Lightweight Probabilistic Broadcast

P. TH. EUGSTER, R. GUERRAOUI, S. B. HANDURUKANDE,
and P. KOUZNETSOV
Distributed Programming Laboratory, EPFL, Switzerland
and
A.-M. KERMARREC
Microsoft Research, Cambridge, UK

Gossip-based broadcast algorithms, a family of probabilistic broadcast algorithms, trade reliability guarantees against "scalability" properties. Scalability in this context has usually been expressed in terms of message throughput and delivery latency, but there has been little work on how to reduce the memory consumption for membership management and message buffering at large scale.

This paper presents lightweight probabilistic broadcast (*lpbcast*), a novel gossip-based broadcast algorithm, which complements the inherent throughput scalability of traditional probabilistic broadcast algorithms with a scalable memory management technique. Our algorithm is completely decentralized and based only on local information: in particular, every process only knows a fixed subset of processes in the system and only buffers fixed "most suitable" subsets of messages. We analyze our broadcast algorithm stochastically and compare the analytical results both with simulations and concrete implementation measurements.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms: Algorithms, Design, Measurement, Performance, Reliability

Additional Key Words and Phrases: Broadcast, buffering, garbage collection, gossip, noise, randomization, reliability, scalability

This work has been supported by Agilent Technologies, Lombard-Odier, Microsoft Research, Swiss National Science Foundation, and the European Project PEPITO (IST–2001–33234).
This article is a revised and extended version of Eugster et al. [2001b] and also contains material from Kouznetsov et al. [2001].
Authors' addresses: P. Th. Eugster, R. Guerraoui, S. B. Handurukande, and P. Kouznetsov, EPFL-I&C-LPD, Bat. IN, CH-1015 Lausanne, Switzerland; email: {Patrick.EUGSTER, Rachid.GUERRAOUI, Sidath.HANDURUKANDE, Petr.KOUZNETSOV}@epfl.ch; Anne-Marie Kermarrec, Microsoft Research Ltd., 7 JJ Thomson Avenue, Cambridge CB3 OFB, UK; email: Annemk@microsoft.com.

1. INTRODUCTION

Large scale event dissemination. Devising adequate algorithms for reliable propagation of events at large scale constitutes an active research area. While traditional reliable broadcast algorithms scale very poorly [Hadzilacos and Toueg 1993], network-level protocols (e.g., Deering [1994]) lack reliability guarantees whatsoever, and also suffer from scalability problems. The well-known *Reliable Multicast Transport Protocol* (RMTP) [Paul et al. 1997] for instance generates a flood of positive acknowledgements from receivers, loading both the network and the sender.

Probabilistic broadcast algorithms. These algorithms (also known as gossip-based algorithms) [Birman et al. 1999; Lin and Marzullo 1999; Sun and Sturman 2000; Eugster et al. 2003] appear to be more adequate in the field of large scale event dissemination than the "classical" strongly reliable approaches [Hadzilacos and Toueg 1993]. To broadcast a message, a process sends the message to a randomly selected subset of processes. Each process that receives the message also sends the message to a randomly selected subset of processes, and so forth. Though such gossip-based approaches have been proven to have good scalability characteristics in terms of message throughput, it is not clear how they scale in terms of membership management and message buffering. In particular they often rely on the assumption that every process knows every other process. When managing large numbers of processes, this assumption becomes a barrier to scalability. In fact, the data structures necessary to store the *view* of such a large scale membership consume considerable amount of *memory resources*, let aside the *communication* required to ensure the consistency of the membership. Similarly it is not clear how message buffering and purging can be handled in a scalable way without hindering the reliability.

Probabilistic membership. Membership management is sometimes delegated to dedicated servers in order to relieve application processes [Aguilera et al. 1999; Carzaniga et al. 2000; TIBCO 1999]. This only defers the problem, since those servers are limited in resources, as well, and it hampers the very nature of a scalable peer-to-peer architecture. To further increase scalability, the membership should also be *split*; in particular, every participating process should only have a *partial* view[1] of the system. That is, a given process should only know a subset of all processes in the system. In order to avoid the isolation of processes or the partition of the membership, especially in the case of failures, membership information should nevertheless be *shared* by processes to some extent: introducing a certain degree of redundancy between the individual views is crucial to avoid single points of failure. While certain systems rely on a deterministic scheme to manage the individual views [Lin and Marzullo 1999; van Renesse 2000], we introduce a non-deterministic approach in this paper. The local view of every individual member consists of a subset of members, which continuously evolves, but never exceeds a fixed size (a maximum length). In short, after adding new processes to a view, the

[1]The view of a process is the set of processes in the system known by the process.

view is truncated to the maximum length by removing another set of entries. To promote a uniform distribution of membership knowledge among processes, every gossip message—besides notifying events—also piggybacks a set of process identifiers which are used to update the views. The membership algorithm and the effective dissemination of events are thus dealt with at the same level.

Message Buffering. In probabilistic broadcast algorithms messages are buffered temporarily at each participating process. Various approaches have been used to remove messages from buffers and limit the size of these buffers. The simplest approach is to remove messages by random selection. Another approach is to gossip a message a fixed number of rounds after initial reception by a process. Then the message is considered out-of-date and is garbage collected. In these cases, the actual propagation of the messages among members is not taken into account in the garbage collection procedure. The approach we propose consists in estimating the actual propagation of every message among the members and then removing the most propagated messages from the buffers when necessary. This approach leads to a better utilization of buffers.

Contributions. This paper presents a new probabilistic broadcast algorithm, called *lpbcast*: *lightweight probabilistic broadcast*. Our algorithm preserves the inherent throughput scalability of traditional probabilistic broadcast algorithms yet adds a new dimension of scalability in terms of membership management and message buffering. We convey our claim of scalability in two steps. First, we analyze our algorithm using a stochastic approach, pointing out the fact that, with perfectly uniformly distributed individual views, the view size has virtually no impact on the latency of event delivery. We similarly show that, for a given view size, the probability of partitioning in the system decreases as the system grows in size. Second, we give practical implementation results that support the analytical approach, both in terms of simulation and prototype measurements.

For presentation simplicity, we proceed by describing a basic version of our algorithm based on a completely randomized approach. Then optimization techniques for message buffering and membership management are introduced. Finally we present performance improvements in terms of message stability, throughput and membership management.

It is important to notice that our membership and message buffering approaches are not intrinsically tied to our *lightweight probabilistic broadcast (lpbcast)* algorithm. We illustrate this by discussing how to apply them to further improve the scalability of the well-known *pbcast* [Birman et al. 1999] algorithm.

Roadmap. Section 2 gives an overview of related probabilistic broadcast algorithms. Section 3 presents a simple version of our *lpbcast* algorithm and explains its underlying randomized approach. Section 4 analyzes our algorithm in terms of scalability and reliability. Section 5 gives some simulation and practical results supporting the analysis. Section 6 presents optimization techniques to improve the performance of the algorithm in terms of message buffering and propagation. An optimization technique to improve the performance of the algorithm with respect to membership management is described in Section 7.

Section 8 discusses the effect of view size for reliability, and the general applicability of our membership approach and optimization techniques on traditional probabilistic broadcast algorithms.

2. BACKGROUND: PROBABILISTIC BROADCAST ALGORITHMS

The achievement of strong reliability guarantees (in the sense of Hadzilacos and Toueg [1993]) in practical distributed systems requires expensive mechanisms to detect missing messages and initiate retransmissions. Due to the overhead of message loss detection and reparation, algorithms offering such strong guarantees do not scale over a couple of hundred processes [Piantoni and Stancescu 1997].

2.1 Reliability vs Scalability

Gossip, or *rumor mongering* algorithms [Demers et al. 1987], are so-called *epidemiologic* algorithms (as being inspired by epidemics), a family of *probabilistic* algorithms. These have been introduced as an alternative to "traditional" reliable broadcast algorithms. They were first developed for replicated database consistency management [Demers et al. 1987]. The main motivation is to trade the reliability guarantees offered by costly deterministic algorithms against weaker reliability guarantees, but in return obtain very good scalability properties. The basic idea is very intuitive. Every process transmits its information to a randomly selected subset of processes. Every process that receives this information is said to be "infected". This process, in turn, also transmits its information to a randomly selected subset of processes.

