research.

2 System model

GSGC is the garbage collection and failure detection framework intended for reliable multicast in a large scale environment where messages may get dropped and processes may crash. In case of network partition, garbage collection is conducted in separate partitions. When the partitions merge into a whole group again, mechanisms in various reliable multicast protocols will allow members to catch up, for example, using state transfer to conduct repairs at the application level [2, 5].

GSGC is based on common assumptions about reliable multicast protocols. One assumption is that a multicast group of size n consists of a set of processes named from 1 to n. Each member of the group can be a sender multicasting data messages to the entire group. Without loss of generality, we assume m ($m < n$) processes are senders and they are numbered 1 through m. Each member is always a receiver. The sender assigns each data message a sequence number that is unique for the particular sender. The second assumption central to GSGC is that the data messages have unique names; this name consists of the global unique sender name and a local unique sequence number. Even though retransmissions for data messages are handled by the various reliable multicast protocols, the GSGC service has its own built-in reliability mechanism. FIFO ordering is not assumed.

3 Protocol description

To do effective garbage collection, one must detect when a message is stable and obtain a consistent view of the current group membership. Therefore, the stability detection protocol and failure detection protocol are two integral parts of garbage collection. They are described in the following sections.

3.1 Stability detection algorithm

In the assumptions made in Section 2, there are m senders in a group of size n and each sender uses an independent sequence space. Each member maintains an m-element sequence number array R where its j-th element $R[j]$ is the maximum sequence number such that all messages with less sequence numbers from sender j have arrived at this member. Each member also maintains an n-element "Live" array L reflecting current group membership (it will be described in section 3.2) and an n-bit "Whom-I've-heard-from" bitmap array W for recording from which members it has received the sequence number arrays. A message is stable if it is received by all the members in the current group.

The simplest way to detect stability is for each member to send its sequence number array R to one designated member, the *coordinator*. After receiving the sequence number arrays from all the members, the coordinator calculates their element-wise minimum, and a stability array S is created where $S[j]$ is the minimum of the j-th element of every member's sequence number array. The coordinator then multicasts the stability array S in the group. After receiving S, each member can release data messages from sender j with sequence numbers less than $S[j]$.

When the group size is large, an implosion problem will occur at the coordinator, which makes the naïve method not scalable. Adding a multi-level hierarchy reduces the implosion problem but introduces new problems. One such problem appears when some interior nodes in the hierarchy crash. The chance for member crashes increases as the group size increases. In a large multicast group, membership change is frequent, requiring the hierarchy to be rebuilt frequently. This fact makes the pure hierarchical approach not scalable [8]. On the other hand, for reliable multicast protocols like LBRM [10] and RMTP [15] with built-in hierarchical structures, these structures can also be used for garbage collection. Since the management of group membership hierarchies is already provided by the multicast protocols, the hierarchical extensions of the naïve method can be used to improve scalability [8].

The goal is to make the stability detection protocol *robust* and *scalable*. A robust protocol tolerates message losses and process crashes. A scalable protocol handles a large multicast group with frequent membership changes. Three design features make the stability detection protocol in GSGC scalable.

- The implosion problem is eliminated completely.
- Traffic load generated by the protocol is minimized.
- The state information passed around by each member does not grow proportionally with the group size.

To achieve robustness and scalability, the *gossip* technique is used. The protocol is divided into equally timed steps. During each step, every member constructs a gossip sub-group consisting of b distinct members with ranks randomly chosen from 1 to n. In the first step, every member sends its sequence number array R to its gossip sub-group. After receiving a gossip message, a member computes the "Min-so-far" array M which is the element-wise minimum of sequence number arrays of itself and of other members that it has heard from. It also computes the "Whom-I've-heard-from" array as the element-wise maximum of "Whom-I've-heard-from" arrays of itself and of other members that it has heard from. In the subsequent steps, every member gossips its "Min-so-far" array M and its "Whom-I've-heard-from" array W to a different random sub-group. Instead of sending their information to one coordinator, each member uses gossip messages to disseminate their information in the group step by step. After certain number of steps, one member receives information about all current members, and the "Min-so-far" array M at this member becomes the stability array S. This is detected when the "Whom-I've-heard from" bitmap array W contains 1's for all current group members. A statistical model to represent this process is under investigation.

At this point, this member starts disseminating S in the group by putting it on the future gossip messages. Upon receiving S, a member discards stable messages accordingly. To save on bandwidth requirement of future gossip messages and to disseminate S faster, one could

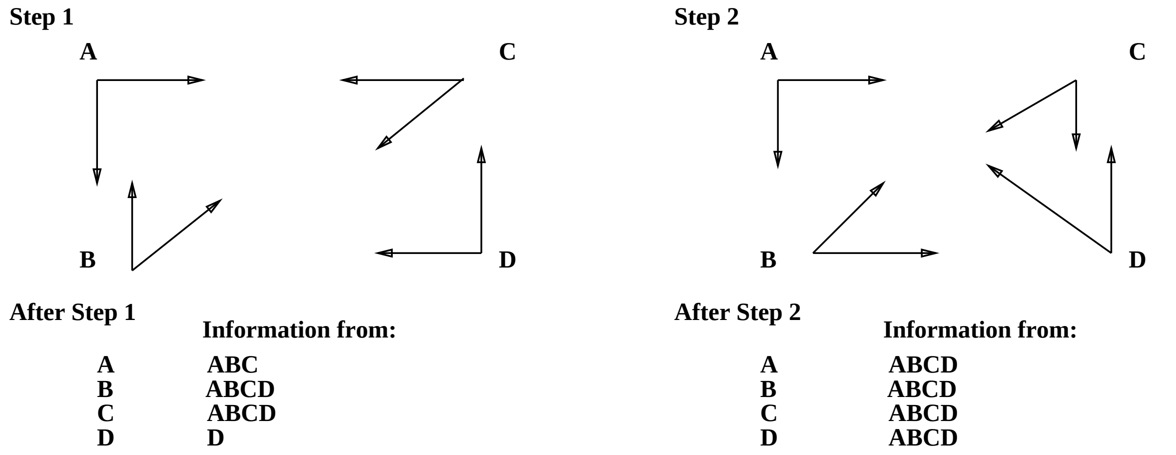


Figure 1: An example run of the stability detection protocol.

multicast S in the entire group. Instead of implementing reliable multicast again, an existing reliable multicast protocol can be used. However, this method has a drawback. Some reliable multicast protocols do not guarantee a multicast message to be received by all members in the group. If these protocols are used to distribute S, there is no guarantee that S will arrive at every member. A hybrid scheme can fix this problem. S is multicast to the entire group, and periodically S is piggybacked on gossip messages in future rounds to reach members which are left out on the original multicast of S.

