Bimodal Multicast[1]

Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao,
Mihai Budiu, Yaron Minsky

There are many methods for making a multicast protocol "reliable". At one end of the spectrum, a reliable multicast protocol might offer atomicity guarantees, such as all-or-nothing delivery, delivery ordering, and perhaps additional properties such as virtually synchronous addressing. At the other are protocols that use local repair to overcome transient packet loss in the network, offering "best effort" reliability. Yet none of this prior work has treated stability of multicast delivery as a basic reliability property, such as might be needed in an internet radio, TV, or conferencing application. This paper looks at reliability with a new goal: development of a multicast protocol which is reliable in a sense that can be rigorously quantified and includes throughput stability guarantees. We characterize this new protocol as a "bimodal multicast" in reference to its reliability model, which corresponds to a family of bimodal probability distributions. Here, we introduce the protocol, provide a theoretical analysis of its behavior, review experimental results, and discuss some candidate applications. These confirm that bimodal multicast is reliable, scalable, and that the protocol provides remarkably stable delivery throughput.

Keywords: Bimodal Multicast, probabilistic multicast reliability, scalable group communications, isochronous protocols, internet media transmission.

Encamped on the hilltops overlooking the enemy fortress, the commanding general prepared for the final battle of the campaign. Given the information he was gathering about enemy positions, his forces could prevail. Indeed, if most of his observations could be communicated to most of his forces the battle could be won even if some reports reached none or very few of his troupes. But if many reports failed to get through, or reached many but not most of his commanders, their attack would be uncoordinated and the battle lost, for only he was within direct sight of the enemy, and in the coming battle strategy would depend critically upon the quality of the information at hand.

Although the general had anticipated such a possibility, his situation was delicate. As the night wore on, he dispatched wave upon wave of updates on the enemy troupe placements. Some couriers perished in the dark, wet forests separating the camps. Worse still, some of his camps were beset by the disease that had ravaged the allies since the start of the campaign. Chaos and death ruled there; they could not be relied upon.

With the approach of dawn, the general sat sipping coffee – rotgut stuff – reflectively. In the night, couriers came and went, following secret protocols worked out during the summer. At the appointed hour, he rose to lead the attack. The general was not one to shirk a calculated risk.

[1] This work was supported by DARPA/ONR contracts N0014-96-1-10014 and ARPA/RADC F30602-96-1-0317, the Cornell Theory Center and the Turkish Research Foundation. Our web site is http://www.cs.cornell.edu/Info/Projects/Ensemble/; all software described here is available for public use. Contact information for first author: Kenneth P. Birman; Dept. of Computer Science; Cornell University; Ithaca, New York 14853. Email ken@cs.cornell.edu.

1. Introduction

Although many communication systems provide software support for reliable multicast communication, the meaning given to “reliability” splits them into two broad classes. One class of definitions corresponds to “strong” reliability properties. These typically include *atomicity*, which is the guarantee that if a multicast is delivered to any destination that remains operational it will eventually be delivered to all operational destinations. An atomic multicast may also provide message delivery ordering, support for virtual synchrony (an execution model used by many group communication systems), security properties, real-time guarantees, or special behavior if a network partitioning occurs [Birman97]. A criticism is that to obtain these strong reliability properties one employs costly protocols, accepts the possibility of unstable or unpredictable performance under stress, and tolerates limited scalability ([CS93], but see also [OSR94]). As we will see shortly, transient performance problems can cause these protocols to exhibit degraded throughput. Even with a very stable network, it is hard to scale these protocols beyond several hundred participants [PC97].

Protocols belonging to the second class of "reliable multicast" solutions focus upon best-effort reliability in very large-scale settings. Examples include the Internet MUSE protocol (for network news distribution) [LOM94], the Scalable Reliable Multicast protocol (SRM) [FJMLZ95], the XPress Transfer Protocol [XTP95], and the Reliable Message Transport Protocol (RMTP) [LP96, PSLM97]. These systems include scalable multicast protocols which overcome message loss or failures, but are not provided with an "end to end" reliability guarantee. Indeed, as these protocols are implemented, "end to end" reliability may not be a well-defined concept. There is no core system to track membership in the group of participants, hence it may not be clear what processes belong to the destination set for a multicast, or even whether the set is small or large. Typically, processes join anonymously by linking themselves to a multicast forwarding tree, and subsequently interact only with their immediate neighbors. Similarly, a member may drop out or fail without first informing its neighbors.

The reliability of such protocols is usually expressed in "best effort" terminology: if a participating process discovers a failure, a reasonable effort is made to overcome it. But it may not always be possible to do so. For example, in SRM (the most carefully studied among the protocols in this class) a router overload may disrupt the forwarding of multicast messages to processes down-stream from the router. If this overload also prevents negative acknowledgements and retransmissions from getting through for long enough, gaps in the message delivery sequence may not be repaired. Liu and Lucas report conditions under which SRM can behave pathologically , re-multicasting each message a number of times that rises with system scale [Liu97, Lucas98]; here, we present additional data of a similar nature. (Liu also suggests a technique to improve SRM so as to partially overcome the problem). The problematic behavior is triggered by low levels of system-wide noise or by transient elevated rates of message loss, phenomena known to be common in Internet protocols [LMJ97, Paxson97]. Yet SRM and similar protocols do scale beyond the limits of the virtual synchrony protocols, and when messages loss is sufficiently uncommon, they can give a very high degree of reliability.

In effect, the developer of a critical application is forced to choose between reduced scalability but stronger notions of reliability in the first class of reliable multicast protocol, and weaker guarantees but better normal-case scalability afforded by the second class. For critical uses, the former option may be unacceptable because of the risk of a throughput collapse under unusual but not "exceptionally rare" conditions. Yet the latter option may be equally unacceptable, because it is impossible to reason about the behavior of the system when things go wrong.

The present paper introduces a new option: a bimodal multicast protocol that scales well, and also provides predictable reliability even under highly perturbed conditions. For example, the reliability and throughput of our new protocol remain steady even as the network packet loss rate rises to 20% and even when 25% of the participating processes experience transient performance failures. We also present data showing that the LAN implementation of our protocol overcomes bursts of packet loss with minimal disruption of throughput.

The sections that follow start by presenting the protocol itself and some of the results of an analytic study (the details of the analysis are included as an Appendix). We show that the behavior of our protocol can be predicted given simple information about how processes and the network behave most of the time, and that the reliability prediction is strong enough to support a development methodology that would make sense in critical settings. Next, we present a variety of data comparing our new protocol with prior protocols, notably a virtually synchronous reliable multicast protocol, also developed by our group, and the SRM protocol. In each case we use implementations believed to be the best available in terms of performance and tuned to match the environment. Our studies include a mixture of experimental work on an SP2, simulation, and experiments with a bimodal multicast implementation for LAN's (possibly connected by WAN gateways). Under conditions that cause other reliable protocols to exhibit unstable throughput, bimodal multicast remains stable. Moreover, we will show that although our model makes simplifying assumptions, it still makes accurate predictions about real-world behavior. Finally, the paper examines some critical reliable multicast applications, identifying roles for protocols with strong properties and roles for bimodal multicast.

Bimodal multicast is not a panacea: the protocol offers a new approach to reliability, but uses a model that is weaker in some ways than virtual synchrony, despite its stronger throughput guarantees. We see it as a tool to offer side-by-side with other reliability tools, but not as a solution that "competes" with previous protocols.

2. Multicast throughput stability

In our work with reliable multicast we participated in the development of communication infrastructures for applications such as stock markets (the New York and Swiss Stock Exchanges) and air traffic control (the French console replication system called PHIDEAS) [Birman99, PC97, PHIDEAS]. The critical nature of such applications means that developers will have to know exactly how their systems behave under expected operational conditions, and to do so they need detailed information about how the reliable communication primitives they use will behave. These applications also demand high performance and scalability. In particular, they often have a data transport subsystem that will produce a sustained, fairly high volume of data considered critical for safe operation. In the past, such subsystems were often identified as critical real-time applications, but today's computers and networks are so fast that the real need is for stable throughput.

Our new protocol permits designers of such applications to factor out the soft real-time data stream. Bimodal multicast will handle this high volume workload, leaving a less demanding lower-volume residual communication task for protocols like the virtual synchrony ones, which work well in less stressful settings. Because the communication demands of bimodal multicast can be predicted from the multicast rate and a small set of parameters, a designer can anticipate that bimodal multicast will consume a fixed percentage of available bandwidth and memory resources, and configure the system with adequate time for the virtual synchrony mechanisms.

Bimodal multicast is a good choice for this purpose, for several reasons. First, as just noted, the load associated with the protocol is predictable and largely independent of scale. The protocol can be shown to have a bimodal delivery guarantee: given information about the environment – information that we believe is reasonable for typical networks running standard Internet protocols – our protocol can be configured to have a very small probability of delivering to a small number of destinations (counting failed ones), an insignificant risk of delivering to "many" but not "most" destinations, and a very high probability of delivering the message to all or almost all destinations. Our model lets us tune the actual probabilities to match the intended use. And we will show how to use the model to evaluate the safety of applications, such as the ones mentioned above.

Secondly, our protocol has stable throughput. Traditional reliable multicast protocols– atomic broadcast in its various incarnations – suffer from a form of interference between flow control and reliability mechanisms. This can trigger unstable throughput when the network is scaled up, and some application programs exhibit erratic behavior. We are able to demonstrate the stability of our new protocol both theoretically and experimentally. For the types of applications that motivate our work, this stability guarantee is extremely important: one needs to know that the basic data stream is delivered at a steady rate *and* that it is delivered reliably.

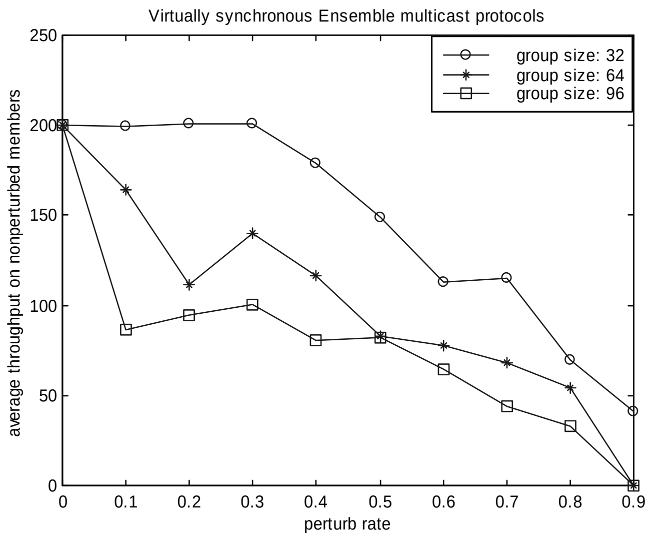


Figure 1: Throughput as one member of a multicast group is "perturbed" by forcing it to sleep for varying amounts of time.

To give some sense of where the paper is headed, consider Figure 1. Here we measured throughput at a healthy process in virtually synchronous multicast groups of various size: 32, 64 and 96 members. One of these members attempts to inject 7k-byte multicast messages at a rate of 200/second. Ideally, 200 messages per second emerge. But the graph shows that as we "perturb" even a single group member by causing it to sleep for the percentage of each second shown on the x-axis, throughput collapses for the unperturbed group members. The problem becomes worse as the group grows larger (it would also be worse if we increase the percentage of perturbed members). In the experimental sections of this paper, we will see that the bimodal multicast achieves the "ideal" output rate of 200 msgs/sec under the same conditions, even with 25% of the members perturbed. Details of the experiment used to produce Figure 1 appear in the experimental section of this paper.

As mentioned earlier, studies of SRM have identified similar problems. In the case of SRM, network-wide noise and routing problems represent the worst case. For example, Lucas, in his doctoral dissertation [Lucas98], shows that even low levels of network noise can cause SRM to broadcast high rates of retransmission requests and retransmitted data messages, so that each multicast triggers multiple messages on the wire. Lucas finds that the rate of retransmissions rises in proportion to the SRM group size. Liu, studying other problems (but of a similar nature) proposes a number of changes to SRM that might improve its behavior in noisy networks. Our own simulations, included in the experimental section of this paper, confirm these problems and make it clear that as SRM is scaled up, the protocol will eventually collapse, much as does the virtually synchronous protocol shown in Figure 1.

What causes these problems? In the case of the virtually synchronous protocols, a perturbed process is particularly difficult to accommodate. On the one hand, the process is not considered to have failed, since it is sending and receiving messages. Yet it is slow to acknowledge messages and may experience high loss rates, particularly if operating systems buffers fill up. The sender and healthy receivers keep copies of unacknowledged messages until they get through, exhausting available buffering space and causing flow control to kick in. One could imagine setting failure detection parameters more aggressively (this is what [PC97] recommends), but now the risk of an erroneous failure classification will rise roughly as the square of the group size. The problem is that all group members can be understood as monitoring one-another, hence the more aggressive the failure detector, the more likely that a paging or scheduling delay will be interpreted as a crash. Thus, as one scales these protocols beyond a group size of about 50-100 members, the tension between throughput stability and failure detection accuracy becomes a significant problem. Not surprisingly, most successes with virtual synchrony use fairly small groups, sometimes structured hierarchically. The largest systems are typically ones where performance demands are limited to short bursts of multicasts, far from the rates seen in Figure 1 [Birman99].

Turning to SRM, one can understand the problem as emerging from a form of stochastic attack on the probabilistic assumptions built into the protocol. Readers familiar with SRM will know that the protocol includes many such assumptions: they are used to prevent duplicated multicasts of requests for retransmission and duplicated retransmissions of data, and to estimate the appropriate time-to-live (TTL) value to use for each multicast. Such assumptions have a small probability of being incorrect, and in the case of SRM, as the size of the system rises, the absolute likelihood of mistakes rises, causing the background overhead to rise. Eventually, these forms of overhead interfere with normal system function, causing throughput to become unstable. For sufficiently large configurations or loads, they can trigger a form of "meltdown".

We believe that our paper is among the first to focus on stable throughput in reliable multicast settings. Historically, reliable multicast split early into two "camps." One camp focused on performance and scalability, emphasizing peak performance under ideal situations. The other camp focused on providing rigorous definitions for reliability and protocols that could be proved to implement their reliability specifications. These protocols tended to be fairly heavyweight, but performance studies also emphasized their best-case performance. Neither body of work viewed stable scalable throughput as a fundamental reliability goal, and as we have seen, stable throughput is not so easily achieved.

The properties of the bimodal multicast protocol seem to be ideal for many of the applications where virtual synchrony encounters its limits. These include internet media distribution (such for radio and TV broadcasts, or conferencing), distribution of stock prices and trade information on the floor of all-electronic stock exchanges, distribution of flight telemetry data in air traffic control systems and military theatre of operations systems, replication of medical telemetry data in critical care systems, and communication within factory automation settings. Each of these is a setting within which the highest volume data sources have an associated notion of "freshness", and the importance of delivering individual messages decreases with time. Yet several are also "safety critical" applications for which every component must be amenable to analysis.

3. A bimodal multicast protocol

Our protocol is an outgrowth of work which originated in studies of *gossip* protocols done at Xerox [DJHI87], the Internet MUSE protocol [LOM94], the SRM protocol of Floyd *et. al.* [FJML95], the NAK-only protocols used in XTP [XTP95], and the lazy transactional replication method of [LLSG92]. Our protocol can be understood as offering a form of weak real-time guarantee; relevant prior work includes [CASD85] and [BMR96, BPRS96].