The analysis of these algorithms is usually based on the theory of epidemics [Bailey 1975], where the execution is broken down into steps. Generally probabilities are associated to these steps and the degree of reliability is expressed by a probability. For example, Birman et al. [1999] captured reliability in the following way: the probability that a message reaches *almost all* is high, the probability that a message reaches *almost nobody* is small, and the probability that it reaches some intermediate number of processes is vanishingly small. Ideally, the probabilities as well as the "almost" fraction above are precisely quantifiable.

2.2 Decentralization

Decentralization is the key concept underlying the scalability properties of probabilistic broadcast algorithms—the overall load of retransmissions is reduced by decentralizing the effort. More precisely, retransmissions are initiated in most probabilistic broadcast algorithms by having every process periodically (every T ms—*step interval*) send a digest of the messages it has delivered to a randomly chosen subset of processes inside the system (*gossip subset*). The size of the subset is usually fixed, and is commonly called *fanout* (F). Probabilistic broadcast algorithms differ in the number of times the same information is gossiped: every process might gossip the same information only a limited number of times (*repetitions* are limited) [Birman et al. 1999] and/or the same information might be forwarded only a limited number of times (*hops* are limited).

2.3 Memory Management

Memory management in probabilistic broadcast algorithms is a challenging issue. In a highly scalable system, the memory requirement for a process should ideally not change with the system size. Memory is however required to store membership information and messages, until these messages are propagated among "enough" members.

Early approaches [Golding 1992] do not prevent individual views of processes from diverging temporarily, but assume that they eventually converge in "stable" phases. These views however represent the "complete" membership, and this becomes a bottleneck at an increased scale.

2.4 Related Approaches

We exemplify the above-mentioned characteristics of broadcast protocols through short descriptions of the *Bimodal Multicast* [Birman et al. 1999] and *Directional Gossip* [Lin and Marzullo 1999] algorithms below.

Bimodal Multicast. This algorithm, also called *pbcast*, relies on two phases. In the first phase, a "classical" best-effort multicast algorithm (e.g., IP multicast) is used for a first rough dissemination of messages. A second phase assures reliability with a certain probability, by using a gossip-based retransmission: every process in the system periodically gossips a digest of its received messages, and gossip receivers can solicit such messages from the sender if they have not received them previously.

The memory management problem in terms of membership is not directly addressed in Birman et al. [1999], but the authors advocate the use of a complementary algorithm called *Astrolabe* [van Renesse 2000]. This algorithm is a gossip-based resource location algorithm for the Internet and can in that sense be seen as a membership algorithm. This algorithm enables the reduction of the view of each individual process: each process has a precise view of its immediate neighbors, while the knowledge becomes less exhaustive at increasing "distance". The notion of distance is expressed according to the depth of the processes in the hierarchy tree. Astrolabe however only considers the propagation of membership information and it is thus not clear how this membership interacts with *pbcast*.

In the bimodal multicast algorithm, each member stores and gossips the messages for a limited number of rounds. When this limit is exceeded, for a given message, the actual message is purged. However, the "age" of the message, from the time of its publishing is not considered. Instead, when a member receives a message, it starts counting from zero irrespective of the real "age" of the message or the degree of propagation.

Directional Gossip. This algorithm is especially targeted at wide area networks. By taking into account the topology of the network, optimizations are performed. More precisely, a *weight* is computed for each neighbour process, representing the connectivity of that given process. The larger the weight of a process, the more possibilities exist thus for it to be infected by other processes. The algorithm applies a simple heuristic, which consists in choosing

processes with higher weights with a smaller probability than processes with smaller weights. That way, redundant sends are reduced. The algorithm is also based on partial views, in the sense that there is a single *gossip server* per LAN that acts as a bridge to other LANs. This however leads to a static hierarchy, in which the failure of a gossip server can isolate several processes from the remaining system.

Though two different gossip algorithms are used for wide area network gossiping and local area network gossiping, *Directional Gossip* does not address the problem of buffering messages, till the messages are propagated among "enough" members.

It is possible to implement less resource intensive broadcast algorithms in terms of memory and network bandwidth based on Harary graphs [Lin et al. 2000]. But it is not clear how these algorithms perform in very large scale WANs where membership is dynamic, that is, in an environment where the members can join and leave the system at runtime. It would be a very difficult task to construct the Harary graph each time the membership changes.

Lpbcast in perspective. In contrast to the deterministic hierarchical membership approaches in Directional Gossip or Astrolabe, our *lpbcast* algorithm has a probabilistic approach to membership: each process has a *randomly* chosen partial view of the system. *Lpbcast* is lightweight in the sense that it consumes little resources in terms of memory and requires no dedicated messages for membership management; gossip messages are used not only to disseminate event notifications and to propagate digests of received event notifications, but also to propagate membership information.

We combine this membership randomization with effective heuristics for purging out-of-date event notifications (messages) and membership information. *Lpbcast* is completely decentralized in that no global knowledge of membership or message dissemination is used.

3. THE BASIC LIGHTWEIGHT PROBABILISTIC BROADCAST *(LPBCAST)* ALGORITHM

In this section, we present a simple version of our *lpbcast* algorithm for event dissemination based on partial views and fully randomized memory management. We present it as a monolithic algorithm. This is done in order to simplify presentation, and to emphasize the possibility of dealing with membership and event dissemination at the same level. As we pointed out earlier, our membership management scheme can be applied separately for a particular application.

3.1 System Model

We consider a set of processes $\Pi = \{p_1, p_2, \ldots\}$. Processes join and leave the system dynamically and have ordered distinct identifiers. We assume for presentation simplicity that there is no more than one process per node of the network.

Though our algorithm has been implemented in the context of a general topic-based publish/subscribe environment [Eugster et al. 2000; Eugster et al.

2001a], we present it with respect to a single topic, and do not discuss the effect of scaling up topics. In other terms, Π can be considered as a single topic or group, and joining/leaving Π can be viewed as subscribing/unsubscribing from the topic. Such subscriptions/unsubscriptions are assumed to be rare compared to the large flow of events, and every process in Π can subscribe to and/or publish events. Typically events are infrequent relative to their propagation delay.

3.2 Gossip Messages

Our *lpbcast* algorithm is based on non-synchronized periodic gossips, where a gossip message contains several types of information. More precisely, a gossip message serves four purposes:

Event Notifications. A message piggybacks event notifications received (for the first time) since the last outgoing gossip message. Each process stores these event notifications in a buffer named *events*. Every such event notification is gossiped at most once. Older event notifications are stored in a different buffer, which is only required to satisfy retransmission requests.

Event Notification identifiers. Each message also carries a digest (history) of event notifications that the sending process has received. To that end, every process stores identifiers of event notifications it has already delivered in a buffer named *eventIds*. We suppose that these identifiers are unique, and include the identifier of the originator process. That way, the buffer can be optimized by only retaining for each sender the identifiers of event notifications delivered since the last identifier delivered in sequence.

Unsubscriptions. A gossip message also piggybacks a set of unsubscriptions (see Section 3.4 for more details). This type of information enables the gradual removal of processes that have unsubscribed from individual views. Unsubscriptions that are eligible to be forwarded with the next gossip(s) are stored in a buffer named *unSubs*.

Subscriptions. A set of subscription information (see Section 3.4 for more details) is attached to each message. These subscriptions are buffered in a specific buffer named *subs*. A gossip receiver uses these subscriptions to update its view, stored in a buffer *view*.

Note that none of the outlined data structures contain duplicates. That is, trying to add an already contained element to a list leaves the list unchanged. Furthermore, every list has a maximum size, noted $|L|_m$ for a given list L ($\forall L,\ |L| \leq |L|_m$). As a prominent parameter, the maximum length of *view* ($|view|_m$) is denoted l.

3.3 Procedures

The algorithm is composed of two procedures. The first is executed upon reception of a gossip message, and the second is repeated periodically in an attempt to propagate information to other processes.