An example is given in Figure 1 to illustrate the protocol. The group size is 4, and the sub-group size for gossip is 2. One round of the protocol is finished after 2 steps. At the end of the first step, member B and C construct the stability array, whereas A and D only have partial information. During the second step, A receives D's sequence number array from D directly, and then constructs the stability array. Meanwhile, D obtains the stability array directly from B and/or C. At the end of the second step, all 4 members have the stability array, therefore they can discard messages accordingly. In the optimized scheme, B or C multicasts the stability array in the group at the end of the first step, and only one step is needed to detect stability.

The end of a round of the protocol is reached at each member when the member receives S. Each member keeps a round number to distinguish gossips from different rounds. The starting points of gossip for group members are scattered randomly during the interval of one step rather than concentrated at the beginning of each time step. This effectively reduces message bursts. The pseudo-code is presented in Figure 2.

Notation:

ArrayMin is element-wise minimum of the input arrays.
ArrayMax is element-wise maximum of the input arrays.
For two arrays A and B, $A < B$ means $A[i] < B[i]$ for all i.
There are n members in the group numbered from 1 to n.

Each member i keeps four arrays and one number.
R_i: Sequence number array.
M_i: Minimum-so-far array.
W_i: Whom-I've-heard-from array.
S_i: Stability array at the end of the previous round.
r_i: Round number for the current round.

Initially, every member i has $M_i = R_i$, $S_i = [0 \ldots 0]$, $W_i[i] = 1, W_i[j] = 0$ for $j \neq i$ and $r_i = 0$.

Periodically each member sends out a gossip message containing three arrays and one number.
(M, W, S, r) where $M = M_i$, $W = W_i$, $S = S_i$, and $r = r_i$.

Every member reacts to received messages as follows:
Upon receipt of a data message, member i updates R_i.

```
Upon receipt of a gossip message (M, W, S, r), member i takes the following actions:
if (r == r_i)
      /* receive a message in the current round */
      if (W_i > W) /* this message is redundant */
            do nothing;
      else if (W_i < W) /* the received message is more up-to-date */
            M_i = M;
            W_i = W;
      else /* normal process */
            M_i = ArrayMin(M_i, M);
            W_i = ArrayMax(W_i, W);
      end if
      S_i = ArrayMax(S_i, S); /* each round can have many (up to n) concurrent S_i's,
      the maximum is the most up-to-date one. */
      if (W_i contains all 1's) /* start next round */
            S_i = M_i;
            r_i = r_i +1;
            M_i = R_i;
            W_i[i] = 1; W_i[j] = 0 for j ≠ i;
      end if
else if (r == r_i + 1)
      /* receive a message from the next round */
      M_i = M;
      W_i = W;
      S_i = ArrayMax(S, S_i);
      r_i = r;
      if (W_i contains all 1's) /* start another round */
            S_i = ArrayMax(S_i, M_i);
            r_i = r_i + 1 ;
            M_i = R_i;
            W_i[i] = 1; W_i[j] = 0 for j ≠ i;
      end if
else if (r > r_i + 1) /* should never receive a message like this */
      error;
else if (r < r_i)
      /* receive a message from previous rounds, ignore, since it is out of date */
      do nothing;
end if
```

Figure 2: Stability Detection Protocol (In this version, the stability array S is piggybacked on all gossip messages.)

3.2 Failure detection algorithm

Failure detection in GSGC uses a similar style of gossip. Initially, there are n members in the group numbered 1 through n. As time passes by, some member might crash or might leave the group voluntarily. Many failure detection algorithms have the same underlying principle. Under the assumption that every member is constantly sending out messages, if a member has not been heard from after a certain time, it is assumed to have crashed or left the group. But having each member periodically multicast "I'm alive" session messages in the group is not optimal because it adds unnecessary load to the system.

The gossip style failure detection algorithm works as follows. Every member maintains an n-element "Live" array L which is filled with 0's initially. This protocol is divided into equally timed steps. During every gossip step, each member i increments all the other elements in its "Live" array L by 1 while keeping $L[i] = 0$, then it gossips L to a random subset. Upon receiving any type of message from member j, a member sets $L[j] = 0$. Upon receiving another "Live" array L', a member replaces its own "Live" array L with the element-wise minimum of its old L and L'. Small values in the live array indicate that the corresponding members are active, and large values signify that the corresponding members have not been heard from recently. The pseudo-code is presented in Figure 3.

```
Each member i keeps a live array L_i.

Initially L_i = [0 0 ... 0] at every member i.

Periodically, member i does the following:
    L_i[j] = L_i[j] + 1; for all j ≠ i
    sends out a gossip message containing L = L_i;

Every member reacts to received messages as follows:
    Upon receiving a data message from j, member i does the following:
        L_i[j] = 0;
    Upon receiving L, member i does the following:
        L_i = ArrayMin(L_i, L);
```

Figure 3: Failure Detection Protocol

It takes time for the "Live" arrays from each member to propagate throughout the entire group. This time is called the *diffusion time* D. A threshold value K, the maximum "Live" array value is set. Once K is reached, the corresponding member is declared faulty. The value K depends on the diffusion time D. Assuming each element of the "Live" array occupies 1 byte, the size of a gossip message becomes n bytes where n is the group size. A hierarchical structure can be employed in the failure detection protocol to reduce gossip message sizes and improve scalability. Since the stability detection protocol is the focus of this paper, the failure detection protocol is not discussed further and will be analyzed in a future paper.

3.3 Integration of the stability detection and failure detection algorithms

When there are membership changes, the failure detection protocol assists the stability detection protocol. If a faulty member is detected, this information is propagated throughout the group. Members always check the "Whom-I've heard from" array W against the current group membership before deciding if the stability array S is reached, thus preventing an indefinite wait for faulty members' sequence number arrays.

Recall from Section 1, a small set of members are designated as Late-Join Handlers (LJHs) and the LJHs will provide new members with data necessary to catch up with existing members. When a new receiver joins the group, after receiving necessary information from the LJHs, it also joins the stability detection protocol. The "Whom-I've-heard-from" bitmap array W adds one more bit at the end representing the new receiver. Any member which hears indirectly or directly from this new receiver notices the change in W from gossip messages and adds one bit to its own W.

The invocation of the stability detection protocol depends on patterns of message sending and membership changing. Since there is a limit on buffer space at multicast group members, a round of the stability protocol should start whenever the buffers reach some threshold. An analytical model for determination of this threshold is discussed in [8].

During a round of the stability detection protocol, a steady stream of new member joins creates new sequence number arrays needed for the calculation of the stability array. Message stability will not be reached unless new members stop joining the multicast group. In other words, when the time interval between two consecutive new member joins is smaller than the time it takes for the stability detection protocol to finish a round when the membership is stable, this round will continue until the membership array W stops expanding.