The idea underling gossip protocols dates back to the original USENET News protocol, NNTP, developed in the early 1980's. In this protocol, a communications graph is superimposed on a set of processes, and neighbors gossip to diffuse News postings in a reliable manner over the links. For example, if process A receives a News posting and then establishes communication with process B, A would offer B a copy of that News message, and B solicits the copy if it has not already seen the message.

The Xerox work considered gossip communication in the context of a project developing wide-area database systems [DJHI87]. They showed how gossip communication is related to the mathematics underlying the propagation of epidemics, and developed a family of gossip-based multicast protocols. However, the frequency of database updates was low (at most, a few per second), hence the question of stable throughput did not arise. The model itself considered communication failures but not process failures. Our work addresses both aspects.

In this paper, we actually report on multiple implementations of our new protocol. The version we study most carefully was implemented within Cornell University's Ensemble system [Hayden98], which offers a modular plug-and-play framework that includes some of the standard reliable multicast protocols, and can easily be extended with new ones. This plug-and-play architecture was important both because it lets our new work sit next to other protocols, and because it facilitated controlled experiments. Ensemble supports group-communication protocol stacks that are constructed by composing micro-protocols, an idea that originated in the Horus project [VBM96, Birman97].

Because we implemented this version of our protocol within Ensemble, the system model is greatly simplified. An Ensemble process group executes as a series of *execution periods* during each of which group membership is static, and known to all members (see Figure 2, where time advances from left to right, and each time-line corresponds to the execution of an individual process). One execution period ends and a new one begins when membership changes by the addition of new processes, or the departure (failure) of old ones. Below, we will discuss our new protocol for just a single execution period; this restriction is especially important in the formal analysis we present. The mechanisms whereby Ensemble switches from one period to the next depend only on knowledge of when the currently active set of multicast instantiations have terminated – *stabilized*. In our new protocol, this occurs at a given group member when that member garbage collects the multicasts known to it.

The second version of the protocol is more recent, and while we include some experimental data obtained using it, we won't discuss it separately. This implementation is the basis of a new system we are developing called Spinglass, and operates as a free-standing solution. Unlike the Ensemble solution, which we use primarily for controlled studies on an SP2 computer, the newer implementation runs on a conventional LAN, and is being extended to also run in a WAN. The protocols are not identical, but the differences are not of great importance in the present setting, and we will treat them as a single implementation. Spinglass uses gossip to track membership as well as to do

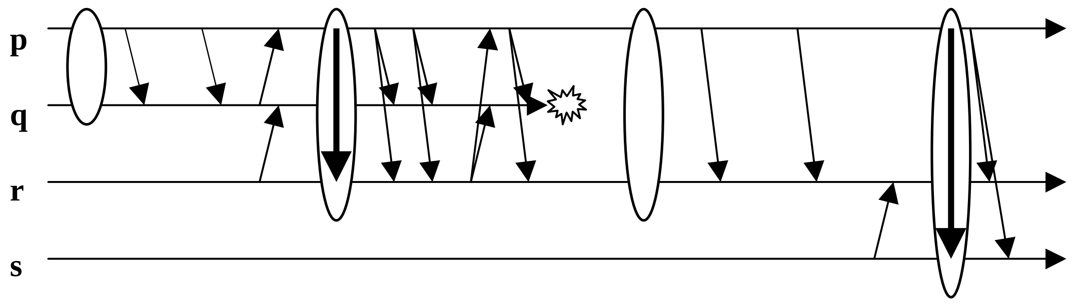


Figure 2: Multicast execution periods in Ensemble. Initially, the group consists of p and q; multicasts sent by p are delivered to q (and vice versa). r then joins and state is transferred to it. After a period of additional multicasting, q fails; s later joins and receives an additional state transfer. The period between two consecutive membership lists is denoted an "execution period".

communication [RMH98], but the behavior of the bimodal protocol is unaffected (formal analysis of the combined gossip mechanisms is, however, beyond our current ability).

In the remainder of this paper, we'll refer to our new protocol as *pbcast:* "probabilistic broadcast", since "bimodal multicast" and the obvious contractions seem awkward. A pbcast to a process group satisfies the following properties:

Atomicity: The protocol provides a bimodal delivery guarantee, under which there is a high probability that each multicast will reach almost all processes, a low probability that a multicast will reach just a very small set of processes, and a vanishingly small probability that it will reach some intermediate number of processes. The traditional "all or nothing" guarantee thus becomes "almost all or almost none".

Throughput stability: The expected variation in throughput can be characterized and, for the settings of interest to us, is low in comparison to typical multicast rates.

Ordering. Messages are delivered in FIFO order on a per-sender basis. Stronger orderings can be layered over the protocol, as discussed in [Birman97]. For example, [HB96] includes a protocol similar to pbcast with total ordering layered over it.

Multicast stability. The protocol detects stability of messages, meaning that the bimodal delivery guarantee has been achieved. A stable message can be safely garbage collected, and if desired, the application layer will also be informed. Protocols such as SRM generally lack stability detection, whereas virtual synchrony protocols include such mechanisms.

Detection of lost messages. Although unlikely at a healthy process, our bimodal delivery property does admit a small possibility that some multicasts will not reach some processes, and message loss is common at faulty processes. Should such an event occur, processes that did not receive a message are informed via an upcall. Section 5 discusses recovery from message loss.

Scalability. Costs are constant or grow slowly as a function of the network size. We will see that most pbcast overheads are constant as a function of group size, and that throughput variation grows slowly (with the log of the group size).

For purposes of analysis, our work assumes that the protocol operates in a network for which throughput and reliability can be characterized for about 75% of messages sent, and where network errors are *iid.* We assume that a correctly functioning process will respond to incoming messages within a known, bounded delay. Again, this assumption needs to hold only for about 75% of processes in the network. We also assume that bounds on the delays of network links are known. However, this last assumption is subtle, because pbcast is normally configured to communicate preferentially over low-latency links, as elaborated in Section 4.

Traditionally, the systems community has distinguished two types of failures. *Hard failures* include crash failures or network partitionings. *Soft failures* include the failure to receive a message that was correctly delivered (normally, because of buffer overflow), failure to respect the bounds for handling incoming messages, and transient network conditions that cause the network to locally violate its normal throughput and reliability properties. Unlike most protocols, which only tolerate hard failures, the goal of our protocol is to also overcome bounded numbers of soft failures with minimal impact on the throughput of multicasts sent by a correct process to other correct processes. Moreover, although this is not a "guarantee" of the protocol, a process that experiences a soft failure will (obviously) experience a transient deviation from normal throughput properties, but can then catch up with the others. Again, this behavior holds up to a bounded number of soft failure events. We do not consider Byzantine failures.

Although our assumptions may seem atypical of local area networks, the designer of a critical application such as an air-traffic control system would often be able to satisfy them. For example, a developer could work with LAN and WAN links isolated from uncontrolled traffic, high speed interconnects of the sort used in cluster-style scalable computers, or a virtually private network with quality of service guarantees.

But our work appears to be more broadly applicable, even in conventional networks. Here, we report on experiments with the Spinglass implementation of pbcast in normal networks, where we observed the protocol's behavior during bursts of packet loss of the sort triggered by transient network overloads. Although such events violate the assumptions of the model, the protocol continues to behave reliably and to provide steady throughput. One open question concerns the expected behavior of pbcast when running over WAN gateways spanning the public Internet, where loss rates and throughput fluctuate chaotically [LMJ97, Paxson97]. Based on our work up to the present, it seems that Spinglass can do fairly well by tunneling over TCP links between gateway processes, provided that enough such links are available. We conjecture that although the formal study of such configurations may be intractable, the idealized network model is considerably more robust than the assumptions underlying it would suggest.

4. Details of the Pbcast Protocol

Pbcast is composed of two sub-protocols structured roughly as in the Internet MUSE protocol [LOM94]. The first is an unreliable, hierarchical broadcast that makes a best-effort attempt to efficiently deliver each message to its destinations. Where IP-multicast is available, it can play this role. The second is a 2-phase anti-entropy protocol that operates in a series of unsynchronized rounds. During each round, the first phase detects message loss; the second phase corrects such losses and runs only if needed. This section describes the protocol; pseudo-code is included as Appendix B. We begin with a basic discussion but then introduce a series of important optimizations.

Optimistic Dissemination Protocol

The first stage of our protocol multicasts each message using an unreliable multicast primitive. This can be done using IP multicast or, if IP multicast is not available, using a randomized dissemination protocol. In the latter case, we assume full connectivity and superimpose "virtual" multicast spanning trees upon the set of participants. Each process has a variety of such pseudo-randomly generated spanning trees for broadcasting messages to the entire group; these are generated in a deterministic manner from the group membership using an inexpensive algorithm. When a member broadcasts a message, it is sent using a randomly selected spanning tree[2]. This is done by attaching a

[2] Although the tree that will be used for a given multicast execution period is not predictable prior to when

tree identifier (a small integer) to the message and sending it to the sender's neighbors in the tree. Upon receipt, those members deliver the message and then forward it using the tree identifier. This dissemination protocol can be tuned with respect to the number of random trees that are used and the degree of nodes in the tree. Because all of the messages are sent unreliably, the choice of tree should be understood purely as an optimization to quickly deliver the message to many members. If some members do not receive the message, the anti-entropy protocol still ensures probabilistically reliable delivery.

In the Ensemble implementation of pbcast, the tree-dissemination protocol uses Ensemble's group membership manager to track membership, but this also limits scalability. Ensemble's group membership system works well up to a hundred members or so, and can probably be scaled to a few hundred. As for the Spinglass version of pbcast, we currently uses a hand-configured multicast architecture, represented as a multicast routing table used by the protocol; again, this makes sense for a few hundred machines but probably not for larger networks. We see management of the multicast dissemination routes for pbcast as a topic for which additional study will be required.

2-Phase Anti-Entropy Protocol

The important properties of pbcast stem from its gossip-based anti-entropy protocol. The term *anti-entropy* is from [DGHI87] and refers to protocols that detect and correct inconsistencies in a system by continuous gossiping. Our anti-entropy protocol progresses through rounds in which members randomly choose other members, send a summary of their message histories to the selected process, and then solicit copies of any messages they discover themselves to be lacking to converge towards identical histories. This is illustrated in Figure 3: after a period during which messages are multicast unreliably (process Q fails to receive a copy of M_0 and process S fails to receive M_1, denoted by the dashed arrows), the anti-entropy protocol executes (gray region). At this time Q discovers that it has missed M_0 and requests a retransmission from P, which forwards it. S does not detect and repair its own loss until the subsequent anti-entropy round.

The figure oversimplifies by suggesting that the protocol alternates between multicasting and running anti-entropy rounds; in practice, the two modes are concurrent. Also, the figure makes the anti-entropy communication look regular; in practice, it is quite random. Thus, Q receives an anti-entropy message from P, but could have obtained it from R or S. Moreover, as a side effect of randomness, a process may not receive an anti-entropy message, at all, in a given round of gossip – here, S receives none, while Q receives two.

Our protocol differs from most prior gossip protocols in that pbcast emphasizes achieving a common suffix of message histories rather than a common prefix. In other words, our protocol prioritizes recovery of recent messages, and when a message becomes old enough the protocol gives up entirely and marks the message as lost. The advantage of this structure is that the protocol avoids scenarios where processes suffer transient failures and are subsequently unable to catch up with the rest of the system. In traditional gossip

that period begins, the same tree will be used throughout the duration of the execution period.

protocols, such a situation can cause the other processes' message buffers to fill, and the overall system to slow down. Our protocol avoids this behavior by eventually giving up on old messages, and instead emphasizing delivery of recent messages. However, even though messages may eventually be marked as "lost," a probabilistic analysis of the protocol shows that—when properly configured—this loss is unlikely to happen except at failed processes, or with messages sent by processes that failed as they sent them. Section 5 discusses our handling of such cases.

Thus, in our example, if process R had experienced a soft failure and missed messages M_0… M_4, it would learn about them in a subsequent anti-entropy message and would request retransmissions in reverse order: M_4 first, then M_3, and so forth. While pulling these messages from other processes, R would participate normally in new multicasts and new rounds of the anti-entropy algorithm.

The anti-entropy protocol is run by all processes in the system, and proceeds through a sequence of rounds. The length of each round must be larger than the typical round-trip time for an RPC over the communications links used by the protocol; in practice, we used much longer rounds which are a substantial fraction of a second in duration (for example, our experiments start a round every 100ms). Clocks need not be synchronized, but we will initially act as if they were for simplicity of the exposition. At the beginning of each round, every member randomly chooses another member with which it will conduct the anti-entropy protocol and sends it a digest (summary) of its message histories. The message is called a "gossip message." The member that receives this message compares the digest with the messages in its own buffers. If the digest contains messages that this member does not have, then it sends a message back to the original sender to request some messages to be retransmitted. This message is called the "solicitation," and causes the receiver to retransmit some of the messages.

Processes maintain buffers of messages that have been received from other members in the group. Every message is either delivered to the application, or if a message cannot be recovered during the retransmission protocol, an upcall is used to notify the application of missing messages. These events occur in FIFO order from each sender. When messages are received, they are inserted in the appropriate location in a message buffer.

Upon receiving a message, a process tags the message with the round in which the message was received. Any undelivered messages that are now in order are delivered. (In some situations, it might make sense to delay delivery until one or more rounds of gossip have been completed. Doing so would clearly reduce the risk that a very small number of

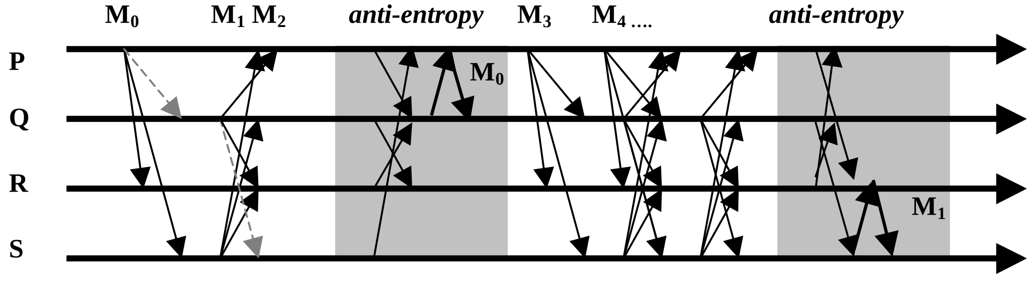


Figure 3: Illustration of the pbcast anti-entropy protocol.

processes deliver the pbcast, but we have not explored the details of such a change). The process will continue to gossip about the message until a fixed number of rounds after its initial reception, after which the message is garbage collected. This number of rounds and the number of processes to which a processes gossips in each round[3] are parameters to the protocol. The product of these parameters, called the *fanout*, can be tuned using the theory we develop in Appendix A (summarized in Section 6). If a process has been unable to recover a missing message for long enough to deduce that other processes will have garbage collected it, it gives up on that message and reports a gap (lost message) to the application layer.

There are several optimizations to the anti-entropy protocol that act to limit its costs under failure scenarios. Without these additions, a normal anti-entropy protocol is liable to enter fluctuating communication patterns whereby poorly performing processes or a noisy network can affect healthy processes, by swamping them with retransmission requests. Here, we summarize six important optimizations. In Section 7 we present experimental evidence that they achieve the desired outcome.

Optimization 1: Soft Failure Detection

Retransmission requests are only serviced if they are received in the same round for which the original solicitation was sent. If the response to a solicitation takes longer than a round (which is normally more than enough time) then the response is dropped. The failure of a process to respond to a solicitation within a round is an indication that the process or the network is unhealthy, and hence that a retransmission is unlikely to succeed. This also protects against cases where a process responds to many solicitations at once and causes the network to become flooded with redundant retransmissions.