Gossip reception. According to the lists that are attached to each gossip message, there are several phases in the handling of an incoming message (Figure 1(a)).

I. The first phase consists in handling unsubscriptions. Every unsubscription is applied to the individual view (*view*), and then added to the buffer named *unSubs*. This buffer is then truncated to respect the maximum size limit by removing random elements.
II. The second phase consists in trying to add not yet contained subscriptions to the individual view. These are also eligible for being forwarded with the next outgoing gossip message. Note that the subscriptions potentially forwarded with the next outgoing gossip message, stored in the buffer *subs*, are a random mixture of subscriptions that are present in the view after the execution of this phase, and subscriptions removed to respect the maximum size limit of *view*. A process that has subscribed and that is also active in gossiping, gossips about itself. This is done by inserting its subscriptions in *subs*. Finally, *subs* is also truncated to respect the maximum size limit.
III. The third phase consists in delivering, to the application, event notifications whose ids have been received for the first time with the last incoming gossip message. Multiple deliveries are avoided by storing all identifiers of delivered event notifications in *eventIds*, as previously outlined. Delivered event notifications are at the same time eligible for being forwarded with the next gossip. If there is an Id of an event not received so far in an incoming gossip, an element containing that Id, the current round, and the sender of the gossip, is inserted into *retrieveBuf* for the purpose of retrieving the event notification later.

Gossiping. Each process periodically (every T *ms*) generates a gossip message—as described in Section 3.2—that it gossips to F other processes, randomly chosen among the individual view (*view*) (Figure 1(b)). This is done even if the process has not received any new event notifications since it last sent a gossip message. In that case, gossip messages are solely used to exchange digests and maintain the views uniformly distributed. The network thus experiences little fluctuation in terms of the overall load due to gossip messages, as long as T and the number of processes inside Π remain unchanged.

Retrieving event notifications. The *retrieveBuf* is processed as shown in Figure 1(c) to retrieve event notifications. As stated earlier, when receiving the Id of an undelivered event, an element containing that Id, together with the round number and the Id of the process from which the event Id was received is inserted into the *retrieveBuf* when processing the gossip messages (Figure 1(a), Phase 3). Then for each element in *retrieveBuf*, a test is done to check whether process p_i has waited enough (k rounds) before start fetching the event from others. Then another test is done (using eventIds) to check whether during the period of waiting the event notification was received in a subsequent gossip message. If the event notification was not received, the process p_i asks for the event notification from the process, from which p_i came to know about the

```
upon RECEIVE (gossip) by process p_i
{Phase 1: Update view and unSubs with unsubscriptions}
for all unsub ∈ gossip.unSubs do
   view ← view \ {unsub}
   subs ← subs \ {unsub}
   unSubs ← unSubs ∪ {unsub}
while |unSubs| > |unSubs|_m do
   remove random element from unSubs
{Phase 2: Update view with new subscriptions}
for all newSub ∈ gossip.subs ∧ newSub ≠ p_i do
   if newSub ∉ view then
      view ← view ∪ newSub
      subs ← subs ∪ newSub
while |view| > l do
   target ← random element in view
   view ← view \ {target}
   subs ← subs ∪ {target}
while |subs| > |subs|_m do
   remove random element from subs
{Phase 3: Update events with new notifications}
for all e ∈ gossip.events do
   if e.id ∉ eventIds then
      events ← events ∪ {e}
      LPB-DELIVER(e)
      eventIds ← eventIds ∪ {e.id}
for all e.id ∈ gossip.eventIds do
   if e.id ∉ eventIds then
      element.e.id ← e.id
      element.round ← currentRound
      element.gossip-sender ← gossip.sender
      retrieveBuf ← retrieveBuf ∪ {element}
while |eventIds| > |eventIds|_m do
   remove oldest element from eventIds
while |events| > |events|_m do
   remove random element from events
```

(a) Gossip reception.

```
every T ms at process p_i
gossip.subs ← subs ∪ {p_i}
gossip.unSubs ← unSubs
gossip.events ← events
gossip.eventIds ← eventIds
choose F random members target_1, ... target_F in view
for all j ∈ [1..F] do
   SEND(target_j, gossip)
events ← ∅
upon LPBCAST(e)
events ← events ∪ {e}
```

(b) Gossip emission.

```
for all element ∈ retrieveBuf do
   if currentRound-element.round > k then
      if element.e.id ∉ eventIds then
         ask element.e.id from element.gossip-sender
         if no reply from element.gossip-sender within r rounds then
            asks from a randomly selected process
            if ¬ receive e then
               get element.e.id from element.e.source
         once received e
         events ← events ∪ {e}
         LPBDELIVER(e)
         eventIds ← eventIds ∪ {e.id}
      else
         retrieveBuf ← retrieveBuf \ {element}
```

(c) Retrieving events.

Fig. 1. *lpbcast* algorithm.

event. If the event notification is not received from that process (e.g., due to crash), a randomly selected process (from *view*) is asked for the event notification. If that also failed, then the original sender of the event notification is asked for the event notification. The retrieval phase relies on the assumption that messages are stored for a limited interval of time at each process once a message is received by that process. A discussion on an adequate storage duration, and which messages are to be stored locally can be found in Xiao et al. [2002] and Xiao and Birman [2001].

3.4 Subscribing and Unsubscribing

For presentation simplicity we have not reported the procedures for subscribing and unsubscribing to Π in Figure 1(a). In short, a process p_i that wants to

subscribe Π must know a process p_j that is already in Π. Process p_i will send its subscription to that process p_j, which will gossip that subscription on behalf of p_i. If the subscription of p_i is correctly received and forwarded by p_j, then p_i will be gradually added to the system. Process p_i will experience this by receiving more and more gossip messages. Otherwise, a timeout will trigger the re-emission of the subscription request.

Similarly, when unsubscribing, the process is gradually removed from individual views. To avoid the situation where unsubscriptions remain in the system forever (since *unSubs* is not purged), there is a timestamp attached to every unsubscription. After a certain time, the unsubscription becomes obsolete. Here we assume that an unsubscription and then a subscription again by the same process are sufficiently spread apart in time. It is important to notice that this scheme is not applied to subscriptions: these are continuously dispatched in order to ensure uniformly distributed views.

As specified in 3.3, a process that has subscribed and correct, gossips about itself: if this is not done for example by a failed process, there is a very high probability that the process is to be removed from all the views in the system after a certain amount of time, due to the evolving nature of the membership scheme.

4. ANALYTICAL EVALUATION

This section presents a formal analysis of our *lpbcast* algorithm. The goal is to measure the impact of the size l of the individual views of the processes both (1) on the latency of delivery and (2) on the stability of our membership. The analysis differs from the one proposed in Birman et al. [1999], precisely because our membership is not global and event notification forwarding is not limited to a particular number of times (hops are not limited), and event notifications can be forwarded several times by the same process without a strict limit (repetitions are not limited). Here, we do not distinguish between event notification and event notification identifiers; that is, we do not consider retransmissions. We first introduce a set of assumptions without which the analysis becomes extremely tedious, but which have very little impact on its validity.

4.1 Assumptions

For our formal analysis, we consider a system Π composed of n processes, and we observe the propagation of a single event notification. We assume that the composition of Π does not vary during the run (consequently n is constant). As mentioned, and according to the terminology applied in epidemiology, a process that has delivered a given event notification will be termed *infected*, otherwise *susceptible*.

The stochastic analysis presented below is based on the assumption that processes gossip in synchronous rounds, and there is an upper bound on the network latency, which is smaller than a gossip period T. T is furthermore constant and identical for each process, just like the fanout F. We assume furthermore that failures are stochastically independent. The probability of a message loss does not exceed a predefined $\varepsilon > 0$, and the number of process

crashes in a run does not exceed $f < n$. The probability of a process crash during a run is thus bounded by $\tau = f/n$. For the following computations and also for the simulations in the next section, we will assume $\tau = 0.01$ and $\varepsilon = 0.05$. We do not take into account the recovery of crashed processes, nor do we consider Byzantine (or arbitrary) failures.