Two approaches can solve this problem. If one knows the pattern for new member joins, an execution control protocol can start rounds of the stability detection protocol strategically during periods when new member joins are scarce. If such periods do not exist, or no pattern for new joins exists, an admission control protocol can regulate new member joins. The admission control protocol either limits the total number of new joins during each round of the stability detection protocol, or specifies a window period for new members to join. This window is relatively small compared to a round of the stability protocol, and is located immediately after the start of a protocol round.

In the execution control approach, invocation of the stability detection protocol is changed according to member joins, whereas in the admission control approach, the stability detection protocol is running as usual, but member joins are restricted. Unlike member joins, member crashes do not delay the detection of message stability. With execution or admission control, membership changes cause little disturbance to the stability detection protocol. This is an important feature that makes the scheme scalable.

4 Simulations of the stability detection protocol

For a given underlying network topology, set of group members, set of senders, and patterns for message sending and message loss, it is possible to analyze the behavior of the GSGC algorithm with a fixed step interval, and a fixed sub-group size. However, interest lies in the performance of the GSGC algorithm across a wide range of network topologies and scenarios. For this, we conducted simulations using the ns [12] simulator. For a large group size, the probability that a particular node in a randomly labeled tree has a degree of at most four approaches 0.98 [13], therefore, the underlying network used in the simulation is a balanced bounded-degree tree where interior nodes all have degree four. The network topologies are based on some generic network simulation schemes used in [7].

To gain insight on scalability of the GSGC protocol, the simulations are conducted in a set of wide-area networks. Each network in the simulations consists of nodes and links. Each link is bi-directional and each direction has a bandwidth of 30K bps allocated for the session messages in GSGC to conduct message stability detection. Message propagation delay on each link is $w = 5$ milliseconds, which is typical for wide area links. A rate-controlled network is assumed in which the data source is shaping its traffic by delaying packet sends to meet the $30K$ bps allocated rate requirement. Under this assumption, the expected time a u-byte message spends on the wire to travel one link is $t_0 = w + u/v$, where v is the bandwidth. The router processing time for a message is 1 millisecond. The time needed for a host to send a message follows the formula $t_s(u) = 100 + 2(94 + 35u/4000 + 50u/1000) + 50 = 338 + 47u/400$ (microseconds) [1, 11]. The time needed for a host to receive a message is normally about 10% higher than the sending time since interrupts need to be handled [1]. It is set to $t_r(u) = 1.1 \times t_s(u)$. The queuing delays incurred at the hosts and routers are also simulated by ns. The message header size is set to $h = 32$ bytes which is enough for most transport protocols [1, 16].

Separate simulations are conducted for the stability and failure detection protocol. This paper focuses on stability detection, therefore, the stability detection protocol is tested in the situation where group membership remains unchanged.

4.1 Performance indices

The most important goal for a message stability detection protocol is to minimize the time to stabilize a message, reducing the buffer space required for data messages at each member to a minimum. The effectiveness of the protocol in achieving this goal is analyzed in terms of time and space.

To measure the time requirement, *Time Per Round* (TPR) is defined as the duration of time between the start of the stability detection protocol and the moment the first member constructs the stability array S. After the first member constructs S, it immediately multicasts the array in the group, therefore every member will receive the stability information within one multicast. A more important time index is *Time-To-Stable* (TTS), which is defined as the time between the moment a data message is multicast and the moment it is detected to be stable. TTS depends on three things: TPR, the frequency to trigger each round of the stability detection algorithm, and the underlying reliable multicast protocol. Analysis of TTS requires implementation or simulation of the reliable multicast protocols. If the reliable multicast protocol can deliver a message to all the receivers within D seconds, the stability detection algorithm is triggered every F seconds, and it can detect the message's stability within TPR seconds, then the maximum TTS becomes $D + F+$TPR seconds. This means that at most $D + F+$TPR seconds after a message is multicast from a sender, it can be deleted from the network. Since TPR is the factor that is determined by the stability detection protocol, this paper only studies TPR.

To measure the space requirement, the queue size at each node is recorded whenever the node sends or receives a message. The *maximum* and *average of the recorded queue sizes* over all nodes in the network are calculated. They indicate the load of processing message sends and receives, and also indicate congestion of the links.

To investigate the behavior of the protocol in detail, two more indices are measured. The first is the *number of steps needed for a round.* The second index is the *average number of messages sent out by each member during unit time.* This is an indicator of the load the GSGC protocol adds to the network. All the indices are used to investigate the behavior of the protocol under a number of scenarios.

In most applications where the multicast group is large, normally a small percent of the members are active sources for data messages. Without loss of generality, the number of senders is set to $m = 50$ where group sizes range from 50 to 500. Recall from Section 3.1, each gossip message used to detect stability contains a 32-byte header, an m-element sequence number array M where each sequence number occupies 4 bytes, an n-element bitmap array W where each element is one bit long and an integer round number which has 4 bytes. The overall gossip message size for a group of size n is $32 + 4 \times m + n/8 + 4 = 236 + n/8$ (with some padding to make it aligned in the packet). In the simulation, n ranges from 50 to 500; and the packet size ranges from 243 to 299 bytes. Since a typical WAN can handle packets shorter than 500 bytes without fragmentation [14], packet fragmentation is not considered in the simulations.

4.2 Simulations with a fixed group size

For a given group size and a given number of senders, there are two control parameters in the stability detection protocol – the step interval and the sub-group size for each gossip. The goal is to analyze the behavior of the GSGC protocol under different values of these two parameters. To see the effect of step interval on the protocol, the step interval is ranged from 1 second to 20 seconds with incremental steps of 1 second. To see the effect of the sub-group size on the protocol, the sub-group size is varied from 1 to 5. In a WAN, IP-multicast is not efficient in sending messages to small groups that are constantly changing [4], thus b unicasts are used to send gossip messages to each sub-group of size b.

For the fixed group size $n = 200$ with $m = 50$ senders, tests are conducted in two categories: the dense test and the sparse test. In the dense test, a balanced bounded-degree tree of size 200 is built, where every node in the tree is a member of the multicast group. In the sparse test, a balanced bounded-degree tree of size 1000 is built. 200 nodes are randomly chosen to be in the multicast group, and the remaining 800 are routers.