Optimization 2: Round Retransmission Limit

The maximum amount of data (in bytes) that a process will retransmit in one round is also limited. If more data is requested than this, then the process stops sending when it reaches this limit. This prevents processes that have fallen far behind the group from trying to catch up all at once. Instead, the retransmission can be carried out over several rounds with several different processes, spreading the overhead in space and time.

Optimization 3: Cyclic Retransmissions

Processes responding to retransmission requests cycle through their undelivered messages, taking into account the messages that were requested in the previous rounds. If the request from the previous round was successful, but the messages might still be in transit, the response will include different messages, avoiding redundant retransmissions.

[3] In our experiments with the protocol, we find that the average load associated with the protocol is minimized if a process gossips to just one other process in each round, but one can imagine settings for which this would not be the case.

Optimization 4: Most-Recent-First Retransmission

Messages are retransmitted in the order of most-recent-first. Oldest-first retransmission requests can induce a behavior in which a temporarily faulty process trying to catch up recovers from its problem, but is left permanently lagging behind the rest of the group.

Optimization 5: Independent numbering of rounds

It may seem as if processes should advance synchronously through rounds, but the protocol actually allows each process to maintain its own round numbers and to run them asynchronously, which is how our implementation actually works. The insight is that the number of rounds that have elapsed is used to determine when to deliver or garbage collect a message, but this is entirely a local decision. The round number also enters in a gossip message and any subsequent solicitation to retransmit, but the solicitation can simply copy the round number used by the sender of the gossip message.

Optimization 6: Random graphs for scalability

If one assumes that large groups would use IP multicast for the unreliable multicast, the basic protocol presented above is highly scalable except in two dimensions. First, as stated, it would appear that each participating process needs a full membership list for the multicast group, since this information is used in the anti-entropy stages of the protocol. Such an approach implies a potentially high traffic of membership updates to process group members, and the list itself could become large. Second, in a wide-area use of the protocol, anti-entropy will often involve communication over high-latency communication paths. In a very large network the buffering requirements and round-length used in the protocol could then grow as a function of worst-case network latency.

Both problems can be avoided. A WAN is typically structured as a collection of LANs interconnected (redundantly) by TCP tunnels or gateways. In such an architecture, typical participants would only need to know about other processes within the same LAN component; only processes holding TCP endpoints would perform WAN gossip.

Generalizing, we can ask about the behavior of pbcast if each participant only knows about, and gossips to, a subset of the other participants – perhaps, in a network of 10,000 processes, each participant might gossip within a set of 100 others. Research on randomized networks has demonstrated that randomized protocols operate correctly on randomly generated graphs with much less than full connectivity [FPRU90]. Drawing upon this theory, we can conclude that pbcast should retain its properties when such a subset scheme is employed. Moreover, the subset to which a given member gossips can be picked to minimize latency, thereby bounding the round-trip times and hence round lengths to a reasonable level. We are currently developing a membership service for our Spinglass implementation of pbcast, which will manage membership on behalf of the system, select these subsets, and inform each pbcast participant of the list of processes to which it should gossip.

Extended in this manner, the protocol overcomes the scalability concerns just identified, leaving us with a protocol having entirely *local* costs, much like SRM, RMTP, XTP or

MUSE. A pbcast message can be visualized as a sort of frontier advancing through the network over a spanning tree. Each process first learns of the pbcast either during the initial multicast, or in rounds of gossip that occur over low-latency links between processes and their neighbors. Irrespective of the size of the network, safety and stability would rapidly be reached. Moreover, only membership service needs the full membership of the multicast group. Typical pbcast participants would know only of the processes to which they gossip, would gossip mostly to neighbors, and the list of gossip destinations would be updated only when that set changes, not on each membership change of the overall group.

Optimization 7: Multicast for some retransmissions

In certain situations, our protocol employs multicast to retransmit a message, although we do this rather carefully to avoid triggering the sort of unscalable growth in overhead seen in the SRM protocol. At present, our protocol uses multicast if the same process is solicited twice to retransmit the same message: the probability of this happening is very low unless a large number of processes have dropped the message. Additionally, suppose that we define distance in terms of common IP address prefixes: processes in the same subnet are “close” to one-another, and processes on different subnetworks are “remote”. Then when a process solicits a (point-to-point) retransmission from a process remote from it, upon receiving that message it immediately re-multicasts it using a “regional” setting for the multicast TTL field. The idea is that since optimization 6 ensures that most gossip will be between processes close to one-another, it is unlikely that a retransmission would be needed from a remote source unless the message in question was dropped within the region of the soliciting process. Accordingly, the best strategy is to re-multicast that message immediately upon receipt, within the receiver’s region.

5. Integration with Ensemble’s Flow Control and State Transfer Tools

Flow Control

Our model implicitly requires that the rate of pbcast messages be limited. Should this rate be exceeded, the network load would threaten the independent failure and latency assumptions of the model, and the guarantees of the protocol would start to degrade. In normal use, some form of application-level rate control is needed to limit the rate of multicasts. For example, the application might simply be designed to produce multicasts at a constant, predetermined rate, calculated to ensure that the risk of overloading the network is acceptably low.

Pbcast can also be combined with a form of flow control tied to the number of buffered messages active within the protocol itself. In this approach, when a sender presents a new multicast to the communication subsystem, the message would be delayed if the subsystem is currently buffering more than some threshold level of active multicasts, from the same or other sources. As pbcast messages age and are garbage collected, new multicasts would be admitted. Ensemble, the multicast framework within which we implemented one of our two versions of pbcast, supports a flow control mechanism that works in this manner. However, for the experiments reported here, we employed application-level rate limitations. We believe that for the class of applications most likely to benefit from a bimodal reliable multicast, the rate of data generation will be predictable, and used to parameterize the protocol. In such cases, the addition of an unpredictable internal flow-control mechanism would reduce the determinism of the protocol, while bringing no real benefits.

Recovery From Delivery Failures

Recall from Figure 1 that in a conventional form of reliable group communication, a single lagging process can impact throughput and latencies throughout the entire group. Our protocol overcomes this phenomenon but suffers from a complementary problem, which is that if a process lags far enough behind the other group members, those processes may garbage collect their message histories, effectively partitioning the slow process away from the remainder of the group. The slow process will detect this condition when normal communication is restored, but has no opportunity to catch up within the basic protocol. Notice that this problem is experienced by a faulty process, not a healthy one, and hence can't be addressed simply by adjusting protocol parameters.

We see two responses to this problem. In Spinglass, we are exploring the possibility of varying the amount of buffering used by each pbcast participant. Most processes would have small buffers, but some might have very large buffers – in the limit, some could spool copies of all messages sent in the system. These would then serve as repositories of last resort, from which a recovering process could request the entire sequence of messages which were lost during a transient outage.

When pbcast is used together with Ensemble, a second option arises. Ensemble includes tools for process join and leave, membership tracking, and traditional reliable multicast within a process group. Included with these is a state transfer feature. The mechanism permits a process joining a process group to receive state from any process or set of processes already present in the group. Such a joining process can also offer its own state to the existing members, hence the protocol supports *state merge*, although in normal usage we prefer to transfer state from a "primary component" of the partitioned group to a "minority component". A pbcast participant that falls behind could thus use the Ensemble state transfer as a recovery mechanism.

6. Graphing the Computational Results

In Appendix A, we show how pbcast can be analyzed under the assumptions of our model. This analysis yields a computational model for the protocol, which we used to generate the graphs in Figure 4. These graphs were produced under the assumption that the initial unreliable multicast failed (only the original sender initially has a copy), that the probability of message loss is 5% and the probability that a process will experience a crash failure during a run of the protocol is 0.1% All of these assumptions are very conservative, hence these graphs are quite conservative. Recall from Section 4 that the fanout measures the number of processes to which the holder of a multicast will gossip before garbage collecting the message.

On the upper left is a graph illustrating pbcast's bimodal delivery distribution, which

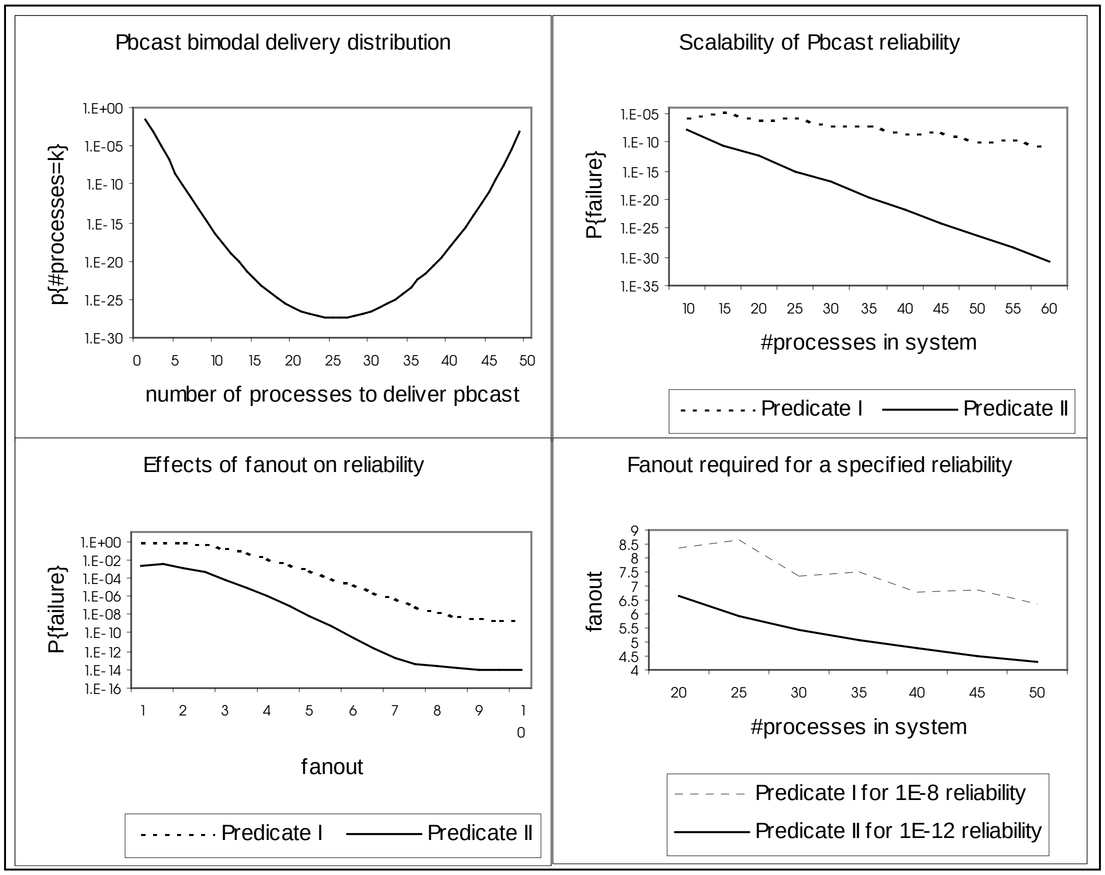


Figure 4: Analytical results (unless otherwise indicated, for 50 processes)

motivates the title of this paper. As the general of our little fable recognized, the likelihood that a very small number of processes will receive a multicast is quite low. The likelihood that almost all receive the multicast is very high. And the intermediary outcomes are of vanishingly low probability.

To understand this graph, notice that the Y axis is expressed as a predicate over the state of the system after some amount of time. Intuitively, the graph imagines that we run the protocol for a period of time and then look on the state S achieved by the protocol. Pbcast guarantees that the probability that S is such that almost none or almost all process have delivered pbcast is high, and the probability that S is such that about a half of participants have delivered pbcast is extremely low. When one considers that the Y axis is on a logarithmic scale, it becomes clear that pbcast is overwhelmingly likely to deliver to almost all processes if the sender remains healthy and connected to the network.

When the initial unreliable multicast is successful, the situation is quite different; so much so that we did not produce a graph for this case. Suppose that 5% of these initial messages are not delivered. The initial state will now be one in which 47 processes are already infected, and if our protocol runs for even a single round, it becomes overwhelmingly probable that the pbcast will reach all 50 processes, limited only by process failures. For this section the worst-case outcomes are more relevant, hence it makes sense to assume that the initial unreliable multicast fails.

The remaining graphs superimpose the behavior of pbcast with respect to two predicates that define exemplary undesired outcomes; in keeping with the idea that these show risk of failure, we produced them under the pessimistic assumption that the initial IP multicast fails. The first predicate is the one our general might have used: it considers a run of the protocol to be a failure (in his case, a gross understatement!) if the multicast reaches more than 10% of the processes in the system but less than 90%. The second predicate is one that arises when pbcast is used in a system that replicates data in a manner having properties similar to those of virtual synchrony, using an algorithm we describe elsewhere [HB96]. In this protocol, updates have a 2-phase behavior, which is implemented using pbcast. An undesired outcome arises if pbcast delivers to roughly half the processes in the system, and crash failures make it impossible to determine whether or not a majority were reached, forcing the update to abort (roll-back) and be restarted. The idea of these three graphs is to employ our model to explore the likelihood that pbcast could be used successfully in applications with these sorts of definitions of failure. Both predicates are formalized in the appendix.

The applications of pbcast discussed in the introduction could also be reduced to failure predicates. For example, in an air-traffic application that uses pbcast to replicate updates to the tracks associated with current flights, the system would typically operate safely unless several updates in a row were lost for the same track. By starting with the controller's ability to tolerate missing track updates or inconsistency between the data displayed on different consoles, one can compute a predicate encoding the resulting risk threshold as a predicate. At the level of our model, each pbcast is treated as an independent event, hence any condition expressed over a run of multicasts can be re-expressed as a condition on an individual outcome.

The graph on the upper right shows how the risk of a "failed" pbcast drops with the size of the system. The lower graphs look at the relation between the expected "fanout" from each participant during the gossip stage of failure and the risk that the run will fail in the sense defined earlier. The graphs were based on the parameters used in our experimental work, and can be used in setting parameters such as the fanout and the number of rounds, so that pbcast will achieve a desired reliability level, or to explore the likely behavior of pbcast with a particular parameterization in a setting of interest. Notice also that predicate I yields a much lower reliability than predicate II. This should not surprise us: predicate I counts a pbcast as faulty if, for example, 30% of the processes in the system fail to receive it. Predicate II would treat such an outcome as a success, since 70% is still a clear majority. Note also that the graph on the lower right compares the fanout required to obtain 1E-8 reliability using predicate I with that required to obtain 1E-12 for predicate II. This was to get the curves onto the same scale; in practice, 1E-8 is probably adequate for the applications discussed later. If the IP multicast were successful, the risks of failure in all three graphs would be reduced by several orders of magnitude.