Assume that at round r, each process p has an *independent uniformly distributed* random view of size l of known subscribers: the probability that a given process belongs to the view of p_i at round r is $l/(n-1)$. The probability that a given process p_j belongs to the view of p_i at round $r+1$ is the sum of the probability that p_j was in the view of p_i at round r and was not removed during round $r+1$ and the probability that p_j entered to the view as a result of a gossip reception at round $r+1$:

$$\frac{l}{n-1}\frac{l}{|subs|_m F + l} + \left(1 - \frac{l}{n-1}\right)\frac{l}{n-1}.$$

Thus, for $l \ll |subs|_m F$, the probability can be roughly estimated as $l/(n-1)$ which corresponds to the uniform distribution. For the analysis below, we indeed take the uniform distribution of views as an assumption on the model. In other terms, every combination of l processes within $(n-1)$ processes (according to the algorithm presented in Figure 1(a), a process p_i will never add itself to its own local view $view_i$) is equally probable for every individual view. For reasons of simplicity, we will also refer to such views as *uniform views* (though this is a language abuse). The *expected* number of processes that know a given process is thus equal to l. These views are not constant, but continually evolving.

4.2 Event Propagation

Let e be an event produced (*lpb-cast*) by a given process. We denote the number of processes infected with e at round r as $s_r \in [1..n]$. Note that when e is first introduced into the system at round $r = 0$, we have $s_r = 1$.

We define a lower bound on the probability that a given susceptible process is infected by a given gossip message as:

$$\begin{aligned} p &= \left(\frac{l}{n-1}\right)\left(\frac{F}{l}\right)(1-\varepsilon)(1-\tau) \\ &= \left(\frac{F}{n-1}\right)(1-\varepsilon)(1-\tau) \end{aligned} \tag{1}$$

In other terms, p is expressed as a conjunction of four conditions, namely that (1) the considered process is known by the process that gossips the message, (2) the considered process is effectively chosen as target, (3) the gossip message is not lost in transit, and (4), the target process does not crash. As a direct consequence of the uniform distribution of the individual views, p does not depend on l.

Accordingly, $q = 1 - p$ represents the probability that a given process is *not* infected by a given gossip message. Given a number i of currently infected processes, we are now able to define the probability that exactly j processes will be infected at the next round ($j - i$ susceptible processes are infected during the

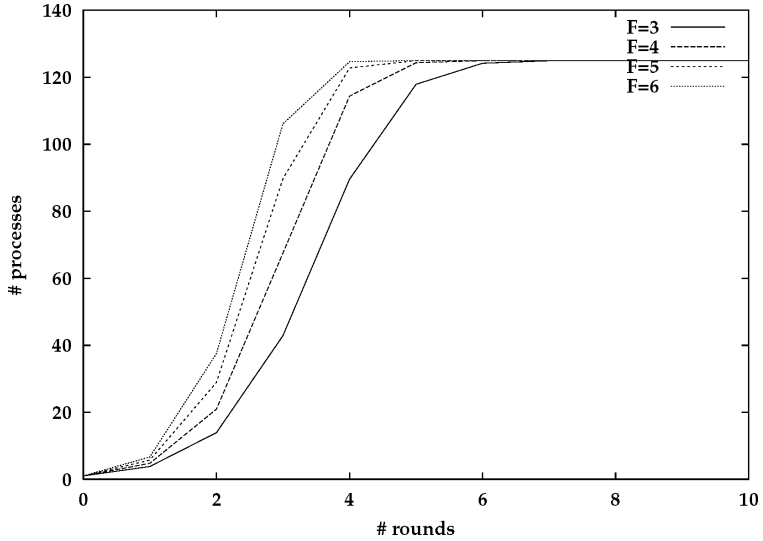


Fig. 2. Analysis: expected number of infected processes for a given round with different fanout values.

current round). The resulting Markov Chain is characterized by the following probability p_{ij} of transiting from state i to state j:

$$\begin{aligned} p_{ij} &= P(s_{r+1} = j \,|\, s_r = i) \\ &= \begin{cases} \binom{n-i}{j-i}(1-q^i)^{j-i}q^{i(n-j)} & j \geq i \\ 0 & j < i \end{cases} \end{aligned} \tag{2}$$

The distribution of s_r can then be computed recursively:

$$\begin{aligned} P(s_0 = j) &= \begin{cases} 1 & j = 1 \\ 0 & j > 1 \end{cases} \\ P(s_{r+1} = j) &= \sum_{i \leq j} P(s_r = i) p_{ij} \end{aligned} \tag{3}$$

4.3 Gossip Rounds

By considering that the two parameters τ and ε are beyond the limits of our influence, the determining factors according to the analysis are the fanout F and of course the system size n.

Fanout. Figure 2 shows the relation between F and the number of rounds it takes to broadcast an event to a system composed of $n = 125$ processes. The figure shows that increasing the fanout decreases the number of rounds necessary to infect all processes. When the product of the fanout and the number of rounds a message is being gossiped in the system is too high, there will be more redundant messages received by each process, which limits performance

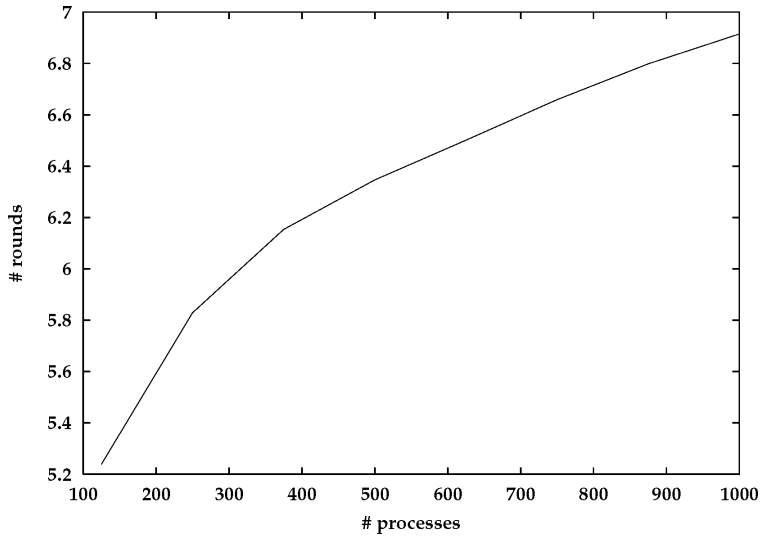


Fig. 3. Analysis: expected number of rounds necessary to infect 99% of Π, given a system size n.

(and overloads the network). Here, we consider the messages that are already disseminated "enough" and that are being gossiped further as redundant messages. These messages do not contribute to the dissemination process or to improve the reliability. Furthermore, F is in our case tightly bound, since $F \leq l$ must always be ensured. The goal of this paper however is not to focus on finding the optimal value for F. In the following simulations and measurements, the default value for the fanout will be fixed to $F = 3$. The optimal choice of fanout value is discussed within a different context in Kermarrec et al. [2003].

System size n. The number of gossip rounds it takes to infect all processes intuitively depends on the number of processes in the system. Figure 3 presents the expected number of rounds necessary for different system sizes. The figure conveys the fact that the number of rounds increases logarithmically with an increasing system size, as detailed in Bailey [1975].

View size l. According to Equation 2, the view size l does not impact the time it takes for an event notification to reach every member. This leads to the conclusion that, besides the condition $F \leq l$, the amount of knowledge concerning the membership that each process maintains does not have an impact on the algorithm performance. The expected number of rounds it takes to infect the entire system depends on F, but not on l. This consequence derives directly from our assumption that the individual views are uniform. Intuitively, the algorithm shown in Figure 1(b) supports this hypothesis by having the following properties: (1) each process periodically gossips, and (2) each process adds its own identity to each gossip message. Based on experimental results, we will discuss the validity and impact of this assumption in more detail in Sections 5 and 8.