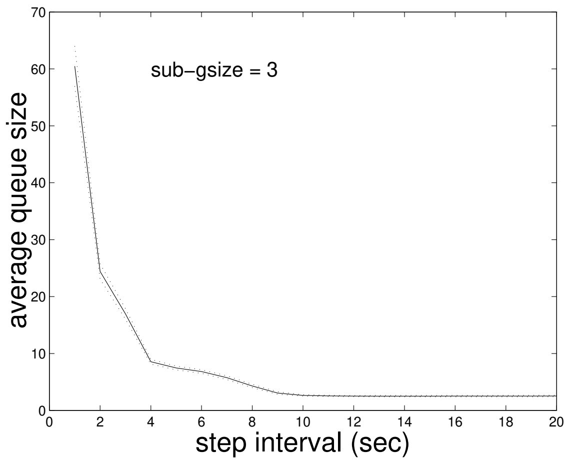
In both tests, the following two simulations are conducted:

I. Every node in the tree has infinite buffer space, thus the stability detection protocol has no message loss;

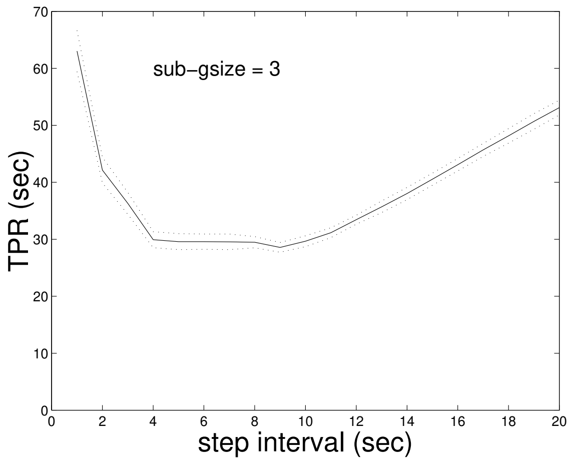
II. Every node in the tree only has enough buffer space to store 64 gossip messages for each connected link[1]. Gossip messages arriving at a node with a full buffer are dropped. Additionally, two random gossip messages in each step interval are dropped at the senders.

An intermediate simulation is also conducted where every node in the tree has infinite buffer space, and two random gossip messages in each step interval are dropped at the senders. The results are similar to the first simulation, therefore not presented.

A suite of 20-run statistical tests are conducted where each test has a given sub-group size and step interval. For dense groups, a random number generator is used for constructing sub-groups, and each run uses a different seed for the generator. For sparse groups, each run corresponds to a different randomly constructed 200-member group in the 1000-node tree.



(A): average queue size



(B): Time Per Round

Figure 4: Simulation I (no message loss) with a dense group of size 200, and sub-group size 3.

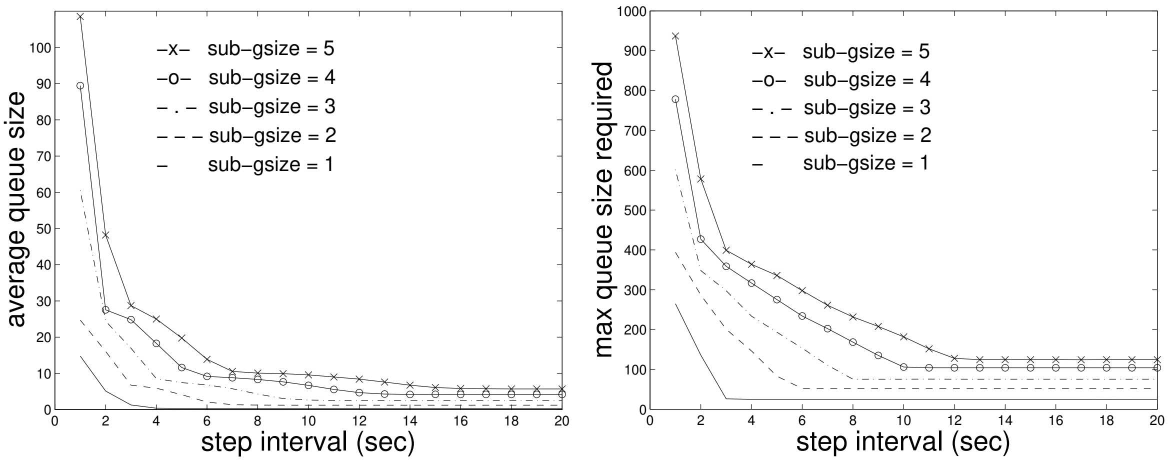
The standard deviation of the sampling results follow similar trends in all the simulations. Figure 4 presents the sample mean and sample standard deviation for two indices in simulation I with a dense group of size $n = 200$ and sub-group size 3. The two indices are the average queue size and TPR. The solid line in each figure is the mean and the two dotted lines plot one standard deviation above and below the mean. The largest ratio of standard deviation over mean among 20 tests, each with 20 samples is 7% for the average queue size and 5% for TPR. Under the general assumption that the indices follow normal distribution, approximately 67% of the sample points fall between the two dotted lines. Since the standard deviation is small, only the mean is reported in the rest of this section.

[1] In Unix, the default buffer space for each TCP connection is 32K bytes. Therefore, each buffer can store $32K/500 = 64$ 500-byte messages. To be conservative, the buffer size is limited to 64 gossip messages.

Out of the 12,000 individual runs of the simulations[2], 17 of them can not detect message stability within 10 minutes. This means that close to 0.15% of the sample points are bad. These bad samples are excluded from the statistical analysis in the rest of this section. The probability for a round not to finish after a relatively long time exists, but is very slim. In practice, a second round of the stability detection protocol can start with a different seed for generating sub-groups. The probability that both rounds take an unreasonably long time is even slimmer – 0.0225%.

4.2.1 Simulation I of dense groups

Since gossip messages are sent out by unicast, the number of messages sent out by each member during each step is the same as the sub-group size. For a given data point (x, y) in the simulation with sub-group size x and step interval y, the number of messages each member sends out per unit time is x/y. Therefore the traffic load generated by the stability detection protocol is proportional to x/y.



(A): average queue size (B): maximum queue size

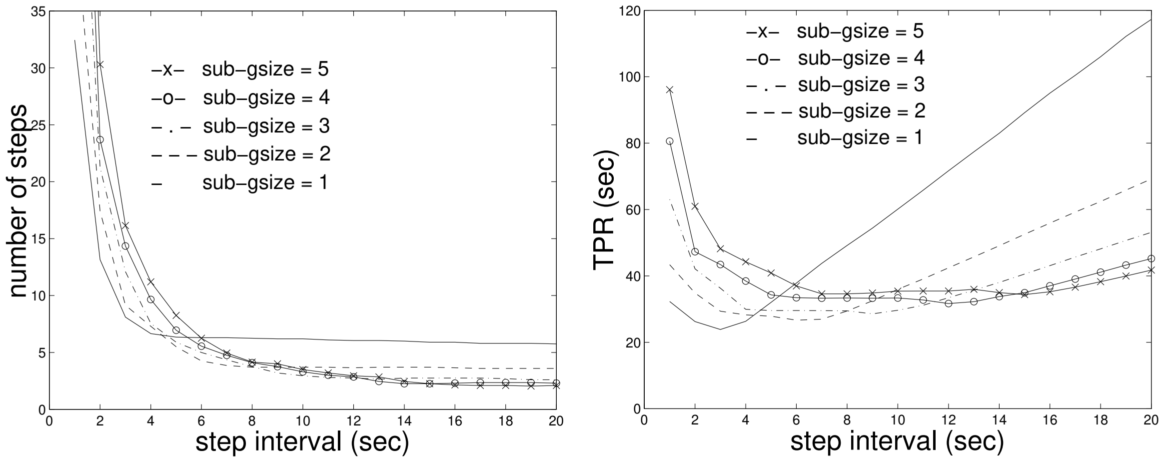
Figure 5: Simulation I (no message loss) with a dense group of size 200.