Since throughput stability lies at the heart of our work, we also set out to analyze the expected variance in throughput rates using formal methods. We first considered the expected situation for a single pbcast where the initial unreliable multicast fails. Our approach was to use the analysis to obtain a series of predictions showing how the number of processes which have yet to receive a copy decreases over time (Figure 5), and

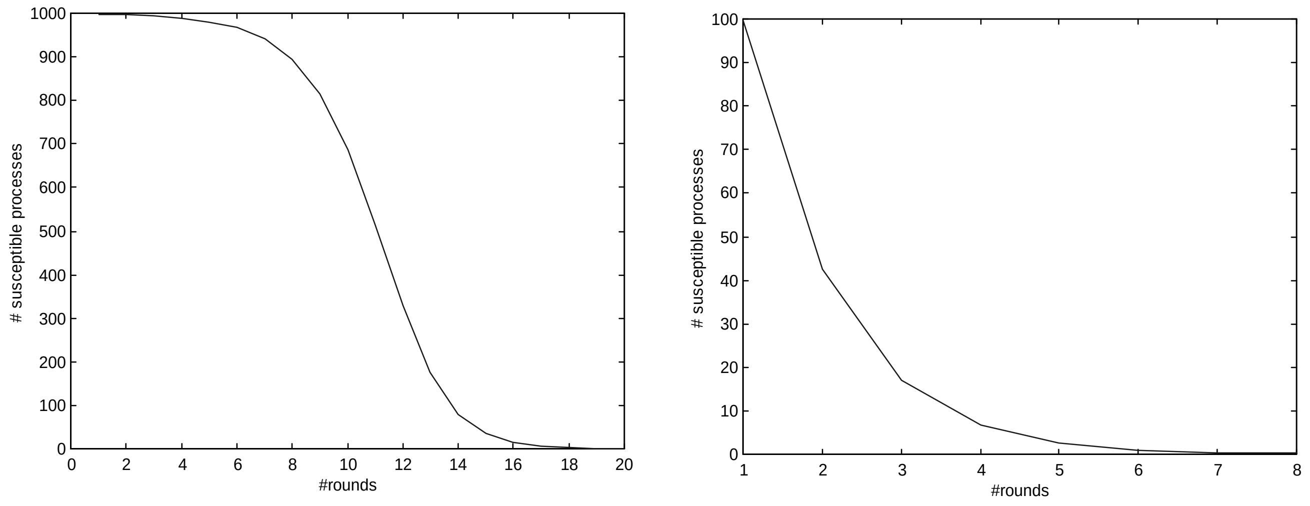


Figure 5: Number of susceptible processes versus number of gossip rounds when the initial multicast fails (left) and when it reaches 90% of processes (right; note scale). Both runs assume 1000 processes.

then to use this data to compute the expected number of rounds before a selected correct participant receives a multicast (Figure 6). The resulting curves peak at roughly the log of the group size, and have variance that also grows with the log of the group size.

Next, we considered the situation when the initial IP multicast or tree-based multicast is successful. In this case, a typical process receives a multicast after it has been "relayed" through some number of intermediary processes, and the length of the relay chain will grow with $log_b(N)$, where the base b is the average branching factor of the forwarding tree used by the initial multicast – 2 in the case of the tree-based scheme used in our experimental work, but potentially much larger for IP multicast. Suppose that this chain is of length c. Each of the relaying processes can be viewed as an independent "filter" that delays messages by some mean amount with some associated variance. If we treat this as a normal distribution, the transit through the entire tree will also have a normal distribution, with its mean equal to the c times the mean forwarding delay, and variance equal to $\sqrt{c}*\sigma$, where σ is the variance of the forwarding delay distribution[4].

From this information, we can make predictions about the average throughput and the variance in throughput that might be observed over various time scales. Consider a period of time during which a series pbcast messages are injected into the system and assume that the messages are independent of one-another (that is, that the rate is sufficiently low so that there are no interference effects to worry about). Based on the analysis above, the expected variance in time to receive the sequence will be $\sqrt{(2c)}*\sigma$. Thus we would expect the throughput variance to grow slowly, as the square-root of the log of the system size.

[4] This is because the distributions are identical normal ones with variance σ, and variance for a summed distribution grows as the square-root of the sum of squares.

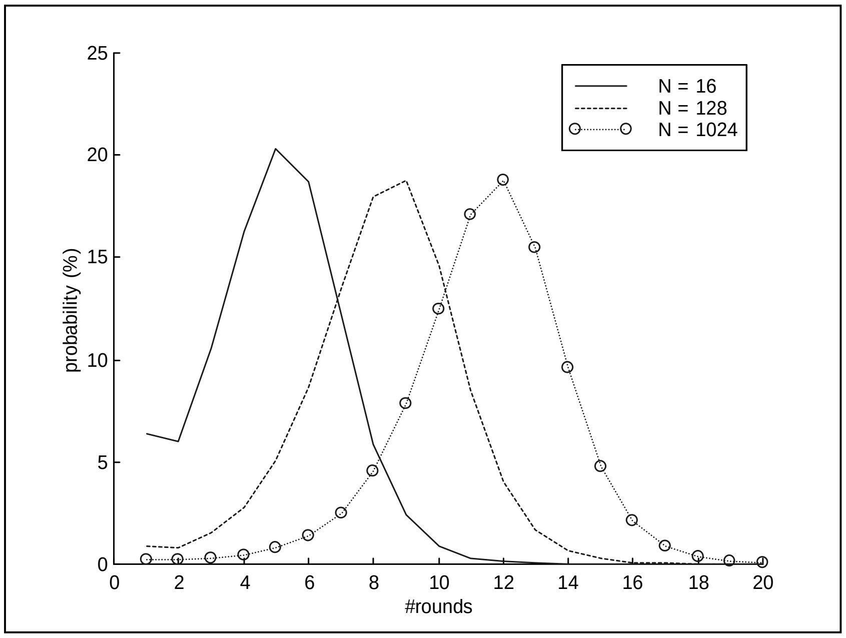


Figure 6: The probability for a correct process to receive a pbcast in a particular round for groups of various sizes. The distributions are essentially normal, with means centered at log(N)

But now we face a problem: the two cases have very different expected delivery latency and variance. If the initial multicast has erratic average reliability, throughput will fluctuate between the two modes.

Our experimental data, reported in Section 7, is for a setting in which the initial tree-based multicast is quite reliable, and we indeed observe very stable throughput with variance that grows slowly and in line with predictions. But suppose that we were to use pbcast in dedicated Internet settings or even over the public Internet. In these cases, the unreliable multicast might actually fail some significant percentage of the time. Compensating for this is optimization 7, which was not treated in our theoretical analysis, but in practice would cause processes to re-multicast messages rapidly in such a situation. Experimentally, optimization 7 does seem to sharpen the delivery distribution dramatically[5].

Under the assumption that the delivery distribution will be reasonably tight, but still have two modes, one option would be to introduce a buffering delay to "hide" the resulting variations in throughput, using the experimental and analytic results to predict the amount of buffering needed. For example, suppose that we assume clock synchronization and that we delay each multicast, delivering it *k* times the typical round-length after the time at which it was sent. Here, *k* could be derived from Figure 6 so that 99% of all messages will have been received in this amount of time – 14 to 16 gossip rounds in the case of N=128, for example – about 1.5 seconds if rounds last for 100ms. At the cost of buffering the delayed messages for this amount of time, we could now smooth almost all variance in throughput rate. Such a method lies at the core of the well-known Δ-T atomic multicast [CASD85]. A version of pbcast that incorporates such a delay (although for a different reason) is discussed in [HB96, Birman97].

[5] One way to visualize this is to think of the networks as the surface of a soccer ball, having regions with high connectivity (surface patches) connected by tunnels (borders between the patches). Pbcast operates like a firework, bursting through the network to infect each region, then bursting again regionally to infect most processes in each patch, and finally fading away with a few last sparks as the remaining processes are infected by unicast gossip.

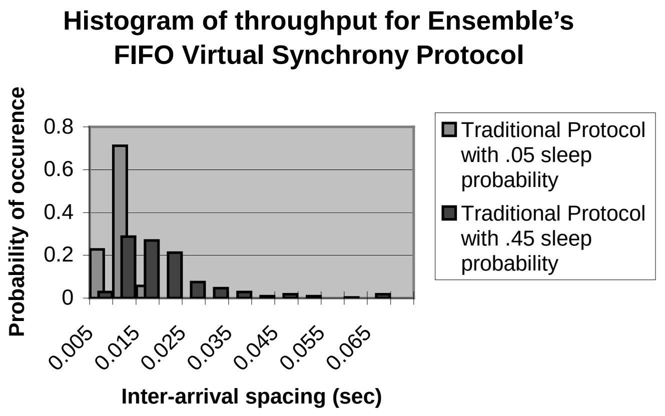
But worst-case delay may exaggerate the actual need for buffering. Our experimental work, which reflects the impact of optimization 7, suggests that even a very small amount of buffering could have a dramatic impact. This is in contrast to the situation in [CASD85], where a deterministic worst-case analysis leads to the somewhat pessimistic conclusion that very substantial amounts of buffering may be needed, and very long delays before delivery. In our setting, the goals are probabilistic and the analysis can focus on the expected situation, not the worst case.

To summarize, formal analysis gives us powerful tools and significant predictive options. The tools permit pbcast to be parameterized for a particular setting, and show us how to bridge the gap between the pbcast primitive itself and the application-level reliability objectives. The predictions concern the distribution of expected outcomes for the protocol and also the degree to which throughput will have the stability properties we seek. In this second respect, we find that pbcast, used in settings where the initial unreliable multicast is likely to be successful (reaching most destinations most of the time) would indeed exhibit stable and steady throughput in a scalable manner. Confirming this, our experiments with Spinglass, reported in the next section, show that even with very small buffers and without any artificial delay at all, the received data rate remains steady when we alternate between a mode in which most multicast are successful, and one in which multicasts reach very few destinations.

7. Performance and scalability of an implementation

In this section, we present experimental results concerning the performance, throughput stability, and scalability of pbcast using runs of the actual protocol. We include several types of experimental work. We start with a study of the Ensemble virtual synchrony protocols, which we run side-by-side with the Ensemble implementation of pbcast. The Ensemble protocols we selected perform extremely well and have been heavily tuned, hence we believe it fair to characterize them as “typical” of protocols in this class. Obviously, however, one must use care in extrapolating these results to other implementations of virtually synchronous multicast. The experiments reported here were conducted using an SP2 parallel computer, which we treated as a network of workstations. The idea was to start by isolating our software (the real software, which can run without changes on a normal Internet LAN or WAN) on a very clean network and then to inject noise.

Next, we compare pbcast with SRM, using the NS-2 simulator, and the two SRM implementations available for that environment. We used NS-2 to construct a simulation of pbcast, and then examined pbcast and SRM side-by-side under various network topologies and conditions, using the SRM parameter settings recommended by the designers of that protocol. The simulation allowed us to scale both protocols into very large networks.



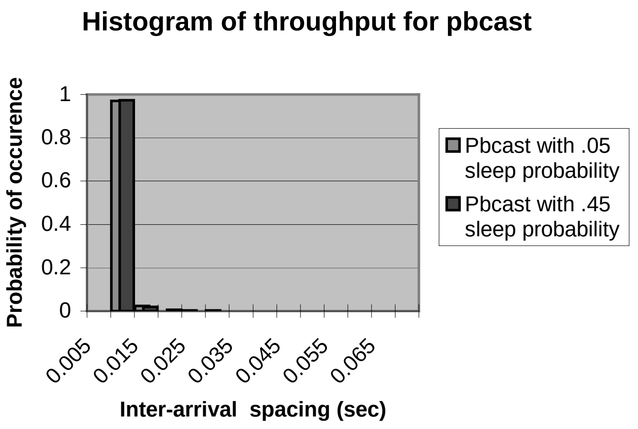
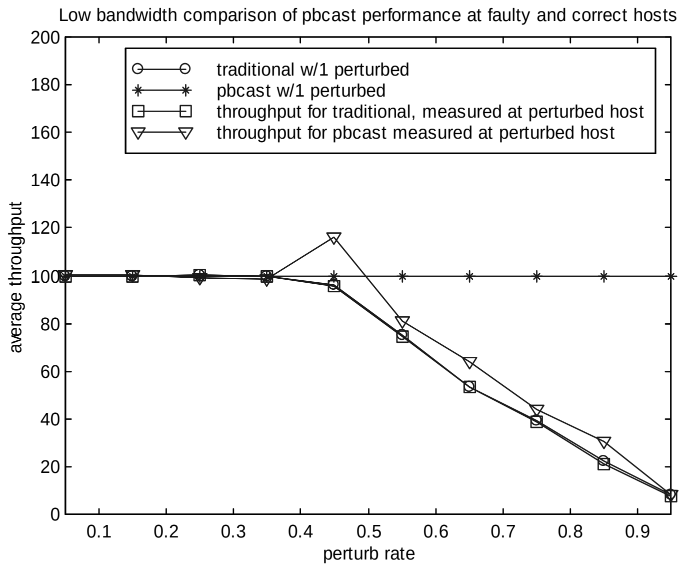


Figure 7: Inter-arrival message spacing histograms for a virtually synchronous protocol at 75 msgs/sec under perturbation (left), and for the Bimodal Multicast protocol (right). The steady delivery characteristics of pbcast are evident.

Finally, we looked at the Spinglass implementation of pbcast on a network of workstations running over a 10-Mbit ethernet in a setting where hardware multicast is available. This last study was less ambitious, because the number of machines available to us was small, but still provides evidence that what we see in simulation and on the SP2 actually does predict behavior of the protocol in more realistic networks.

Accordingly, we start by looking at pbcast next to virtual synchrony on network configurations of various sizes, running on the SP2. We emulate network load by randomly dropping or delaying packets, and emulate ill-behaved applications and overloaded computers by forcing participating processes to sleep with varied probabilities. With this approach we studied the behavior of groups containing as many as 128 processes.

Figure 7 shows the interarrival message spacing for a traditional virtual synchrony



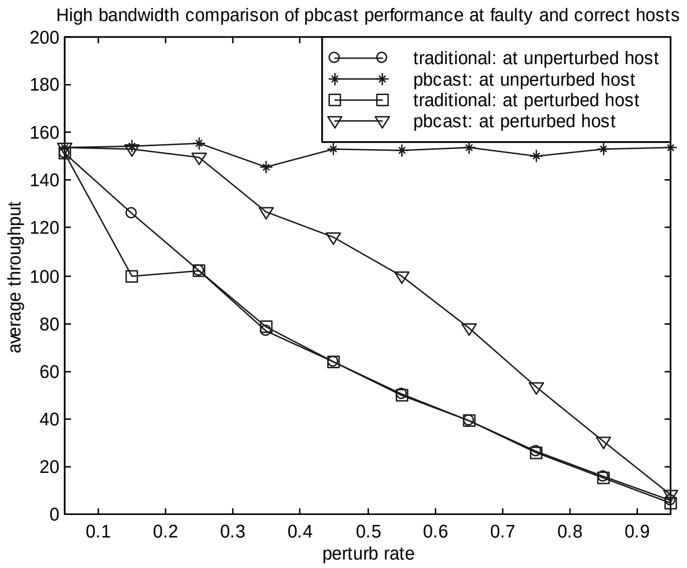


Figure 8: Ensemble ("traditional") and pbcast, side-by-side, in an experiment similar to the one used to produce Figure 1. For a group of 8 processes, we perturbed one and looked at the delivery rate at a healthy group member and at the perturbed process, at 100 messages/second and 150/second. With the virtual synchrony protocol, data rates to the healthy and perturbed process are identical. With pbcast, the perturbed process starts to drop messages (signified by a lower data rate than the injection rate) but the healthy processes are not affected.

protocol, running in Ensemble[6], side by side with the Ensemble implementation of pbcast. These were produced in groups of 8 processes in which one process was perturbed by forcing it to sleep during 100ms intervals with the probability shown. The data rate was 75 7k-byte multicasts per second: relatively light for the Ensemble protocol (which can reach some 300 multicasts per second in this configuration), and also well below the limit for pbcast (about 250 per second). Both graphs were produced on the SP2.

The inter-arrival times for the traditional Ensemble protocols spread with even modest perturbation, reflecting bursty delivery. Pbcast maintains steady throughput even at high perturbation rates.