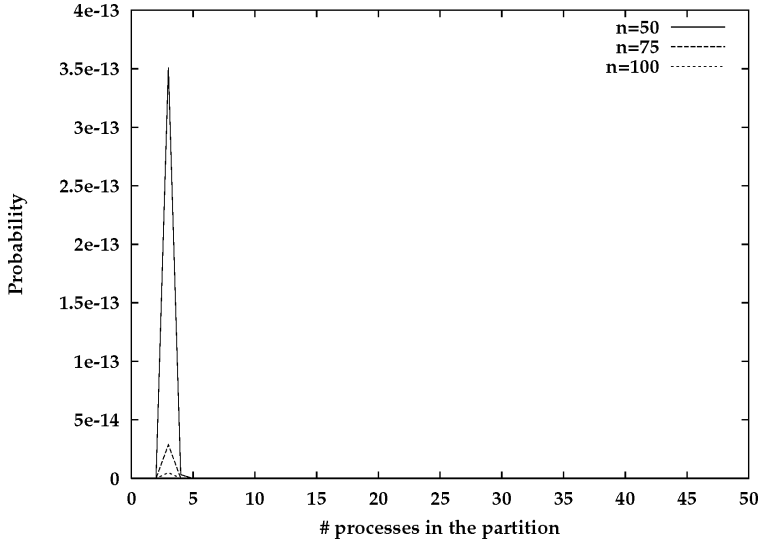


Fig. 4. Analysis: probability of partitioning in systems of different sizes.

4.4 Partitioning

One could derive that the view size l can be chosen arbitrarily small (provided that the requirements with respect to F are met). This is rather dangerous, since with small values for l the probability of system partitioning increases. This occurs whenever there are two or more distinct subsets of processes in the system, in each of which no process knows about any process outside its partition.

Probability of partitioning. The creation of many partitions can be seen as partitioning the system recursively. In other terms, by expressing an upper bound on the probability of creating a partition of size i ($i \geq l + 1$) inside the system, we also include the creation of more than two subsets. The probability $\Psi(i, n, l)$ of creating a partition of size i inside a system of size n with a view size of l is given by the following equation:

$$\Psi(i, n, l) = \binom{n}{i} \left(\frac{\binom{i-1}{l}}{\binom{n-1}{l}} \right)^{i} \left(\frac{\binom{n-i-1}{l}}{\binom{n-1}{l}} \right)^{n-i} \tag{4}$$

It can easily be shown that, for a fixed system size n, $\Psi(i, n, l)$ monotonically decreases when increasing l. Similarly, for a fixed view size l, $\Psi(i, n, l)$ monotonically decreases when increasing n. Figure 4 depicts this for n, by fixing l to 3. The fact that the membership becomes more stable with an increased n can be intuitively reproduced since, with a large system, membership information becomes more sparsely distributed, and the probability of having concentrated exclusive knowledge becomes vanishingly small.

In time. According to our model, the distribution of membership information in a certain round does not depend on the distribution in the previous round.

Thus we can define the probability that there is *no* partitioning up to a given round r as:

$$\phi(n, l, r) = \left(1 - \sum_{l+1 \leq i \leq \lfloor n/2 \rfloor} \Psi(i, n, l)\right)^r \tag{5}$$

This probability decreases very slowly with r. It takes $\approx 10^{12}$ rounds to end up with a partitioned system with the probability of 0.9 with $n = 50$ and $l = 3$. For a given expected system run-time, we can easily compute the minimal view size that guarantees the absence of partitioning with a given probability.

A priori, it is not possible to recover from such a partition. To avoid this situation in practice, we elect a very limited set of privileged processes, which are constantly known by each process. They are periodically used to "normalize" the views (in particular for bootstrapping). Alternatively, we could use a set of dedicated processes to collaborate in keeping track of the total number of processes.

5. EXPERIMENTAL RESULTS

In this section, we compare the analytical results obtained in the previous section with (1) simulation results and (2) results collected from measurements obtained with our actual implementations. In short, the results show a very weak dependency between l and the degree of reliability achieved by *lpbcast*, but we can neglect this dependency in a practical context.

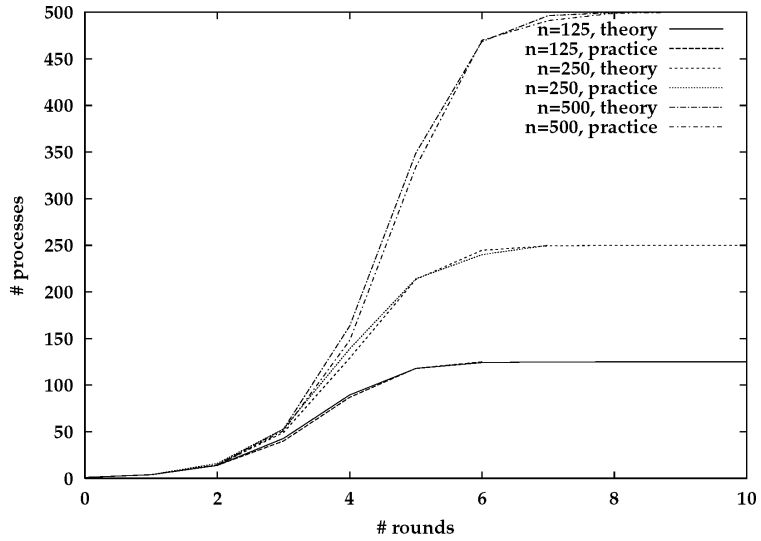
In our test runs, we did not consider retransmissions, that is, once a process has received the identifier of an event notification, the event notification itself is assumed to have been received. This has been done to comply with related work (in some cases it is sufficient for the application to know that it has missed some message(s), and in other cases, subsequent messages can replace the missed messages [Orlando et al. 2000]).

5.1 Simulation

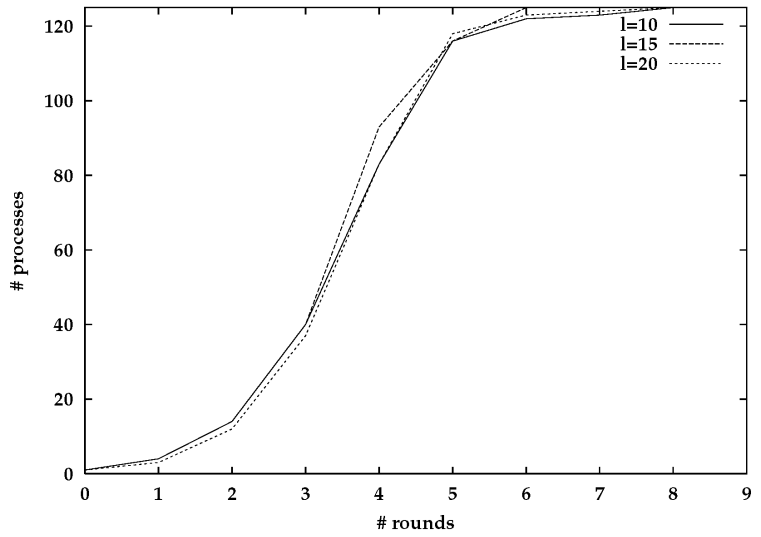
In a first attempt, we have simulated the entire system on a single machine. More precisely, we have simulated synchronous gossip rounds in which each process gossips once. The results obtained from these simulations support the validity of our analysis.

Number of gossip rounds. As highlighted in the previous section, the total number of processes n has an impact on the number of gossip rounds it takes to infect all processes. Figure 5(a) conveys the results obtained from our analysis by comparing them with values obtained from simulation, showing a very good correlation.

Impact of l. According to the analysis presented in the previous section, the size l of the individual views has no impact on the number of gossip rounds it takes to infect every process in the system. Figure 5(b) reports the simulation results obtained for different values of l in a system of 125 processes. It conveys a certain dependency between l and the number of gossip rounds required



(a) Analysis vs simulation.



(b) Number of rounds necessary to infect a system with different values for l.

Fig. 5. Simulation results.

for the successful dissemination of an event in Π, slightly contradicting our analysis. This stems from the fact that we have presupposed uniform views for the analysis, and have considered these as completely independent of any "state" of the system. An exhaustive analysis would have to take into account the exact composition of the view of each process at each round. This would however lead to a very complex Markov Chain, with an impracticable size.