The average queue size reflects message burstiness. When large number of messages are sent out during a short amount of time, buffers at hosts and routers near message sources will receive a large number of messages to be processed and forwarded, resulting in large queue sizes. Figure 5(A) plots the average queue size recorded over all the nodes in the network. For any given curve, the decreasing part comes from the fact that as the step interval increases, the

[2]The dense and sparse tests each contain 3 simulations where each simulation covers 5 sub-group sizes and 20 step intervals. 20 sample runs are executed for a pair of sub-group size and step interval value. The total number of runs is $3 \times 2 \times 5 \times 20 \times 20 = 12,000$.

burstiness of messages decreases until it reaches a minimum. After a certain point, the increase of step interval will not reduce message burstiness any further. Sub-group size also influences message burstiness. For any fixed step interval, as the sub-group size increases, the burstiness from sending to one sub-group increases, which results in the increase of the average queue size. The maximum queue size shows the same trend as the average queue size, and is presented in Figure 5(B).

TPR is the product of the step interval and the number of steps needed in a round. The step interval is a parameter that can be controlled, whereas the number of steps is an indicator of the behavior of the protocol. All the lines in Figure 6(A) show the same trend; as the step interval increases, the number of step decreases to a minimum and remains there with any further increase in the step interval. The step interval at which the minimum number of steps is reached is called the *critical point*.



(A): number of steps needed in a round (B): Time Per Round

Figure 6: Simulation I (no message loss) with a dense group of size 200.

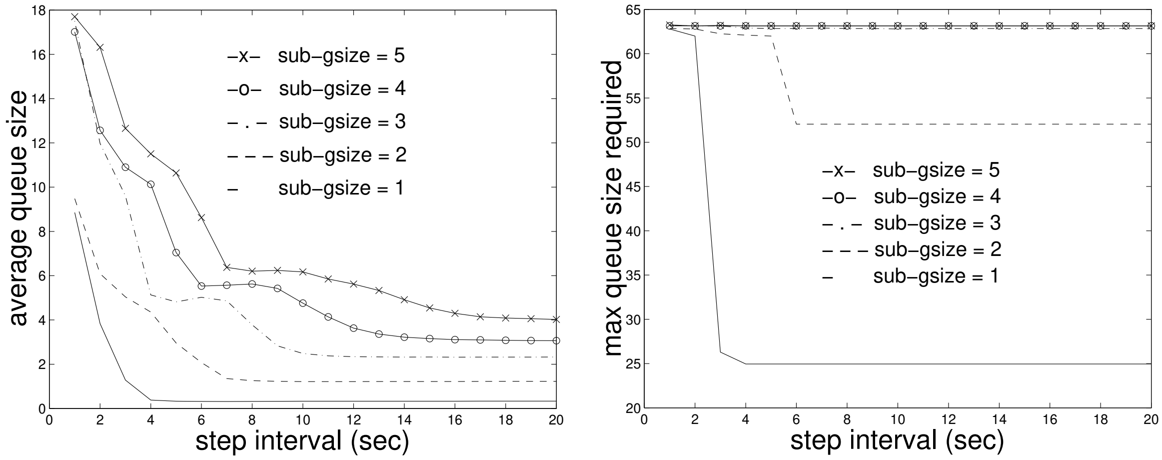
For a given sub-group size, before reaching the critical point, a decrease in the step interval results in an increase in the number of steps. As the step interval decreases, each member is scheduled to gossip its sequence number array in a shorter time period. This shorter time period prevents each member from receiving all the available new information. Therefore, more steps are needed to detect message stability. Moreover, the less new information in each step, the more redundant or repeating information flows in the network, increasing network traffic load and postponing the arrival of new information. These factors contribute to the increase in the number of steps needed for detecting message stability.

For a fixed step interval less than 4 seconds, that is, before the critical point is reached for any curve, the number of steps increases with the sub-group size because traffic load increases from more messages sent out by each member during each gossip. After the critical point is reached for all the curves, the step interval is greater than 15 seconds, and gossip messages are scattered far enough to reduce both traffic load and redundant gossips to a minimum. In this situation, the larger the sub-group size, the more information is exchanged in each step, and a smaller number of steps are needed to reach message stability. When the step interval varies between 4 and 15 seconds, the number of steps goes through a transition from increasing to decreasing with the sub-group size.

The behavior of the TPR curves plotted in Figure 6(B) can be derived from Figure 6(A), because TPR is the product of the number of steps and the step interval. Before reaching the critical point, the steeply declining trend of the number of steps dominates the behavior of TPR, even though the increasing step interval damps the TPR's decline. After passing the critical point, TPR becomes a linear function of the step interval with the coefficient being the value of the number of steps, which is almost a constant. As Figure 6(A) shows, the smaller the sub-group size, the larger the number of steps needed for detecting stability when the step interval passes the critical point. This feature transfered into Figure 6(B) says that the smaller the sub-group size, the steeper the slope of the TPR function is. When the step interval falls in the range between 4 and 15 seconds, the opposite movements of the two components of the TPR function cause TPR to fluctuate in a narrow range from 28 to 40 seconds, except for the case of sub-group size one. For each sub-group size, a window of optimal step intervals exists in which TPR is near-minimum.