The first figure in this paper, Figure 1, illustrated the same problem at various scales. In the experiment used to produce that figure, we measured the throughput in traditional Ensemble virtual synchrony groups of various sizes as we perturbed a single member. We see clearly that although Ensemble can sustain very high rates of throughput (200 7k-byte messages/second is close to the limit for the SP2 used in this manner, since the machine lacks hardware multicast), as the group becomes larger it also becomes more and more sensitive to perturbation. In Figure 8, we examined the same phenomenon in more detail for a small group of 8 processes. Interestingly, even the perturbed process receives a higher rate of messages with pbcast.

Figures 1, 7 and 8 are not intended as an attack upon virtual synchrony, since the model has typically been used in smaller groups of computers, and with applications that generate data in a more bursty manner, rarely maintaining a sustained, high data rates [Birman99]. Under these less extreme conditions, the protocols work well and are very stable. The problems cited here arise from a combination of factors: large scale, high and sustained data rates, and a type of perturbation designed to disrupt throughput without triggering the failure detector.

Figure 9 was derived from the same experiment using pbcast; now, because throughput was so steady, we included error bars. These show that as we scale a process group, throughput can be maintained even if we perturb members, but the variance (computed over 500ms intervals) grows. On the bottom right is a graph of pbcast throughput variance as a function of group size. Although the scale of our experiments was inadequate to test the log-growth predictions of Section 6, the data at least seems consistent with those predictions.

In Figure 10 we looked at the consequences of injecting noise into the system. Here, system-wide packet loss rates were emulated by causing the SP2 to randomly drop the designated percentage of messages. As the packet loss rate grows to exceed 10% of all packets pbcast becomes lossy at the highest message rate we tested (200 per second); the protocol remains reliable even at a 20% packet loss rate when we run it at only 100 messages per second. Increasing the fanout did not help at the high packet injection rate, apparently because we are close to the bandwidth limits of the SP2 interconnect.

[6] Although Ensemble supports a scalable protocol stack, for our experiments that stack was found to behave identically to the normal virtual synchrony stack. Accordingly, the data reproduced here is for a normal Ensemble stack, providing fifo ordering and virtual synchrony.

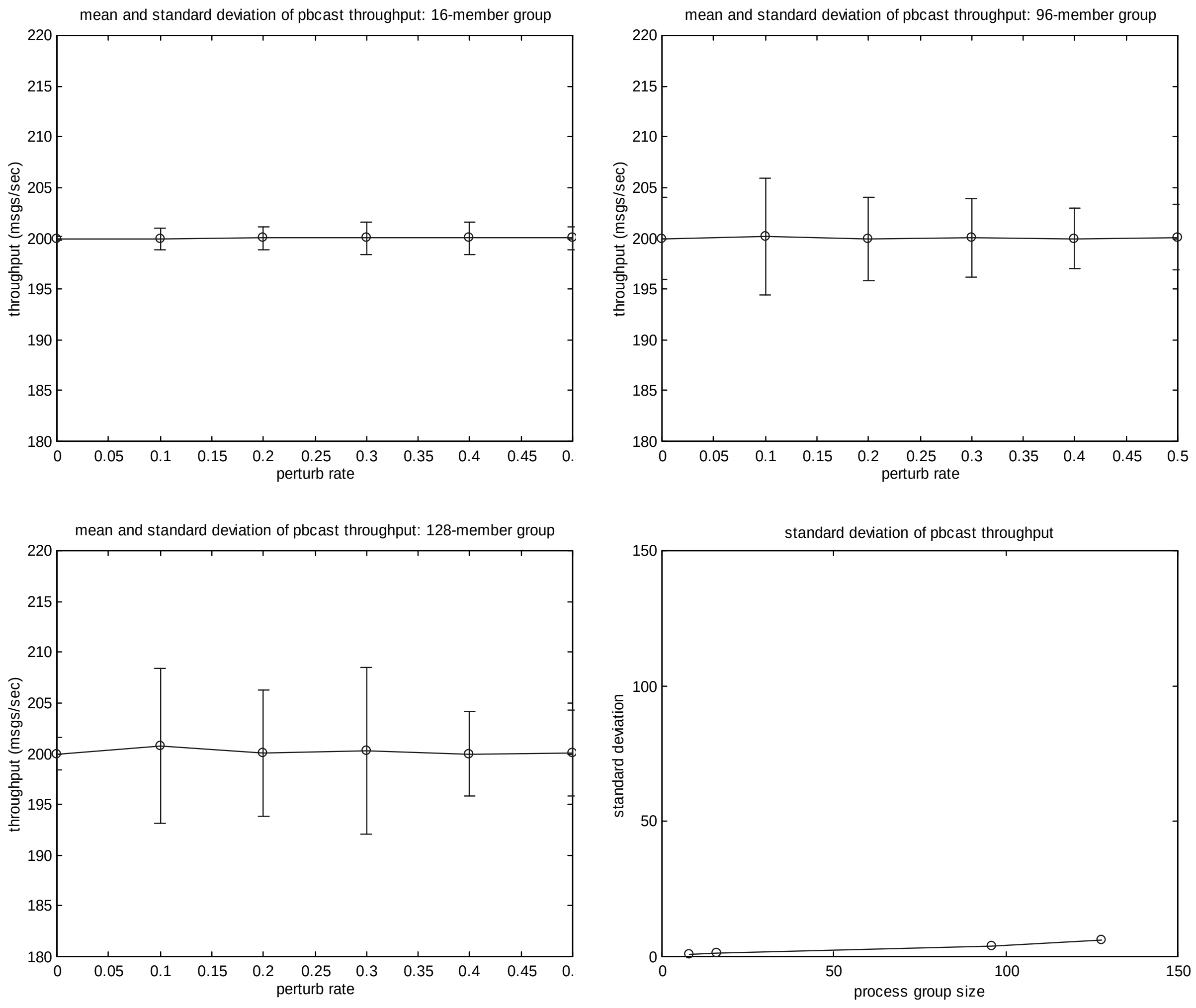


Figure 9: Pbcast throughput is extremely stable under the same conditions that provoke degraded throughput for traditional Ensemble protocols, but variance does grow as a function of group size. For these experiments 25% of group members were perturbed and throughput was instrumented 100 messages at a time. Behavior remains the same as the perturbation rate is increased to 1.0, but for clarity of the graphs, we show only the interval [0, .5]. Although our experiments have no scaled up sufficiently to permit very general conclusions to be drawn, the variance in throughput is clearly small compared to the throughput rate, and is growing slowly.

Figure 10 thus illustrates the dark side of our protocol. As we see here, with a mixture of high data bandwidths and high loss rates, pbcast is quite capable of reporting gaps to healthy processes. This can be understood as a "feature" of the protocol; presumably, if we computed the bimodal curve for this case, it would be considerably less "sharp" than Figure 4 suggests. In general, if the gossip fanout is held fixed, the expected reliability of pbcast drops as the network data loss rate rises or if the network becomes saturated. In the same situation, a virtual synchrony protocol would refuse to accept new multicasts.

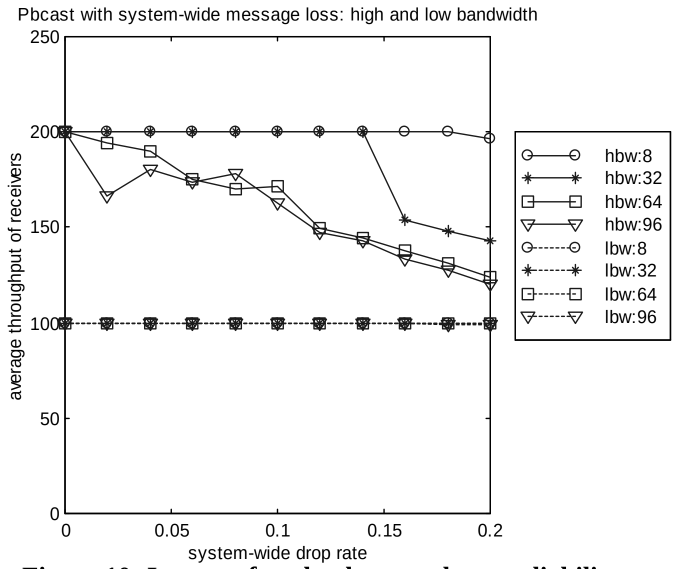


Figure 10: Impact of packet loss on pbcast reliability. At high data rates (200 msgs/sec) noise triggers packet loss in large groups; at lower rates even significant noise can be tolerated.

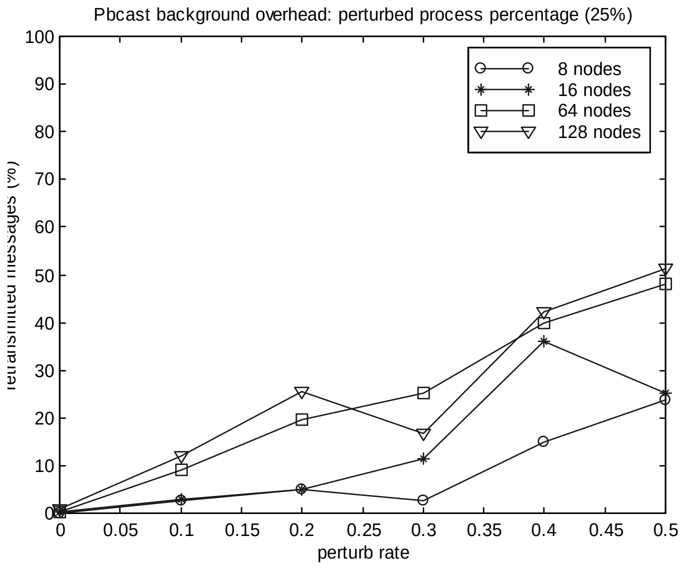


Figure 11: The number of retransmission solicitations received by a healthy process as a function of group size and perturbation rate.

With this one exception, our experiments provoked no data loss at all for healthy pbcast receivers.

Figure 11 shows the background overhead associated with these sorts of tests. Here we see that as the pertubation rate rises, the overhead also rises: for example, in a 16-member group with 25% of processes perturbed 25% of the time, 8% of messages must be retransmitted by a typical participant; this rises to 22% in a 128-member group. Although our analysis shows that overhead is bounded, it would seem that the tests undertaken here did not push to the limits until the perturbation rate was very high.

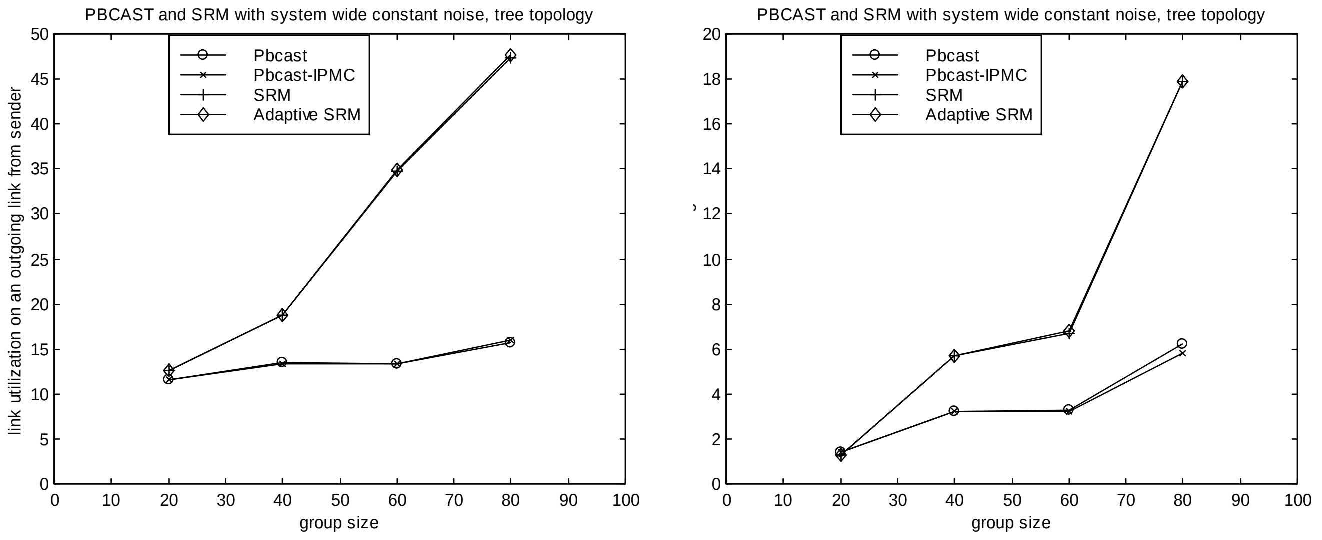


Figure 12: SRM and pbcast. Here we compare link-utilization levels; small numbers are better. With constant noise, both protocols exhibit some growth in overheads but the scalability of pbcast is better. Here, each link had an independent packet-loss probability of 0.1%.

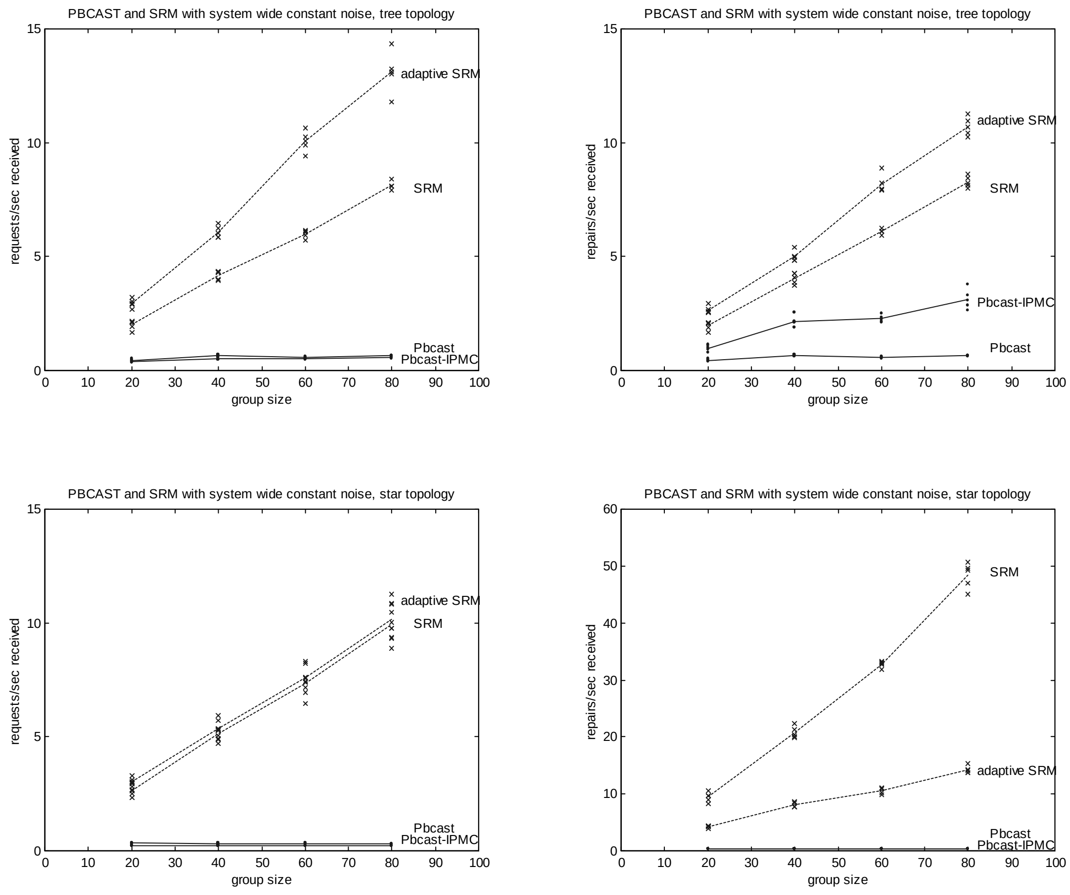


Figure 13: Comparison of the rate of overhead messages received, per second, by typical members of a process group when using SRM and pbcast to send 100 msgs/sec with 0.1% message-loss rate on each link. Here, we look at two topologies: the same balanced 4-level tree as in Figure 12, and a star topology. In some situations, SRM can be provoked into sending multiple retransmissions (repair messages) for each request; pbcast generates lower overheads.