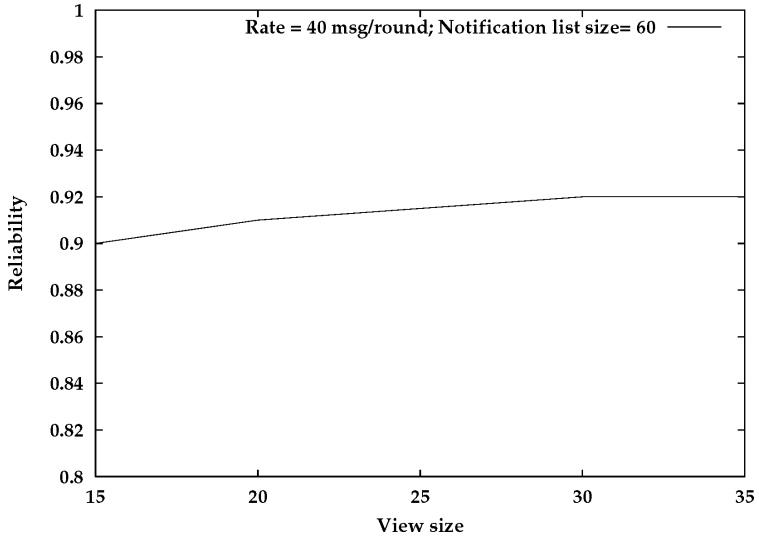


Fig. 6. Measurements: degree of reliability.

Given the very good correlation between simulation and analysis, assuming independent and uniform views seems reasonable.

5.2 Measurements

We present here concrete measurements that attempt to capture the degree of reliability achieved with *lpbcast*, and confirm the results obtained from simulation.

Test environment. Our measurements involved two LANs with, respectively 60 and 65 SUN Ultra 10 (Solaris 2.6, 256 Mb RAM, 9 Gb harddisk) workstations. The individual stations and the different networks were communicating via Fast Ethernet (100 Mbit/s). The measurements we present here were obtained with all 125 processes; in each round 40 new events were injected into the system. To conform to our simulations, F was fixed to 3 and the size of the *events* buffer was set to 60.

Impact of the view size. Figure 6 shows the impact of l on the degree of reliability achieved by our algorithm. The measure of reliability is expressed here by the probability for any given process of delivering any given event notification ($1 - \beta$, cf. Section 2). The reliability of the system seems to deteriorate slightly with a decreasing value for l. Intuitively this is understandable, since our simulation results have already shown that latency *does* increase slightly by decreasing l. With an increased latency, the probability that a given message is purged from all buffers before all processes have been infected becomes higher.

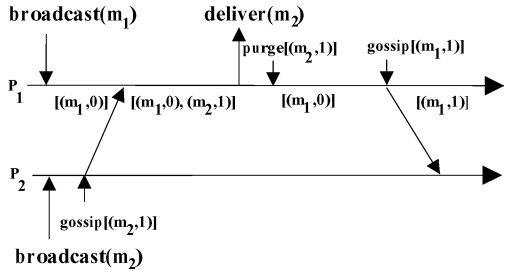


Fig. 7. Age-based purging scenario.

5.3 Optimization

In *lpbcast*, every process locally buffers information about published messages and membership. To preserve scalability, the entities in the buffers need to be removed periodically. So far we considered a simple strategy where buffers are purged in a randomized manner. Instead of randomization it would be better to remove information that is well disseminated among members and keep the least disseminated information.

In *lpbcast*, there are two types of buffers: the *events* buffer for buffering messages (event notifications) and the *subs* buffer for buffering membership information. In the next two sections we discuss optimization schemes that are applicable to each of these two buffers. Though for clarity these two optimization techniques are discussed separately, it should be noted that they can be applied together.

6. AGE-BASED MESSAGE PURGING

Age-based message purging is an optimization that is applied to make a good use of *events* buffers; an *events* buffer stores messages (published events) once received by processes as described in Section 3.

6.1 The Principle

Here, the events are referred to as messages (meaning application-generated messages) that are propogated using *lpbcast*. In age-based message purging, the idea is to associate with every message some integer, corresponding to the number of rounds the message has spent in the system by the current moment. Roughly speaking this number represents the *age* of the message and is updated in every gossip round. Every process participating in a gossip-based information exchange periodically receives updates and stores some of them in the message history buffer. Informally, the age reflects the dissemination degree of a message in the system.

A scenario of age-based memory management is presented in Figure 7 for a simple case where the buffer size is limited to 1 message. At process p_1, the scenario can be described as follows:

I. Message m_1 is broadcast and an item $(m_1, 0)$ is stored in the buffer. The age of the message m_1 is equal to 0 since it has not been gossiped yet.

II. Gossip message $(m_2, 1)$ is received. The age of m_2 is equal to 1 because m_2 has been gossiped once.
III. Message m_2 is delivered to the application layer This is done only if m_2 is not already delivered.
IV. Item $(m_2, 1)$ is purged from the buffer since it is "the oldest" one.
V. The age of the message m_1 is incremented and gossip message $(m_1, 1)$ is sent.

In the purging procedure, useful messages are kept in the buffers with higher probability than the *noisy* ones. Noisy messages represent the event notifications that are already disseminated among "enough" processes and that need to be purged out from buffers. All message purging decisions are taken locally and do not use any form of agreement with the rest of the system.

6.2 Optimized *lpbcast*

Figure 8 presents our variant of lpbcast optimized with aged-based purging. We describe here only the part relevant to age-based message purging and we do not recall other aspects of *lpbcast* introduced in Section 3.

Broadcast message. When a message is broadcast (lpbcast), its age value is initialized to 0. The message is then added to the message history events and if its maximal size is exceeded, the "oldest" elements are purged. This is done by the auxiliary function REMOVE_OLDEST_NOTIFICATIONS() (Figure 9(a)).

Gossip transmission. This phase is executed periodically (every T seconds) and includes randomly choosing the gossip target and sending the gossip. The ages of stored messages are incremented.

Gossip reception. When a received gossip is processed by process p_i, the messages that have not been seen before by process p_i are delivered and stored in the buffer. If a received message has been seen before, and the copy of it is stored in the buffer, its age is updated: the maximum of the ages of received and stored messages is taken. As before, REMOVE_OLDEST_NOTIFICATIONS() is invoked to purge the "oldest" items.

When choosing an element to remove from the buffer, two criteria are applied (see auxiliary function REMOVE_OLDEST_NOTIFICATIONS() in Figure 9(a)). A message is purged if: (1) (out-of-date) the message is received a long time ago, with respect to more recent messages from the same broadcast source. This period of time is measured in gossip rounds and compared with the LONG_AGO parameter. (2) (oldest) the message has the largest age parameter in the buffer.

The truncating criteria are applied sequentially: "out-of-date" first. In other words, if after purging all out-of-date messages, the buffer limit is not exceeded, no further purging occurs.

6.3 Evaluation Criteria

In this section we discuss the evaluation criteria and the measurement environment for comparing the improved version of our *lpbcast* algorithm with our basic version of *lpbcast* (Section 3).

Process p_i:

```
upon LPBCAST(e)
...
e.age← 0
REMOVE_OLDEST_NOTIFICATIONS()

in every T ms
for all e ∈ events do
   e.age←e.age+1
...
SEND_GOSSIP()

upon RECEIVE (gossip)
...
{Update the ages}
for all e ∈ gossip.events do
   if e′ ∈ events such that
   e′.id = e.id and e′.age < e.age then
      e′.age ← e.age
REMOVE_OLDEST_NOTIFICATIONS()
...
for all m∈ gossip.subs do
   if m′ ∈ view such that
   m′ = m then
      m′.Frequency ← m′.Frequency + 1
   else
      m.Frequency ← m.Frequency + 1
      view ← view ∪ m
   if m′′ ∈ subs such that
   m′′ = m then
      m′′.Frequency ← m′′.Frequency + 1
   else
      m.Frequency ← m.Frequency + 1
      subs ← subs ∪ {m}
while |view| > l do
   target ← SELECT_PROCESS (view)
   view ← view \ {target}
   subs ← subs ∪ {target}
while |subs| > |subs|_m do
   target ← SELECT_PROCESS (subs)
   subs ← view \ {target}
```

Fig. 8. The optimized lpbcast.