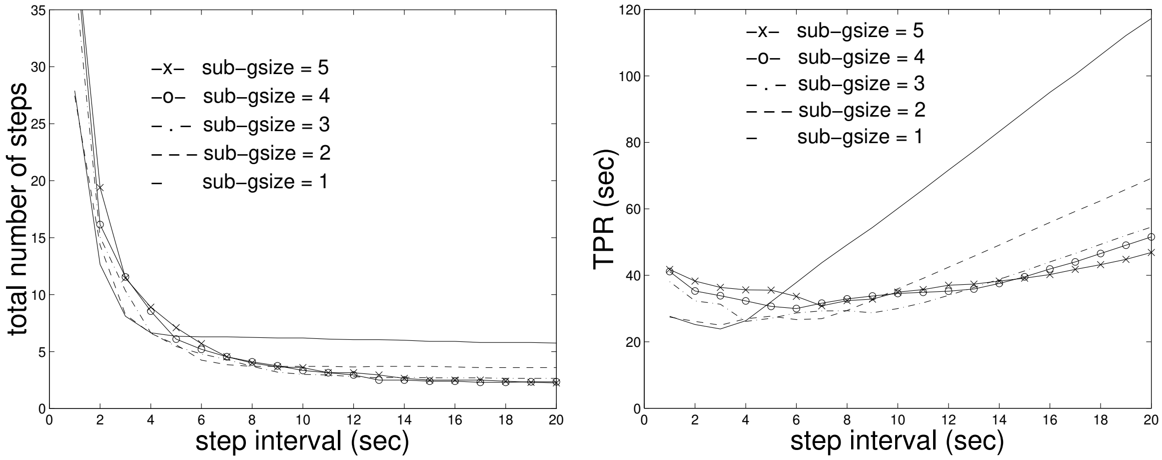
4.2.2 Simulation II of dense groups

Figures 7 and 8 present the four indices in simulation II for a dense group of size 200. When the step interval is smaller than the critical point, a large number of messages are dropped at intermediate and destination nodes because of the 64-message buffer limit. Whereas in simulation I, a lot of bandwidth is wasted carrying gossip messages that have no effect in the determination of stability. This explains the better TPR observed in simulation II than simulation I.



(A): average queue size (B): maximum queue size

Figure 7: Simulation II (queue size = 64) with a dense group of size 200.

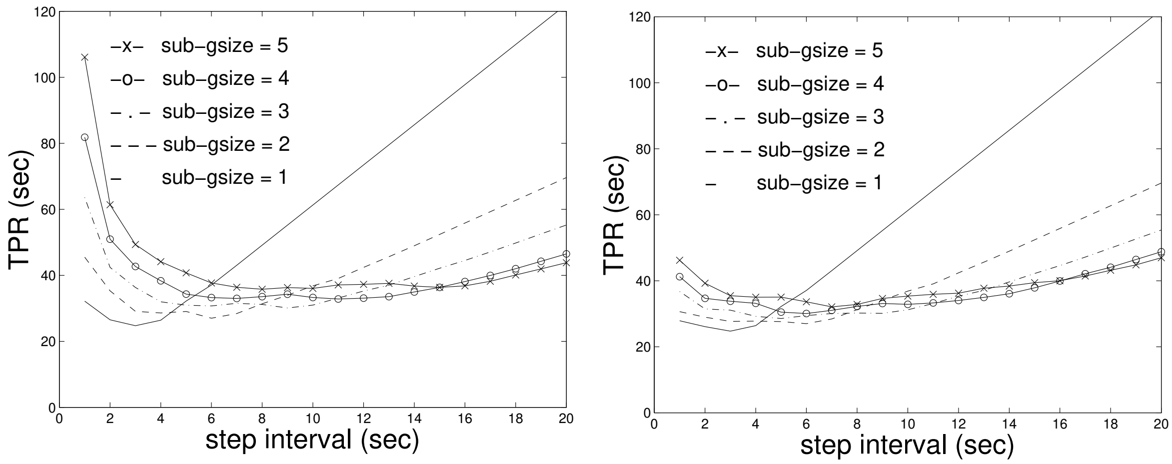


(A): number of steps needed in around (B): Time Per Round

Figure 8: Simulation II (queue size = 64) with a dense group of size 200.

4.2.3 Simulations of sparse groups

The same simulation for a dense group and a sparse group of the same size shows no major difference in all the performance indices. This indicates that location of group members and network topology do not have a noticeable effect on the protocol. This is the result of the combination of the following factors: processing speed, bandwidth and propagation delay. Under the given network condition, bandwidth limitation is the dominating factor in TPR when the step interval is smaller than the critical point, and the length of the step interval dominates TPR when it is larger than the critical point. TPRs for the sparse group simulations are presented in Figure 9.



(A): Simulation I (B): Simulation II

Figure 9: TPR for Simulation I and II with 20 sparse groups of size 200.

4.3 Adaptive method: finding the window of optimal step intervals

As observed from Figures 6(B), 8(B), and 9, every TPR curve has a flat portion in which one can select any step interval to achieve a near-minimum TPR. The process of finding the flat portion has two parts. The first part finds a step interval that achieves a near-minimum TPR, and the second part finds a window around that step interval. If TPR can be expressed as an analytical function, then Newton's method [3] can find its minimum. Otherwise, better approaches exist in experimental optimization. One such approach is the *golden-ratio method* [9]. Figures 10 and 11 present the two-part algorithm for finding the window of optimal step intervals.

Different groups and network topology require different input values for the algorithm. For example, the following input values are given for a 200-member dense group in simulation I: $g = 2$ seconds, $a_0 = 0$, $b_0 = 20$ seconds, $e = 10\%$, and $p = 1$ second. For the TPR curve for sub-group size of 3, part one finishes after 5 iterations, ending up with 9 seconds as the step interval that achieves a minimum TPR of 28.6 seconds. Part two finds the lower bound $s_1 = 4$ seconds and the upper bound $s_2 = 10$ seconds after total of 5 iterations of the while loops. Any step interval in the range from 4 to 10 seconds can be used to achieve a TPR between 28.6 to 31.2 seconds. The window sizes are different for different sub-group sizes. For sub-group size of 1, the window is only from 2 to 4 seconds. As the group size and network condition change over time, this two-part algorithm is executed periodically to find the current window of optimal step intervals.

As observed from Figures 6(B), 8(B), and 9, as the sub-group size increases, the minimum TPR increases slightly, but the window size of optimal step intervals increases significantly. There is a trade-off between the stability of the protocol and the minimum TPR. If the window size is too small, for example, when sub-group size is 1, then a slight perturbation of the network condition will result in dramatic increase of TPR if the step interval is unchanged. A large window size is preferred because slight changes in network condition will result in overlapping between the new and current windows, therefore only slight changes in TPR. Also notice the optimal window size is about the same for sub-group size of 4 and 5. As a result, the recommended sub-group size is 2, 3 or 4.

```
Input:   a0: smallest step interval in the search.
         b0: largest step interval in the search.
         g: distance between the two ends when the search stops.
Output:  s: the step interval that achieves a near-minimum TPR.

a := a0; b := b0; stop := false;
while ¬stop do
    d := b − a;
    if (d > g) then
        m1 := a + 0.382 ∗ d;
        m2 := a + 0.618 ∗ d;
        if (f(m1) ≥ f(m2)) then a := m1; else b := m2;
    else
        stop:= true;
        s := (a + b)/2;
```

Figure 10: Part I of the adaptive algorithm: finding a near-minimum TPR. $f(x)$ is the average measured TPR value for a step interval value x.