Next, we turned to a simulation, performed using NS-2. In the interest of brevity, we include just a small amount of the data we obtained through simulation; elsewhere, we discuss our simulation findings in much more detail [OXB99]. Figure 12 shows data collected using a network structured as a 4-level balanced tree, within which we studied link utilization for pbcast with and without optimization 7 (we treated this separately because our theoretical results did not consider this optimization), and for the two versions of SRM available to us – the adaptive and non-adaptive protocol, with parameters configured as recommended by the developers.

What we see here is that as the group grows larger (experiencing a large number of dropped packets, since all links are lossy), both protocols place growing load on the network links. Much of the pbcast traffic consists of unicasts (gossip and retransmissions), while the SRM costs are all associated with multicast, and rise faster than those for pbcast.

Figure 13 looks specifically at overheads associated with the two protocols, measuring the rate of retransmission requests and copies of messages received by a typical participant when 100 msgs/sec are transmitted in a network with 0.1% packet loss on each link. The findings are similar: SRM overheads grow much more rapidly than pbcast overheads as we scale the system up, at least for this data loss pattern. Readers interested in more detail are referred to [OXB99].

Finally, we include some preliminary data from the Spinglass implementation of pbcast, which runs on local area networks. These graphs, shown in Figure 14, explore the impact of optimizations 2 and 7. Recall that optimization 2 introduced a limit on the amount of data pbcast will retransmit in any single round, while optimization 7 involves the selective use of multicast when retransmitting data. To create these graphs, we configured a group of 35 processes with a single sender, transmitting 100 1-k byte messages per second on a 10Mbit LAN. For the top two graphs, every 20 seconds, we triggered a burst of packet loss by arranging that 30% of the processes will simultaneously discard 50 consecutive messages; we then graphed the impact on throughput at a healthy process. Recall that our throughput analysis has trouble with such cases, because they impact the overall throughput curve of the protocol by increasing the expected mean latency.

What we see here is that without optimization 2 (top left), the perturbation causes a big fluctuation in throughput. But with the optimization, in this case limiting each process to retransmit a maximum of 10K bytes per gossip round, throughput is fairly steady even when packet loss occurs. Obviously, this optimization reflects a tradeoff, since a perturbed process will have more trouble catching up, but the benefit for system throughput stability is much more stable.

The lower graphs examine the case where an outage causes the initial multicast to fail, along the lines of the conservative analysis presented in Section 6 of this paper. The emulation operates by intercepting 10 consecutive multicasts and, in each case, allowing the message to reach just a single randomly selected destination (hence, two processes are initially infected; we comment, however, that the graphs look almost identical if the initial multicast is entirely discarded, so that it initially infects only the sender). Here, even with optimization 2, throughput is dramatically impacted each time the outage occurs. With optimization 7, however, Spinglass re-multicasts the affected messages, and throughput is again very smooth. We monitored memory used during these tests; in no case was more than 256KB of buffering required. Since this smooth delivery was obtained even without delaying received messages (except to put them in fifo order), the data supports our contention that at most a very brief delay is needed to ensure an extremely smooth delivery rate when optimization 7 is in use.

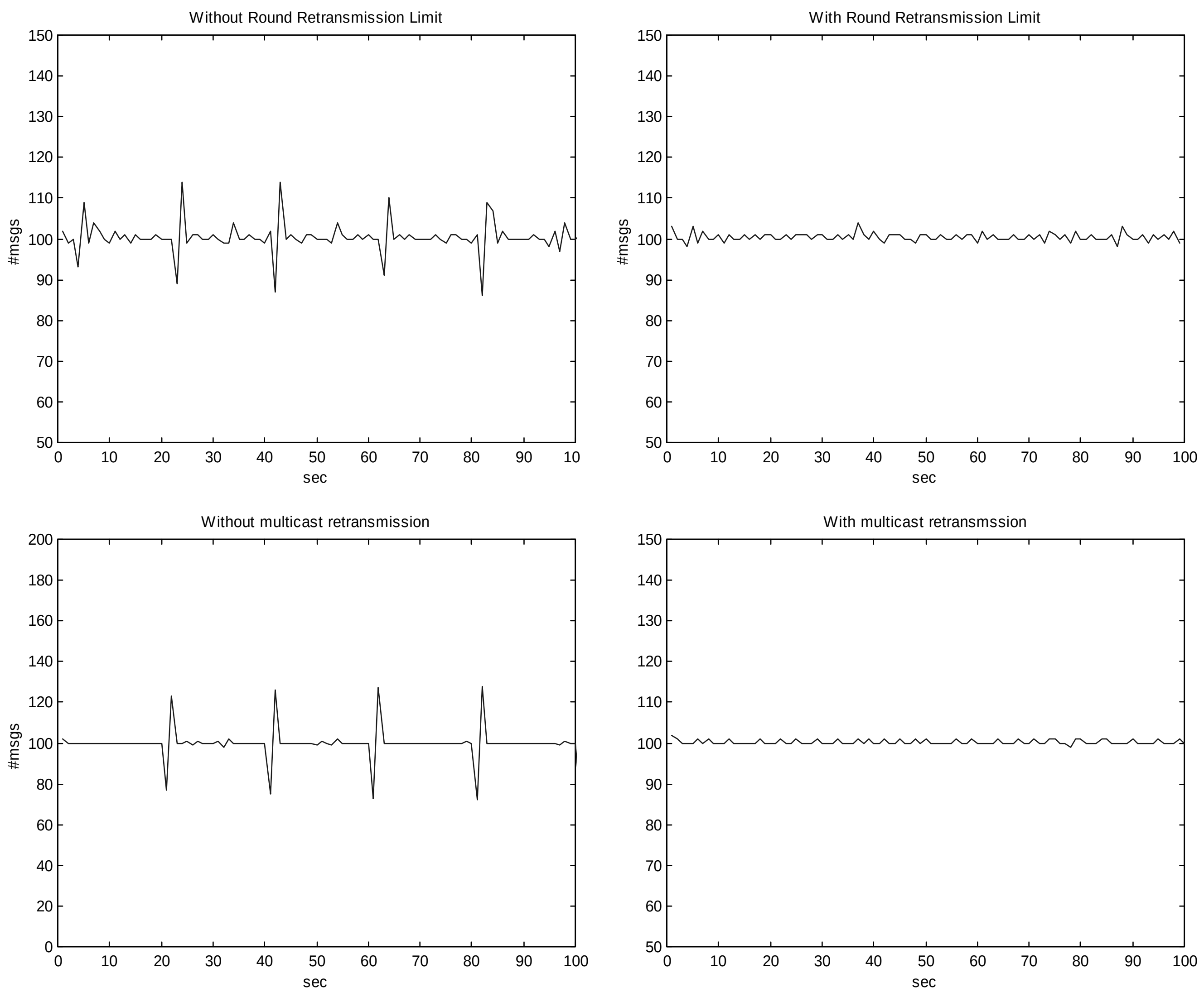


Figure 14 : Representative data for Spinglass, used in a group of 35 members on a 10Mbit local area network.

8. Programming with probabilistic communication tools

Although a probabilistic protocol can be like other types of reliable group communication and multicast tools, the weaker nature of the guarantees provided has important application-level implications. For example, Ensemble's virtually synchronous multicast protocols *guarantee* that all non-faulty members of a process group will receive any multicast sent to that group, even if this requires delaying the entire group while waiting for a balky process to catch up. In contrast, probabilistic protocols could violate traditional atomicity guarantees. The *likelihood* of such an event is known to be low if the network is behaving itself and the soft-failure limits are appropriate ones, but a transient problem could certainly trigger these sorts of problems.

These considerations mean that if data is replicated using our protocol, the application should be one that is insensitive to small inconsistencies, such as the following:

- Applications that send media, such as radio, TV, or teleconferencing data over the internet. The quality predictions provided by pbcast can be used to tune the application for a minimal rate of dropouts.
- In a stock market or equity-trading environment [PC97], actively traded securities are quoted repeatedly. The infrequent loss of a quote would not normally pose a problem as long as the events are rare enough and randomly distributed over messages generated within the system – a property pbcast can guarantee.
- In an air-traffic control setting [PHIDEAS], many forms of data (such as the periodic updates to radar images and flight tracks[7]) age rapidly and are repeatedly refreshed. Dropping updates of these sorts infrequently would not create a safety threat. It is appealing to use a scalable reliable protocol in this setting, yet one needs to demonstrate to the customer that doing so does not compromise safety. The ability to selectively send the more time-critical but less safety-critical information down a probabilistic protocol stack that guarantees stable throughput and latency would be very desirable.

  In this setting, there are also problems for which stronger guarantees of a virtually synchronous nature are needed. For example, the Phideas system replicates flight plan updates within small clusters of 3-5 workstations, using state machine replication. Each event relevant to the shared state is reliably multicast to the cluster participants, which superimpose a terse representation of the flight plan on a background of radar image and track data. The rate of updates to the foreground data – the flight tracks – may be as low as one or two events per second. A virtually synchronous multicast is far more appropriate for this second class of uses.
- In a health-care setting, many forms of patient telemetry are refreshed frequently. Data of this sort can be transmitted using pbcast because if a reading is lost, a new reading will almost certainly be received soon afterwards. On the other hand, it would be important to use a stronger form of reliable multicast if a critical data item, such as a medication change order, is to be replicated at multiple sites in the hospital system. Here, the application cannot tolerate even a very small risk that the multicast will "fail", because the multicast is only sent once and has critical importance within the system. Such a problem represents a good match with protocols having end-to-end guarantees. The doctor's computer, at one end of the dosage-changing operation, needs the guarantee that the various systems displaying medication orders at the other end will reflect the changed dosage. (Otherwise the doctor should be warned).

Each of these examples is best viewed as a superposition of two (or more) uses of process groups. Notice, however, that the different uses are independent – virtual synchrony is not used to overcome the limitations of pbcast. Rather, a pbcast-based application (such as the one used to update the radar images on the background of a controllers screen) co-exists with a virtually synchronous one (the application used to keep track of flight plans

[7] A flight *track* plots the observed position and trajectory of a flight, as measured by radar and telemetry. The flight *plan* is a record of the pilot's intentions and the instructions given by the controller.

and instructions which the controller has issued to each flight). The application decomposes cleanly into two applications, one of which is solved with pbcast, and the other with the traditional form of reliable multicast.

Traditional forms of reliable multicast can and should be used where individual data items have critical significance for the correctness of the application. Examples include security keys employed for access to a stock exchange system, flight plan data replicated at the databases associated with multiple air-traffic control centers, or the medication dosage instructions from the health-care example. But as just seen, other kinds of data may be well matched to the pbcast properties. Interestingly, in the above examples, *frequent* message traffic would often have the properties needed for pbcast to be used safely, while *infrequent* traffic would be typical for objects such as medical records, which are updated rarely, by hand. Thus pbcast would relieve the more reliable protocols of the sort of load that they have problems sustaining in a steady manner. These examples are representative of a class of systems with mixed reliability requirements.

A second way to program with pbcast is to develop algorithms that make explicit use of the probabilistic reliability distribution of the protocol, as was done in the data replication algorithm mentioned in Section 6 [HB96]. One can imagine algorithms that use pbcast to replicate data with probabilistic reliability, and then employ decision-theoretic methods to overcome this uncertainty when basing decisions on the replicated data. For example, suppose that pbcast was used to replicate all forms of air-traffic control data – an idea which might be worth pursuing, since the protocols are lightweight, easy to analyze, and very predictable. The quality of each flight plan will now be probabilistic. Under what conditions is it safe to make a safety-critical decision, and under what conditions should we gossip longer (to sharpen the quality of the decision)? In the future, we hope to explore these issues more fully.

9. Comparison with Prior Work

As noted earlier, our protocol builds upon a considerable body of prior work. Among this, the strongest connections are to the epidemic algorithms studied by Demers *et. al.* in the Xerox work on data replication. However, the Xerox work looked at systems under light load, and did not develop the idea of probabilistic reliability as a property one might present to the application developer. Our work extends the Xerox work by considering runs of the protocol, and by using IP-multicast. In addition to the the work reported here, our group at Cornell also explored other uses of gossip, such as gossip-based membership tracking [RHM98] and gossip-based stability detection [Guo98].

Our protocol can also be seen as a "soft" real-time protocol, with connections to such work as the Δ-T protocol developed by Cristian [CASD85], and Baldoni and Raynal's δ-causal protocol [BMR96, BPRS96]. None of this prior work investigated the issue of steady load and steady data delivery during failures, nor does the prior work scale particularly well. For example, the Δ-T protocol involves delaying messages for a period of time proportional to the worst-case delay in the system and to estimates of the numbers of messages that might be lost and processes that might crash in a worst-case failure pattern. In the environments of interest to us, these delays would be enormous, and would rise without limit as a function of system size. Similar concerns could be expressed with regard to the δ-causal protocol, which guarantees causal order for delivered messages while discarding any that are excessively delayed. It may be possible to extend these protocols into ones with steady throughput and good scalability, but additional work would be needed.

10. Conclusion

Although many reliable multicast protocols have been developed, reliability can be defined in more than one way, and the corresponding tools match different classes of applications. Reliable protocols that guarantee delivery can be expensive, and may lack stable throughput needed in soft realtime applications, where data is produced very regularly and delivery must keep up. Best effort delivery is inexpensive and scalable, but lacks end-to-end guarantees that may be important when developing mission-critical applications. We see these two observations as representing the core of a debate about the virtues of reliable multicast primitives in building distributed systems.

In this paper, we introduced a new region in the spectrum; one that can be understood as falling between the two previous endpoints. Specifically, we showed that a multicast protocol with bimodal delivery guarantees can be built in many realistic network environments – although not all of them, at least for the present – and that the protocol is also scalable and gives stable throughput. We believe that this new design point responds to important application requirements not adequately addressed by any previous option in the reliable multicast protocol space.

Epilogue

As the military guard led him away in shackles, the Baron suddenly turned. "General," he snarled, "When the Emperor learns of your recklessness, you'll join me in his dungeons. It is impossible to reliably coordinate an attack under these conditions." "Not so," replied the General. "The bimodal guarantee was entirely sufficient."

11. Acknowledgements

Matt Lucas was extremely helpful in providing insight and data documenting conditions under which SRM throughput becomes unstable. Eli Upfal pointed us to work on random graphs used in the scalability analysis. Srinivasan Keshav, Fred Schneider, and Robbert van Renesse all made extensive suggestions after reading an early version of this paper, and the anonymous reviewers of this paper made further suggestions. Their help is gratefully recognized. Werner Vogels and Michael Kalantar also made useful suggestions. Tom Coleman and Jay Blaire helped with access to the Cornell Theory Center SP2, the platform used in our experimental work. The Cornell Theory Center itself was also very helpful, particularly for prioritizing our experiments so that they would be scheduled without long delays.