The measurements we present in Section 6.4 and Section 7.4 have been obtained with 60 processes. The message history buffer size at every process is limited to 30. The fanout, the number of other processes each process gossips to per round, is fixed to 4. For modelling failures, we use a process crash ratio equal to 5% and a message loss ratio equal to 10%. Where not explicitly mentioned, the broadcast rate is 30, that is, 30 new messages are introduced into the system per gossip round. In this section as well as in the next we used only 60 as opposed

```
REMOVE_OLDEST_NOTIFICATIONS()
{Out-of-date}
while |events| > |events|_m and events contains
e and e' such that (e.source = e'.source and (e.id
- e'.id) > LONG_AGO) do
   events ← events/ {e'}
{Age}
while |events| > |events|_m do
   let e' ∈ events such that
        e'.age = max_{e∈events}(e.age)
   events ← events/ {e'}
```

(a) REMOVE_OLDEST_NOTIFICATION function.

```
SELECT_PROCESS(List)
found ← false
avg ← average of Frequency in the List
while (¬ found) do
   target ← random element in List
   if target.Frequency > k(avg) then
      found=true
   else
      target.Frequency ← target.Frequency + 1
return target
```

(b) SELECT_PROCESS function.

Fig. 9. Auxiliary functions.

to 125 processes as in the previous experiments due to the large amount of data that need to be stored in the secondary storage device. This is the data used for the analysis.

The criteria we use for the comparative analysis are the following:

Delivery ratio. This is the ratio between the average number of messages delivered by a process per round and the number of messages broadcast per round. We analyze long run behaviour of simple and optimized versions of the algorithm comparing this ratio. The delivery ratio represents the efficiency of the algorithm in terms of message dissemination.

Redundancy. We measure the proportion of redundant messages that are received by the same process in a given round.

Throughput. We measure the throughput of a broadcast algorithm as a maximum broadcast rate the algorithm can stand, providing certain stability level. In our case the stability level is fixed to 90%. A message becomes *stable* when it has been delivered by all or a predefined part of the processes, at which point it can be discarded. In this experiment we considered messages that are delivered to all the processes. In other words, we found the throughput at which 90% of the produced messages are delivered to all the processes. The criterion captures the relationship between the throughput stability and minimal view size that guarantees it (for two schemes of garbage collecting).

Fault tolerance. We model system failures and estimate the delivery ratio to demonstrate that our age-based memory management does not impact the high level of fault tolerance, an inherent property of gossip-based algorithms.

6.4 Results

We present here experimental results for the two versions (*lpbcast* and optimized *lpbcast*) in our prototype implementation. The practical evaluation

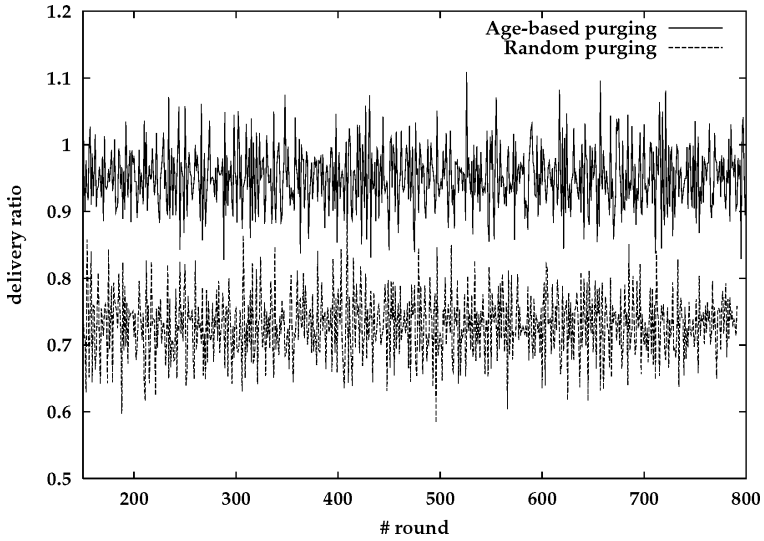


Fig. 10. Measurements: delivery ratio of initial (random purging) and optimized (age-based purging) versions of lpbcast.

results clearly confirm the fact that our age-based purging scheme enhances the performance of the broadcast algorithm in terms of message delivery efficiency and throughput.

In our measurements, 30 messages are published at each round. Messages can be delivered a few rounds after publication. As a result, some particular processes can deliver more than 30 messages in some particular rounds (i.e. some messages published in the present round as well as some previously published messages). Because of this, delivery ratio can be more than 1 in some particular rounds.

We can summarize the improvements as follows:

Delivery ratio. Figure 10 depicts the message delivery efficiency provided by the broadcast algorithms implementing age-based and randomized buffering schemes. The delivery ratio for age-based buffering is considerably higher.

Throughput. The throughput estimation presented in Figure 11 shows that age-based buffering enables a broadcast algorithm to improve the throughput by at least a factor of 2 while providing the same level of message stability.

Reduction of noise. Figure 12 shows that the proportion of redundant messages given by our age-based message purging scheme is smaller in comparison with random purging.

Robustness. Despite process crashes and message losses, reliability is not sacrificed by our age-based message purging (Figure 13): the average delivery ratio is almost the same for both age-based and randomized buffering schemes.

Our age-based message purging scheme does not decrease the *useful* redundancy level of an algorithm. The average number of gossips per round is the same for a randomized and an age-based message purging scheme: it does

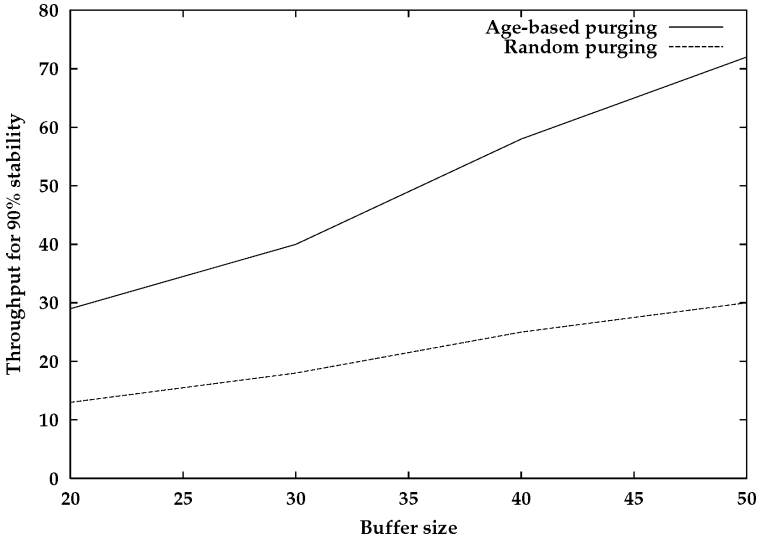


Fig. 11. Measurements: throughput of initial (random purging) and optimized (age-based purging) versions of lpbcast (message stability level 90%).