Input: s: *the step interval that achieves a near-minimum TPR from part I.*
$f(s)$: *the near-minimum TPR from part I.*
e: *the fluctuation index. Step intervals that achieve TPR in the range from* $(1-e)*f(s)$ *to* $(1+e)*f(s)$ *should be in the window.*
p: *initial search step size.*
Output: s_1: *lower bound of the window.*
s_2: *upper bound of the window.*

```
s1 := s; k := 1;
while ((s1 − k ∗ p > 0) ∧ (|f(s1 − k ∗ p) − f(s)| < e)) do
    s1 := s1 − k ∗ p; k := 2 ∗ k;
k:= k/2;
while ((k ≥ 1) ∧ (|f(s1 − k ∗ p) − f(s)| < e)) do
    s1 := s1 − k ∗ p; k := k/2;

s2 := s; k := 1;
while (|f(s2 + k ∗ p) − f(s)| < e) do
    s2 := s2 + k ∗ p; k := 2 ∗ k;
k:= k/2;
while ((k ≥ 1) ∧ (|f(s2 + k ∗ p) − f(s)| < e)) do
    s2 := s2 + k ∗ p; k := k/2;
```

Figure 11: Part II of the adaptive algorithm: finding the optimal step interval window given a near-minimum TPR. $f(x)$ is the average measured TPR value for a step interval value x.

4.4 Simulations with different group sizes

Simulations I and II are conducted for dense and sparse groups with various group sizes, and the same pattern is observed as in the tests for 200 members. Simulation II for sparse groups is done for different group sizes in a balanced bounded-degree tree of size 1000, and its near-minimum TPR is presented in Figure 12.

The group sizes are 50, 100, 200, 300, 400 and 500. For each group size, 20 simulations are conducted with sub-group size of 1 and an optimal step interval. For each simulation, a new group is randomly constructed in the 1000-node tree. Each simulation is represented by a dot in Figure 12. The solid line represents the mean of the TPRs. The minimum TPR increases linearly with group size n. When $n = 500$, it reaches 71 seconds.

The simulations are run with $30K$ bps network bandwidth allocated to the protocol. The same simulations are also conducted with a $300K$ bps bandwidth and a 10 time decrease in TPR is observed. More bandwidth combined with an optimal step interval will result in a faster stability detection time.

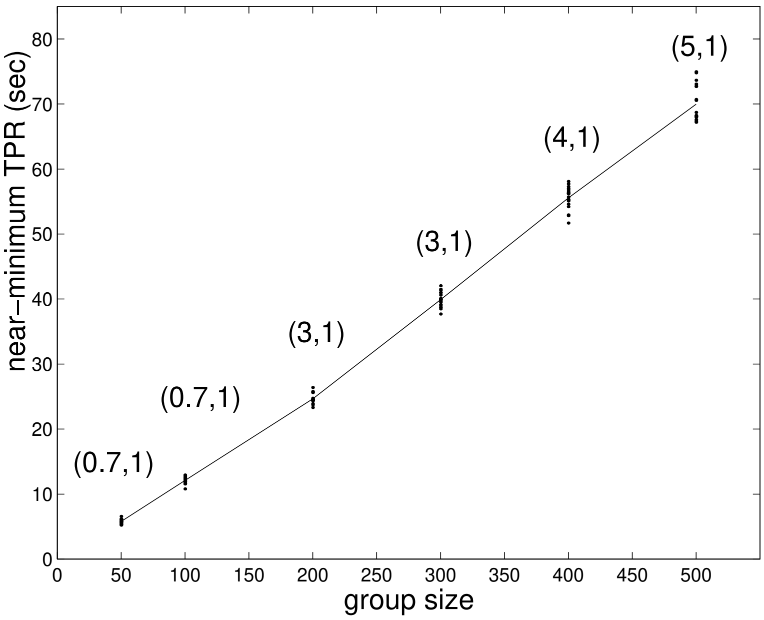


Figure 12: Near-minimum TPR for sparse groups in Simulation II using the global gossip scheme.

5 Extending the basic approach – local gossip

With the GSGC's global stability detection framework described above, a member gossips to a random set of members during each step. As a result, TPR increases linearly with the group size no matter how dense or sparse the group is on the network. In the global scheme, every group member on a subnet propagates their sequence number information by sending gossip messages to some random remote members. A more efficient way would be to combine and compress their information first into one sequence number array of the local group, and then allow one or several designated members to send this array to remote members.

We propose a hierarchical scheme to apply the GSGC protocol. The multicast group is divided into a number of local groups according to their location on the network. Since many reliable multicast protocols employ built-in local groups [10, 15], their group division can be used by the GSGC protocol. Each local group has G Stability Controllers (SCs) where G is a user-defined parameter which determines how robust the protocol is in case of SC failures. The protocol proceeds in two phases. In the first phase, each member gossips to other members in its local group trying to obtain stability information within the local group. After the local stability arrays are constructed, the SCs start the second phase by gossiping among all the SCs. After one SC receives the stability information from SCs representing the other local groups, the global stability array is constructed and multicast to the entire group.

The following simulation is conducted to show the effectiveness of the local gossip scheme.

The group size n ranges from 50 to 500. For each n, a sparse group of size n is built in a 1000-node balanced bounded-degree tree. The group is randomly divided into $n/50$ local groups of size 50, and in each local group, 5 SCs are randomly chosen. For gossiping in each local group of size 50, sub-group size of 1 and an optimal step interval of 700 milliseconds are chosen to achieve a near-minimum TPR. When n is 100, 200, 300, 400, and 500, the size of the SC group is 10, 20, 30, 40, and 50 respectively, and the optimal step interval is 200, 300, 500, 600, and 700 milliseconds respectively. The near-minimum TPRs achieved by the local gossip scheme is plotted in Figure 13. Compared with the global scheme, the hierarchical scheme dramatically reduces TPR.