12. References

Our software is available from: http://www.cs.cornell.edu/Info/Projects/Ensemble/.

[BMR96] R. Baldoni, A. Mostefaoui, and M.Raynal, Causal Delivery of Messages with Real-Time Data in Unreliable Networks, Journal of Real-time Systems, Vol. 10, no. 3, 1996. pp 245-262.

[BPRS96] R. Baldoni, R. Prakash, M. Raynal and M. Singhal. Broadcast with Time and Causality Constraints for Multimedia Applications. Techical Report 2976, INRIA (France), Sept. 1996.

[BFHR98] Ken Birman, Roy Friedman, Mark Hayden, Injong Rhee. Middleware Support for Distributed Multimedia and Collaborative Computing. ACM *Multimedia Computing and Networking* 1998 (MMCN98), Jan. 1998 (San Jose, CA).

[Birman97] K.P. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Company and Prentice Hall, Greenwich, CT. Jan. 1997. URL: http://www.browsebooks.com/Birman/index.html

[Birman99] K.P. Birman. A Review of Experiences with Reliable Multicast. *Accepted for publication, Software Practice and Experience, 1999.*

[CASD85] Flaviu Cristian, H. Aghili, Ray Strong and Danny Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *Proc. 15th International FTCS (1985),* 200-206. See also “Atomic Broadcast in a Real-Time Environment” in *Fault Tolerant Distributed Computing,* Springer-Verlag LNCS 448 (1990), 51-71.

[CS93] David Cheriton and Dale Skeen. Understanding the Limitations of Causal and Totally Ordered Multicast. *Proceedings of the 1993 Symposium on Operating Systems Principles,* Dec. 1993.

[DGHI87] A. Demers et. al. Epidemic Algorithms for Replicated Data Management. Proceedings of the 6th Symposium on Principles of Distributed Computing. (Vancouver, CA; Aug. 1987) 1—12. Also Operating Systems Review 22:1 (Jan. 1988), 8—32.

[FJMLZ95] Sally Floyd, Van Jacobson, Steven McCanne, Ching-Gung Liu, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *Proceedings of the ’95 Symposium on Communication Architectures and Protocols (SIGCOMM).* ACM. August 1995, Cambridge MA. http://www-nrg.ee.lbl.gov/floyd/srm.html

[FPRU90] Uriel Feige, David Peleg, Prabhakar Raghavan and Eli Upfal. Randomized Broadcast in Networks. *Random Structures and Algorithms* 1:4 (1990), 447-460.

[Guo98] Katie Guo. Scalable Membership Detection Protocols. Cornell

University Ph.D. thesis (May 1998), available as TR-98-1684.

[GT92] Richard Golding and Kim Taylor. Group Membership in the Epidemic Style. Technical report UCSC-CRL-92-13, University of California at Santa Cruz, May 1992.

[Hayden98] Mark G. Hayden. *The Ensemble System.* Ph.D. dissertation, Cornell University Dept. of Computer Science. January 1998 (expected).

[HB96] M. G. Hayden and K.P. Birman. Probabilistic Broadcast. Cornell University Computer Science TR 96-1606, September 1996.

[Liu97] Ching-Gung Liu. *Error Recovery in Scalable Reliable Multicast* (Ph.D. dissertation), University of Southern California, Dec 1997.

[LLSG92] Rivka Ladin, Barbara Lishov, L. Shrira and S. Ghemawat. Providing Availability using Lazy Replication. *ACM Transactions on Computer Systems* 10:4 (Nov. 1992), 360-391.

[LMJ97] Craig Labovitz, G. Robert Malan, Farnam Jahanian, “Internet Routing Instability”. Proc. SIGCOMM `97, Sept. 1997, 115-126.

[LP96] J. C. Lin and Sanjoy Paul, “A Reliable Multicast Transport Protocol” Proceedings of IEEE INFOCOM ‘96, Pages 1414-1424. http://www.bell-labs.com/user/sanjoy/rmtp.ps

[LOM94] Kurt Lidl, Josh Osborne, Joseph Malcome. Drinking from the Firehose: Multicast USENET News. USENIX Winter 1994, January, 1994. 33-45.

[Lucas98] Matt Lucas. *Efficient Data Distribution in Large-Scale Multicast Networks* (Ph.D. dissertation), Dept. of Computer Science, University of Virginia, May 1998.

[OSR94] *Various authors.* Rebuttals to [CS93] appearing in *Operating Systems Review*, January 1994.

[OXB99] Oznur Ozkasap, Zhen Xiao, Kenneth P. Birman. Scalability of Two Reliable Multicast Protocols. *Forthcoming, June 1999.*

[Paxson97] Vern Paxson, “End-to-End Internet Packet Dynamics”, Proc. SIGCOMM `97, Sept. 1997, 139-154.

[PC97] Rico Piantoni and Constantin Stancescu. Implementing the Swiss Exchange Trading System. FTCS 27 (Seattle, WA), June 1997, 309-313.

[PHIDEAS] http://www.stna.dgac.fr/projects/, http://aigle.stna7.stna.dgac.fr/

[PSLM97] Sanjoy Paul, K. K. Sabnani, J. C. Lin, S. Bhattacharyya “Reliable Multicast Transport Protocol (RMTP)”, IEEE Journal on Selected Areas in Communications, special issue on Network Support for Multipoint Communication, April 97, Vol 15, No. 3, http://www.bell-labs.com/user/sanjoy/rmtp2.ps

[RBM96] Robbert van Renesse, Kenneth P. Birman, Silvano Maffeis. Horus: A

Flexible Group Communication System. Communications of the ACM 39:4 (April 1996), 76-83.

[RHM98] Robbert van Renesse, Yaron Minsky and Mark Hayden. Gossip-Based Failure Detection Service. *Proc. Middleware '98*. England, Sept, 1998.

[XTP95] XTP Forum. *Xpress Transfer Protocol Specification.* XTP Rev 4.0, 95-20, March 1995.

Appendix A: Formal Analysis of the protocol

In this appendix, we provide an analysis of the pbcast protocol. The analysis is true to the protocol implementation, except with respect to three simplifications. Note that the experimental results suggest that the actual protocol behaves according to predictions even in environments which deviate from our assumptions: in effect, the model is surprisingly robust.

The first of these concerns the initial unreliable multicast. When the process that initiates a pbcast does not crash, remains connected to the network, and the initial multicast is successful, the protocol provides very strong delivery guarantees because the state of the system after the multicast involves widespread knowledge of the message. If this initial multicast fails, however, one could be faced with a pbcast run in which there is just a single process with a copy of the message at the outset. Below, we focus on this conservative assumption: only the initiator initially has a copy of the message. However, we point out the step at which a more realistic assumption could have been made.

A second simplification relates to the model. In the protocol as developed above, each process receives a message and then gossips about that message in subsequent rounds of the protocol. But recall that these rounds are asynchronous and that message loss is independent for each message send event. Accordingly, our protocol is equivalent to one in which a process gossips to all at once to randomly selected processes in the first round after it hears of a message and then ceases to gossip about that message (such a solution might not be as scalable because load would be more bursty, but this doesn't enter into the analysis that follows). This transformation simplifies the analysis, and we employ it below.

Finally, the analysis omits solicitations and retransmissions, collapsing these into the single "gossip" message. As will become clear shortly, this simplification is justifiable for the purposes of our analysis, although there are certainly questions one could ask about pbcast for which it would not be appropriate.

In what follows, R is the number of rounds during which the protocol runs, P is the set of processes in the system, and $\beta*|P|$ is the expected fanout for gossip. The first round (like all others) is gossip based. Using this notation, the abstract protocol that we analyze is illustrated in Figure 15. We give pseudo-code in Appendix B; there, the additional parameter G is used to denote the number of rounds after which a participant should garbage collect a message.

Before undertaking the analysis, we should also comment briefly as to the nature of the guarantees provided by the protocol. With a traditional reliable multicast protocol, any run has an all-or-nothing outcome: either all correct destinations receive a copy of a multicast, or none does so. This section will demonstrate that pbcast has a bimodal delivery distribution: With very high probability, all, or almost all, correct destinations receive a copy. With rather low probability, a small number of processes (some or all of which may be faulty) receive a copy. And the probability of intermediate outcomes, for

```
(* Auxiliary function. *)
to deliver_and_gossip(msg,round):
  (* Do nothing if already received  it. *)
  if received_already then return

  (* Mark the message as being seen and deliver. *)
  received_already := true
  deliver(msg)

  (* If last round, don't gossip. *)
  if round = 0 then return

  let S be a randomly selected subset of P, |S|=|P|*β

  foreach p in S:
    sendto p Gossip(msg,round-1)
```

```
(* State kept per pbcast:  *)
(* have I  received a message *)
(* regarding this pbcast yet? *)
let received_already = false

(* Initiate a pbcast. *)
to pbcast(msg):
  deliver_and_gossip(msg,R)

(* Handle message receipt. *)
on receive Gossip(msg,round):
  deliver_and_gossip(msg,round)
```

Figure 15: Abstract version of pbcast used for the analysis. The abstract version considers just a single multicast message, and differs from the protocol as described earlier in ways that simplify the discussion without changing the analytic results. Pseudo-code for the true protocol appears in Appendix B.

example in which half the processes receive a copy, is so small as to be negligible[8].

System Model

The system model in which we analyze pbcast is a static set of processes communicating synchronously over a fully connected, point-to-point network. The processes have unique, totally ordered identifiers, and can toss weighted, independent random coins. Runs of the system proceed in a sequence of rounds in which messages sent in the current round are delivered in the next. There are two types of failures, both probabilistic in nature. The first are process failures. There is an independent, per process probability of at most τ that a process has a crash failure during the finite duration of a protocol. We call these crashed processes "faulty". The second type of failures are message omission failures. There is an independent, per-message probability of at most ε that a message between non-faulty processes is lost in the network. Message failure events and process failure events are mutually independent. There are no malicious faults, spurious messages, or corruption of messages. We expect that both ε and τ are small probabilities (For example, the values used to compute Figure 4 are $\varepsilon=0.05$ and $\tau=0.001$).

The impact of the failure model above can be described in terms of an adversary attempting to cause a protocol to fail by manipulating the system within the bounds of the model. Such an adversary has these capabilities and restrictions:

- An adversary cannot use knowledge of future probabilistic outcomes, interfere with

[8] Notice, however, that when using the protocol, these extremely unlikely outcomes must still be recognized as possibilities. The examples cited in the body of the paper share the property that for application-specific reasons, such outcomes could be tolerated if they are sufficiently unlikely. An application that requires stronger guarantees would need to use a reliable multicast protocol such as Ensemble's virtually synchronous multicast, tuning it carefully to ensure steady throughput.

random coin tosses made by processes, cause correlated (non-independent) failures to occur, or do anything not enumerated below.

- The adversary has complete knowledge of the history of the current run.
- At the beginning of a run of the protocol, it has the ability to individually set process failure rates, within the bounds [0..τ].
- For faulty processes, it can choose an arbitrary point of failure.
- For messages, it has the ability to individually set message failure probabilities within the bounds of [0..ε], and can arbitrarily select the "point" at which messages are lost.

Note that although probabilities may be manipulated by the adversary, it may only make the system "more reliable" than the bounds, ε and τ.

Over this system model, we layer protocols with strong probabilistic convergence properties. The probabilistic analysis of these properties is, necessarily, only valid in runs of the protocol in which the system obeys the model. The independence properties of the system model are quite strong and are not likely to be continuously realizable in actual system. For example, partition failures are correlated communication failures and do not occur in this model. Partitions can be "simulated" by the independent failures of several processes, but are of vanishingly low probability. Similarly, the model gives little insight into how a system might behave during and after a brief network-wide communication outage. Both types of failures are realistic threats, and in future work we intend to explore their impact on the protocol.

Pbcast Protocol

The version of the protocol used in our analysis is simplified, as follows. We will assume that a run of the pbcast protocol consists of a fixed number of rounds, after which a multicast vanishes from the system because the corresponding message is garbage collected. A process initiates a pbcast by unreliably multicasting the message, and it is received by a random subset of the processes. These gossip about the message, causing it to reach processes that did not previously have a copy, which gossip about it in turn. For our analysis, we consider just a single multicast event, and we adopt the view that a process gossips about a multicast message only during the round in which it first receives a copy of that message. Processes choose the destinations for their gossip by tossing a weighted random coin for each other process to determine whether to send a gossip message to that process. Thus, the parameters of the protocol studied in the analysis are:

- P: the set of processes in the system. N=|P|.
- R: the number of rounds of gossip to run.
- β: the probability that a process gossips to each other process (the weighting of the coin mentioned above). We define the *fanout* of the protocol to be β*N: this is the expected number of processes to which a participant gossips.

Described in this manner, the behavior of the gossip protocol mirrors a class of disease epidemics which nearly always infect either almost all of a population or almost none of it. The pbcast bimodal delivery distribution, mentioned earlier, will stem from the "epidemic" behavior of the gossip protocol. The normal case for the protocol is one in which gossip floods the network in a random but exponential fashion.

Pbcast Analysis

Our analysis will show how to calculate the bimodal pbcast delivery distribution for a given setting, and also how to bound the probability of a pbcast "failure" using a definition of failure provided by the application designer in the form of a predicate on the final system state. It would be preferable to present a closed form solution, however doing so for non-trivial epidemics of the kind seen here is an open problem in epidemic theory. In the absence of closed form bounds, the approach of this analysis will be to derive a recurrence relation between successive rounds of the protocol, which will then be used to calculate an upper bound on the chance of a failed pbcast run.

Notation and Probability Background

The following analysis uses standard probability theory. We use 3 types of random variables. Lower case variables, such as f, r and s, are integral random variables; upper case variables, such as X, are binary random variables (they take values from {0,1}); and upper case bold variables, such as **X**, are integral random variables corresponding to sums of binary variables of the same letter: $\mathbf{X} = \Sigma X_i$.

P{v = k} refers to the probability of the random variable v having the value k. For binary variables, $P\{X\} \equiv P\ \{X = 1\}$. With lower-case integral random variables, in P{r} the variable serves both to specify a random variable and as a binding occurrence for a variable of the same name.

The distributions of sums of independent, identically distributed binary variables are called binomial distributions. If $\forall\ 0 \le i < n$: $P\{X_i\} = p$, then:

$$P\{\mathrm{X} = k\} = \binom{n}{k}(p)^k (1-p)^{n-k}$$

We use relations among random variables to derive bounds on the distributions of the weighted and unweighted sums of the variables. Let X_i, Y_i, and Z_i form finite sets of random variables, and g(i) be a non-negative real valued function defined over integers. If $\forall\ 0 \le i < n$: $P\{X_i\} \le P\{Y_i\} \le P\{Z_i\}$, then:

$$P\{\mathbf{Y} = k\} \le P\{\mathbf{Z} \ge k\} - P\{\mathbf{X} \ge k+1\} \tag{1}$$

$$\sum_{0 \le i < n} P\{\mathbf{X} = i\} g\{i\} \le \sum_{0 \le i < n} P\{\mathbf{Y} = i\} \max_{0 \le j \le i} g(j) \tag{2}$$

These theorems will be applied later in the analysis.

A recurrence relation

The first step is to derive a recurrence relation that bounds the probability of protocol state transitions between successive rounds. We describe the state of a round using three integral random variables: s_t is the number of processes that may gossip in round t (or in epidemic terminology the infectious processes), r_t is the number of processes in round t that have not received a gossip message yet (the susceptible processes), and f_t is the number of infectious processes in the current round which are faulty.