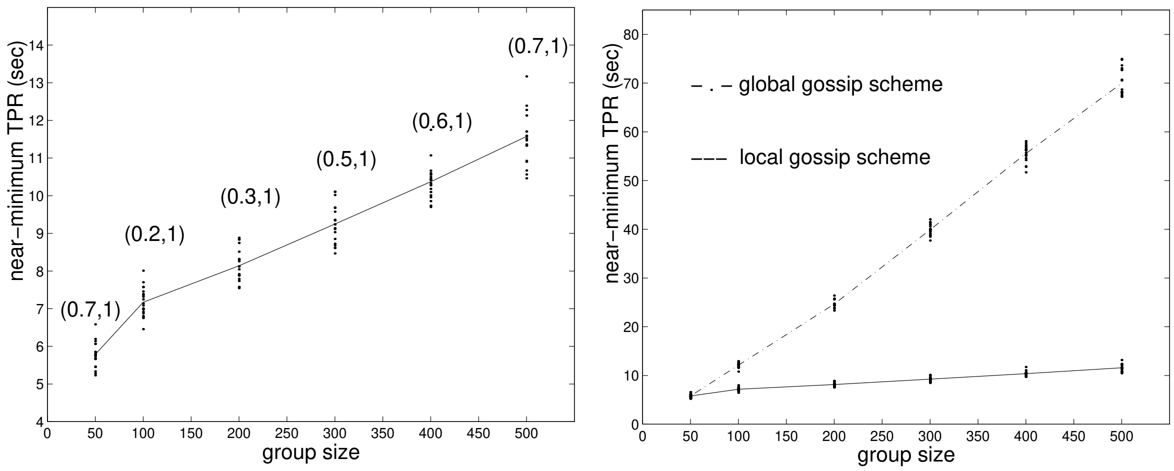


Figure 13: Near-minimum TPR for sparse groups using the local gossip scheme.

This significant improvement in performance comes from two factors. One, a hierarchy is built in the communication structure. Two, stability information is combined and compressed; all the stability arrays of members from each local group are combined into one stability array of the same size, effectively reducing the traffic load. This technique applies to any information gathering protocol to improve performance and scalability.

Ideally, the local groups should be constructed based on the location of the group members in the network and the SCs should be selected dynamically based on the load of group members and network topology. As a result, the performance of the protocol can be improved. This is currently being investigated.

The global scheme requires group membership information at every member. This is not likely to be feasible in a WAN. The local gossip scheme solves this problem by only requiring each local group member to maintain the addresses of other members on the same subnet.

A multi-level hierarchy can be built on top of the basic gossip scheme, further reducing TPR.

But a trade-off exists between stability detection time and the complexity and fault-tolerance of the protocol. For example, local group members need to know the local group membership, and SCs need to know who the other SCs are in order to gossip to them. The resulting protocol is more complex. One can trade this complexity with the $log(n)$ increase of TPR instead of the linear increase in the global scheme. However, the hierarchies in reliable multicast protocols can be used for the stability detection protocol. In this case, the complexity is pushed out of the GSGC scheme.

6 Conclusion and future work

The GSGC protocol achieves the twin goals of fault tolerance and scalability. It tolerates message losses without requiring a reliable multicast protocol underneath, by periodically sending gossip messages to random sets of group members. This scheme overcomes routing errors, transient link failures and omission failures, because messages are randomly sent to other members, and a message may take a different route in different steps.

In two ways, it tolerates group membership changes caused by member crashes, leaves, or joins. First, it incorporates a gossip style failure detection protocol. Second, it propagates membership changes throughout the entire group using gossip messages, and the normal operation of each group member is not affected when membership changes.

The scalability of GSGC protocol comes from four features. First, the state information maintained at each multicast participant is minimal. Each member keeps the current group membership, the current round number, an m-element sequence number array, an m-element "Min-so-far" array, and an n-element "Whom-I've-heard-from" bitmap array.

Second, the message size does not increase linearly with the group size. As shown in the simulations, the majority of a gossip message is occupied by the "Min-so-far" arrays of size proportional to the number of senders. Only a small portion is a bitmap array which is of the size of the group size n in bits. (When $n = 1000$, it occupies 125 bytes.) Commonly, the number of senders is much less than the group size, thus the message size does not increase linearly with group size. For instance, in a group of 1000 members, the message size is still less than 400 bytes.

Third, group membership changes do not affect operation of the stability detection protocol. Most applications in a WAN environment do not require immediate actions by group members when some member crashes or leaves the group, and late joiners need to receive enough information before they start participating in the failure detection/membership protocol. Although the current group membership information is maintained at each member for the failure detection protocol, agreement and instant update of group membership in the GSGC protocol is not required. A similar technique is used in routing table updates. It is necessary to propagate route changes to all the routers without shutting down the network.

Fourth, no hot spot exists in the protocol, because no member receives a lot of messages in a short amount of time. The protocol is completely free of the implosion problem.

Under the receiver reliable multicast model where group members do not know the individual addresses of other members, one has to rely on other schemes to construct sub-groups for gossip messages. One such scheme that uses the Time-To-Live field in IP packets to limit the scope of a gossip is currently under investigation.

Acknowledgments

We would like to thank S. Keshav for many insightful discussions.

References

[1] R. Ahuja, S. Keshav, and H. Saran. Design, implementation, and performance of a native mode ATM transport layer. *IEEE/ACM Transactions on Networking*, 4(4):502–515, August 1996.

[2] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 9(12):36–53, December 1993.

[3] J. E. Dennis, Jr. and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice Hall Series in Computational Mathematics. Prentice-Hall, Inc., 1993.

[4] M. Doar and I. Leslie. How bad is naïve multicast routing? In *Proceedings of IEEE INFOCOM'93*, pages 82–89, San Francisco, CA, March 1993.

[5] D. Dolev, D. Malki, and R. Strong. An asynchronous membership protocol that tolerates partitions. Research report, The Hebrew University of Jerusalem, 1993.

[6] A. Erramili and R. P. Singh. A reliable and efficient multicast protocol for broadband broadcast networks. In *Proceedings of ACM SIGCOMM'87*, pages 343–352, Stowe, VT, August 1987.

[7] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, November 1996.

[8] K. Guo, R. van Renesse, W. Vogels, and K. Birman. Hierarchical message stability tracking protocols. Technical Report CS-TR 97, Department of Computer Science, Cornell University, August 1997.

[9] *Handbook of Mathematics*. High Education Publishing House, Beijing, 1979.

[10] H. Holbrook, S. Singhal, and D. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. In *Proceedings of ACM SIGCOMM'95*, pages 28–341, Cambridge, MA, August 1995.

[11] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of ACM SIGCOMM'93*, San Francisco, CA, September 1993.

[12] S. McCanne and S. Floyd. Ns (network simulator). Available via http://www-nrg.ee.lbl.gov/ns, 1995.

[13] E. Palmer. *Graphical Evolution: An Introduction to the Theory of Random Graphs.* John Wiley & Sons, 1985.

[14] Transmission control protocol. RFC 793, September 1981.

[15] K. Sabnani, J.C. Lin, S. Paul, and S. Bhattacharyya. Reliable Multicast Transport Protocol(RMTP). *IEEE Journal on Selected Areas in Communication*, April 1997.

[16] R. van Renesse. Masking the overhead of protocol layering. In *Proceedings of ACM SIGCOMM'96*, pages 96–104, August 1996.