Recall from the outset of this chapter that our analysis is pessimistic, assuming that the initial unreliable broadcast fails and reaches none of the destinations, leaving an initial state in which a single process has a copy of the message and all others are susceptible:

$$s_0 = 1,\ r_0 = N - 1,\ f_0 = 0$$
$$r_{t+1} + s_{t+1} = r_t$$
$$\Sigma_{0 \le t \le R}\ s_t + r_R = N$$

The recurrence relation we derive, $R(s_t, r_t, f_t, s_{t+1})$, is a bound on the conditional probability, given the current state described by (s_t, r_t, f_t), that s_{t+1} of the r_t susceptible processes receive a gossip message from this round. Expressed as a conditional probability, this is: $P\{s_{t+1} \mid s_t, r_t, f_t\}$.

For each of the r_t processes, we introduce a binary random variable, X_i, corresponding to whether a particular susceptible process receives gossip this round. s_{t+1} is equal to the sum of these variables, ΣX_i or equivalently $\mathbf{X}$. In order to calculate $R(s_t, r_t, f_t, s_{t+1})$, we will derive bounds on the distribution of $\mathbf{X}$. Our derivation will be in four steps. First we consider $P\{X_i\}$ in the absence of faulty processes and with fixed message failures. Then we introduce, separately, generalized message failures and faulty processes, and finally we combine both failures. Then we derive bounds on $P\{\mathbf{X} = k\}$ for the most general case.

Fixed message failures. The analysis begins by assuming that there are no faulty processes, and that message delay failures occur with exactly ε probability, no more and *no less*. This second assumption limits the system from behaving with a more reliable message failure rate. In the absence of these sort of failures, the behavior of the system is the same as a well-known (in epidemic theory) epidemic model, called the chain-binomial epidemic. The literature on epidemics provides a simple method for calculating the behavior of these epidemics when there are an unlimited number of rounds and no notion of failures [Bai75]. We introduce constants $p = \beta(1-\varepsilon)$ and $q = 1-p$. p is the probability that both an infectious process gossips to a particular susceptible process and also the message does not experience a send omission failure under the assumption of fixed message failures. (Note that this use of p is unrelated to the reliability parameter p employed elsewhere in the paper; the distinction is clear from context.)

For each of the r_t susceptible processes and its corresponding variable, X_i, we consider the probability that at least one of the s_t infectious processes sends a gossip message which gets through. Expressed differently, this is the probability that not all infectious processes fail to send a message to a particular susceptible process:

$$P\{X_i\} = 1 - (1-p)^{s_t} = 1 - q^{s_t}$$

Generalized message failures. A potential risk in the analysis of pbcast is to assume, as may be done for many other protocols, that the worst case occurs when message loss is maximized. Pbcast's failure mode occurs when there is a partial delivery of a pbcast. A pessimistic analysis must consider the case where local increases in the message delivery probability decrease the reliability of the overall pbcast protocol. We extend the previous analysis to get bounds on $P\{X_i\}$, but where the message failure rate may be anywhere in the range of $[0..\varepsilon]$.

Consider every process i that gossips, and every process j that i sends a gossip message to. With generalized message failures, there is a probability εij that the message experiences a send omission failure, such that $0 \leq \varepsilon_{ij} \leq \varepsilon$. This gives bounds $[p_{lo}..p_{hi}]$ on p_{ij}, the probability that process i both gossips to process j and the message is delivered: $\beta(1 - \varepsilon) = p_{lo} \leq \beta(1 - \varepsilon_{ij}) = p_{ij} \leq p_{hi} = \beta$ (We also have $q_{lo} = 1 - p_{lo}$ and $q_{hi} = 1 - p_{hi}$).

This in turn gives bounds on the probability of each of the r_t processes being gossiped to, expressed using the variables X_{hi} and X_{lo} which correspond to a fixed message failure rate model:

$$1 - q_{lo}^{s_t} = P\{X_{lo}\} \leq P\{X_j\} \leq P\{X_{hi}\} = 1 - q_{hi}^{s_t}$$

Process failures. Introducing process failures into the analysis is done in a similar fashion to that of generalized message failures. For simplicity in the following discussion, we again fix the probability of message failure to ε.

We assume that f_t of the s_t infectious processes that are gossiping in the current round are faulty. For the purposes of analyzing pbcast, there are 3 ways in which processes can fail. They can crash before, during, or after the gossip stage of the pbcast protocol. Regardless of which case applies, a process always sends a subset of the messages it would have sent had it not been faulty: a faulty process never introduces spurious messages. If all ft processes crash before sending their gossip messages then the probability of one of the susceptible processes receiving gossip message, $P\{X_i\}$, will be as though there were exactly $s_t - f_t$ correct processes gossiping in the current round. If all crash after gossiping then the probability will be as though all s_t processes gossiped and none of the f_t processes had failed. All other cases cause the random variables, X_i, to behave with some probability in between:

$$1 - q^{s_t - f_t} = P\{X_{lo}\} \leq P\{X_i\} \leq P\{X_{hi}\} = 1 - q^{s_t}$$

Combined failures. The bounds from the two previous sections are "combined" to arrive at:

$$1 - q_{lo}^{s_t - f_t} = P\{X_{lo}\} \leq P\{X_i\} \leq P\{X_{hi}\} = 1 - q_{hi}^{s_t}$$

Then we apply theorem 1 to get bounds on $P\{\Sigma X_j = k\}$, or $P\{X = k\}$:

$$P\{\mathbf{X} = k\} \leq P\{\mathbf{X}_{hi} \geq k\} - P\{\mathbf{X}_{lo} \geq k+1\}$$

Expanding terms, we get the full recurrence relation:

$$P\{s_{t+1} \mid s_t, r_t, f_t\} \leq \sum_{s_{t+1} \leq i \leq N} \binom{r_t}{i} \left(1 - q_{hi}^{s_t}\right)^i \left(q_{hi}^{s_t}\right)^{r_t - i} - \sum_{s_{t+1}+1 \leq i \leq N} \binom{r_t}{i} \left(1 - q_{lo}^{s_t - f_t}\right)^i \left(q_{lo}^{s_t - f_t}\right)^{r_t - i} \tag{3}$$

We define the right hand side of relation 3 to be $R(s_t, r_t, f_t, s_{t+1})$, "an upper bound on the probability that with s_t gossiping processes of which f_t are faulty, and with r_t processes that have not yet received the gossip, that s_{t+1} processes will receive the gossip this round."

Predicting Latency to Delivery

Still working within the same model[9], we can compute the distribution of latency between when a message is sent and when it is delivered. For the case where the initial multicast is successful, this latency will be determined by the multicast transport protocol: IP-multicast or the tree-based multicast introduced earlier. Both protocols can be approximated as simple packet-forwarding algorithms operating over forwarding trees. If the typical per-round fanout for a node is *b* then a typical message will take $log_b(N)$ hops from sender to destination. Given some information about the distribution of response times for forwarding nodes, we could then calculate a distribution of latency to delivery and an associated variance. Our experience suggests that both mean latency and variance will grow as $log_b(N)$.

When the initial multicast doesn't reach some[10] destinations, the analysis is quite another matter. Suppose the initial multicast infects $(N-1)*(1-\varepsilon_0)$ processes, for some constant ε_0, i.e. $s_0 = 1+(N-1)*(1- \varepsilon_0)$ (the sender always has a copy of the message). If we denote by r_t the number of correct processes that have not yet received a copy of the message by time *t*, then $r_0 = (N-1)* \varepsilon_0$. Given s_t and r_t we now derive a recurrence relation for s_{t+1} and r_{t+1}.

As before, we introduce constants $p = \beta(1-\varepsilon)$ and $q = 1-p$. First, we assume that processes do not crash. For a susceptible process, the probability that at least one of the s_t

[9] Actually, we differ in one respect: the analysis of this subsection explicitly treats gossip to *b* processes during each round. The previous analysis treated all gossip as occuring in the first round.

[10] The case where the initial multicast reaches no processes corresponds to $\varepsilon_0 = 1$, $s_0 = 1$ and $r_0 = N-1$.

infectious processes sends a gossip message which gets through is $1\text{-}q^{s_t}$. Let k be the expected number of newly infected processes. We have: $k = r_t * (1\text{-} q^{s_t})$, $s_{t+1} = s_t + k$, $r_{t+1} = r_t - k$.

Now we can introduce process failures into the analysis. There are three ways that a process can fail: they can crash before, during and after the gossip stage of the protocol. Here we are investigating the relationship between the number of susceptible (hence, correct) processes and the number of gossip rounds. The worst case occurs when all faulty processes fail before the gossip stage (similarly, we can relax the message failure rate, but the worst case occurs when the loss rate is ε). We now have $s_t = s_t * (1\text{-}\tau)$, $r_t = r_t * (1\text{-}\tau)$; $k = r_t * (1\text{-} q^{s_t})$; $s_{t+1} = s_t + k$; $r_{t+1} = r_t - k$. From these relations we can produce graphs such as Figure 6 in Section 6, which shows the number of susceptible processes as a function of the number of gossip rounds with N=1000, the gossip fanout is 1, $\tau = .001$, $\varepsilon = 0.05$ and $\varepsilon_0 = 1.0$ (the initial multicast fails).

Now, define v_t to be the probability that a susceptible process gets infected in any round prior to round t, and w_t to be the probability that a susceptible process gets infected in round t. Observe that in any round, all the currently susceptible processes have equal probability of getting infected. We have $v_t = 1\text{-}r_t/N$ and $w_t = v_t - v_{t-1}$. From this we are able to produce Figure 7 in Section 6, showing the probability for a correct process to receive a message in a certain round. The figure superimposes curves for various values of N: 10, 128 and 1024. Notice that the curve has roughly the same shape in each case and that the peak falls close to $log_{fanout}(N)$.

Failure Predicates

In this section we show how to calculate a bound on the probability of a pbcast, in a particular round and state, ending in an undesired ("failed") state on round R:

$$F_t(s_t, r_t, \bar{f}_t)$$

$$(\bar{f}_t \text{ is the total number of faulty processes that have failed prior to time } t,\ \bar{f}_t = \textstyle\sum_{0\le i<t} f_i)$$

Given F, the reliability of pbcast can be found by examining the value of F for the initial state of the protocol, or $F_0(1, N\text{-}1, 0)$. (Making a more optimistic assumption, we could compute $F_0(N*(1\text{-}\varepsilon), N*\varepsilon, 0)$, giving the expected outcome if the initial multicast reaches all but $N*\varepsilon$ processes). This computation would then yield values for the pbcast parameters which will give a sufficiently high reliability for the desired use.

Values of F are calculated in the context of a predicate that defines whether a run of the protocol failed or not, according to its final state. Failure states correspond to outcomes we wish to avoid. The predicate, P(S, F), is defined over the total number of infected processes (S) (possibly including some faulty processes) and the total number of faulty processes (F). This predicate can be defined differently, depending on the use of pbcast.

To illustrate this, we now give two predicates and use them to explore the predicted reliability of pbcast in the environment of interest to us. The first, predicate I, defines a failed pbcast to be one that reaches more than σN processes in a system but less than (1-

σ)N processes. For a value of σ=0.1 this captures the notion of failure that might make intuitive sense in the examples of the introduction:

$$P(S,F) = (S \geq \sigma N) \wedge (S \leq (1-\sigma)N) \tag{I}$$

Predicate II would make sense if pbcast were used as the basis of a quorum replication algorithm (a topic discussed in [Birman97]. For such applications, the worst possible situation is one in which pbcast reaches about half the processes in the system: neither a clear majority nor a clear majority, with failed processes representing a possible "swing vote". To capture this, the predicate counts failed processes twice: it pessimistically totals all of the processes that may have been infected, so that a failed pbcast is one in which both the percentage of infected processes is less than a majority (with faulty processes counting as being uninfected) and the percentage of infected processes is larger than a minority (with faulty processes counting as infected):

$$P(S,F) = \left(S - F < \frac{N+1}{2} \right) \wedge \left(S + F \geq \frac{N+1}{2} \right) \tag{II}$$

The calculation works backwards from the last round. For each round, we sum over the possible number of failures in this round and the number of infectious processes in the next round. This is done using the calculations for the next round and the recurrence relation, *R*, in order to get the two following equations. The first equation calculates bounds on the probabilities for round R; the second equation calculates bounds for the previous rounds (here we take P(S, F) = 1 if true and 0 if false):

$$F_R(s_R, r_R, \bar{f}_R) \leq \sum_{0 \leq f_R \leq s_R + r_R} P\{f_t\} P(N - r_R, \bar{f}_R + f_R)$$

$$F_t(s_t, r_t, \bar{f}_t) \leq \sum_{0 \leq f_t \leq s_t} (P\{f_t\} \sum_{0 \leq s_{t+1} \leq r_t} R(s_t, r_t, f_t, s_{t+1}) F_{t+1}(s_{t+1}, r_t - s_{t+1}, \bar{f}_t + f_t)) \tag{4}$$

We do not know the exact distribution of P{f$_t$} because individual processes can fail with probabilities anywhere in [0..τ]. However, we can apply theorem 2 to get bounds on the two equations above. For example, the bound for equation 4 is:

$$F_t(s_t, r_t, \bar{f}_t) \leq \sum_{0 \leq f_t \leq s_t} \left(\binom{s_t}{f_t} (\tau)^{f_t} (1-\tau)^{s_t - f_t} \max_{0 \leq i \leq f_t} \sum_{0 \leq s_{t+1} \leq r_t} R(s_t, r_t, i, s_{t+1}) F_{t+1}(s_{t+1}, r_t - s_{t+1}, \bar{f}_t + i) \right)$$

Given the parameters of the system and a predicate defining failed final states of the protocol, we can now compute bounds on the probability of pbcast ending up in a failed state. This was done to obtain the graphs presented in Section 6 of the paper.

Appendix B: Pseudo-Code for the Protocol

The following code is executed, concurrently, by all processes in the system. Notice that, per optimization 5, the "rounds" need not be synchronous. Although round numbers arise in the protocol, they are used in a manner that does not require processes to be in the same round at the same time. For example, if process p is in round n when it sends a gossip message to process q, process q's round number is not relevant. Instead, if q solicits a retransmission from p, it does so using p's round number from the gossip message.

```
pbcast(msg):
    add_to_msg_buffer(msg);
    unreliably_multicast(msg);

first_reception(msg):
    add_to_msg_buffer(msg);
    deliver messages that are now in order;  report gaps after suitable delay;

add_to_msg_buffer(msg):
    slot := free_slot;
    msg_buffer[slot].msg := msg;
    msg_buffer[slot].gossip_count := 0;

gossip_round:                (* Runs every 100ms in our implementation *)
    my_round_number := my_round_number+1;
    gossip_msg := <my_round_number, digest(msg_buffer)>;
    for(i = 0; i < β*N/R; i := i+1)
        { dest := randomly_selected_member;    send gossip_msg to dest;   }
    foreach slot
        msg_buffer[slot].gossip_count := msg_buffer[slot].gossip_count+1;
    discard messages for which gossip_count exceeds G, the garbage-collection limit;

rcv_gossip_msg(round_number, gmsg):
    compare with contents of local message buffer;
    foreach missing message, most recent first
        if this solicitation won't exceed limit on retransmissions per round
            send solicit_retransmission(round_number, msg.id) to gmsg.sender;

rcv_solicit_retransmission(msg):
    if I am no longer in msg.round, or if have exceeded limits for this round
        ignore
    else
        send make_copy(msg.solicited_msgid) to msg.sender;
```