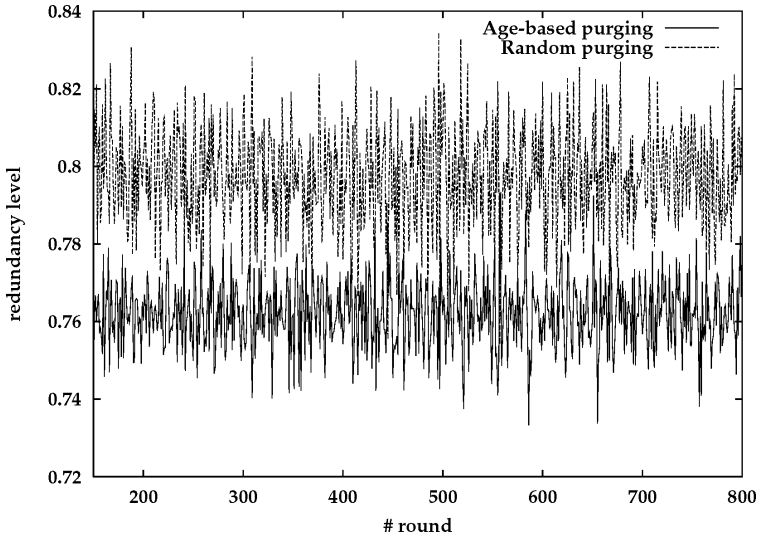


Fig. 12. Measurements: redundancy level for initial (random purging) and optimized (age-based purging) versions of lpbcast.

not depend on the way the messages are buffered. At the same time, the *distribution* of messages that are gossiped is different: when age-based message purging is implemented, it is less probable to gossip a noisy message compared to an algorithm where a random approach is used. This is the source of the considerable performance gains shown in Figures 10 and 11.

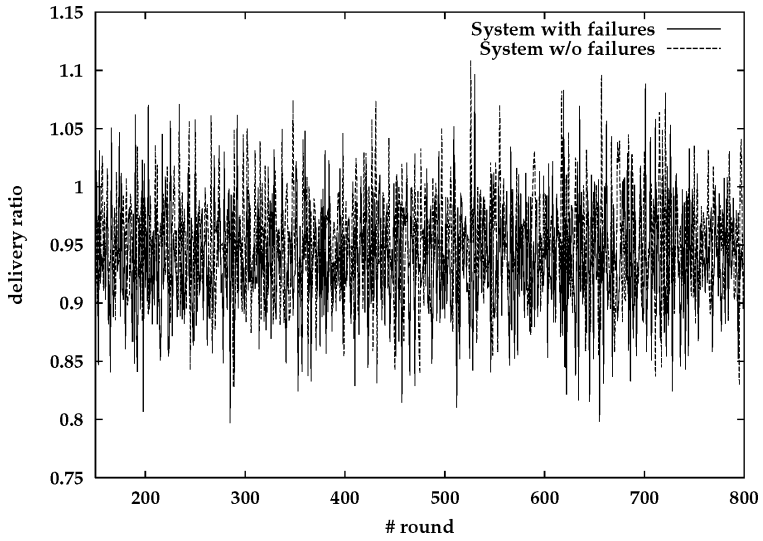


Fig. 13. Measurements: impact of failures on overall delivery ratio. Process crash ratio = 5%, message loss ratio = 10 %.

We should mention that for the practical evaluations, we study here the circumstances that are somewhat beneficial for the age-based buffering scheme: we consider only the gossip-based dissemination phase (there is no "unreliable" phase as in Birman et al. [1999]) and we model the high and regular broadcast rate. We have run a number of experiments in less stressful conditions, in particular, when the broadcast rate is small with respect to the buffer sizes. Those results are not so impressive, although the advantages of our age-based memory management scheme over a random one still hold.

7. FREQUENCY BASED MEMBERSHIP PURGING

Frequency-based membership purging is an optimization that is applied to *subs* buffers; the *subs* buffer stores information about subscribers once received by processes as described in Section 3.

7.1 The Principle

In the simple version of our *lpbcast* algorithm, the membership information is stored in the buffer *subs* and, as it grows, the entities are removed from it by random selection. However there could be well known members in the group as well as less known members. For example, a newly joined member will not be known by most of the members initially. If we use random selection to maintain the size of the *subs* buffer, there could be a possibility where a lesser known member in the group is removed from the buffer, while keeping the information about well known members. For this reason, a new member would not be able to join the group "quickly". Apart from this, as lesser known members could be removed from the buffers, there could be isolation where a member is not known by any other member.

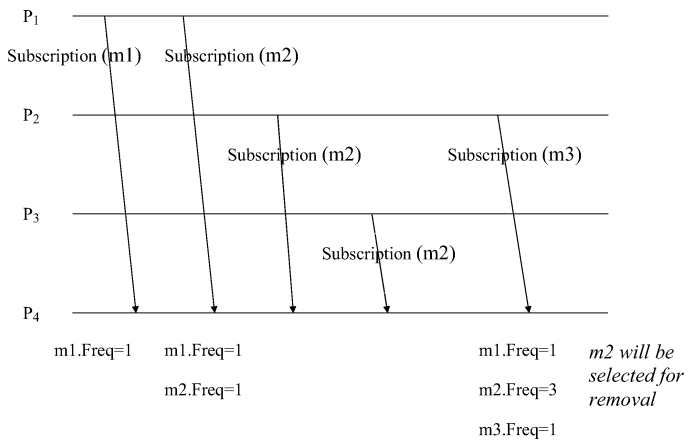


Fig. 14. Simple example of memory management for membership information.

To avoid this drawback, we suggest purging membership information based on a heuristic value. In this approach, an integer known as *frequency* is associated with each membership information stored in the *subs* buffer. The frequency variable represents the number of times the information about a member is heard by a particular member. When elements from the *subs* buffer need to be removed, the frequency variable is used in combination with a random selection. We use a random selection to promote uniform distribution of membership information.

Figure 14 shows a simple scenario of frequency-based memory management for membership. The description of the events at p_4 is as follows:

I. Process p_4 receives subscription information about member m_1 and puts it into the *subs* buffer after setting the frequency associated with m_1 to 1.
II. Subsequently, p_4 receives subscription information about member m_2 three times. Each time p_4 increments m_2.*frequency* by one.
III. p_4 receives subscription information about member m_3 and sets m_3.*frequency*=1.
IV. If we assume that the maximum size of *subs* buffer is 2 (in number of elements), once m_3 is received, m_2 will be selected for removal by SELECT_PROCESS() function.

7.2 Optimized *lpbcast*

Figure 8 presents a variant of *lpbcast*. We describe here the part relevant to frequency-based membership purging.

I. Subscribe: p_j sends subscription message m after setting m.*frequency* to 0.
II. Once a gossip message is received by p_i (Figure 8):
    1. if m is in the view v, p_i increments the frequency of m contained in v; if m is not in v then p_i adds m to the v and increments the value of frequency.

2. if m is in the *subs* s, p_i increments the frequency of m contained in s; if m is not in the s then p_i adds m to s and increments the value of frequency.
3. when the size of the view v or *subs* is above the allocated value, p_i truncates the view v or *subs* by selecting an element using the SELECT_PROCESS function.

III. The operation of the SELECT_PROCESS function is shown in Figure 9(b) and described below.
1. p_i finds the average (avg) of the frequency of all the elements.
2. p_i selects an element (e) from the given *List* randomly.
3. If *e.frequency* > k(avg) then return e as the selected value; Else increment *e.frequency* by one and go to Step II and proceed. 0<k<=1

7.3 Evaluation Criteria

In this section we discuss the evaluation criteria and the measurement environment we used to compare the improved version of *lpbcast* with our basic version (Section 3), with respect to frequency-based membership purging.

The measurements we present here have been obtained with 60 processes. The fanout—the number of other processes each process gossips to per round, is again fixed to 4.

The criteria we use for the comparison are:

Propagation Delay. We measure how fast information about a new member propagates among other members, with and without optimization. This represents how fast a new member can effectively join the group.

Membership Management. Removal of process Ids from *subs* is analyzed to check the performance improvement in terms of buffer utilization due to the optimization. Degree of propagation of removed process Ids are measured. The number of times membership information about a particular process is seen by other processes is considered as the degree of propagation. This is equivalent to the value of *Frequency* associated with each process Id. Process Ids are removed from the *subs* list when their size grows beyond the limited size. Once this is done if lesser known processes are removed from the list it could lead to isolation. By using optimization we try to avoid this. To test the effectiveness of the optimization we considered a large number of removals from the *subs* list and found the degree of propagation for each removed process Id (i.e., the value of *Frequency* of each removed Id).

7.4 Results

The simulation results show that frequency-based membership management enhances the membership information propagation. The improvements obtained due to the optimization can be summarized as follows.

Reduction of propagation delay. Figure 15 is a graph of the case where a new member sends a subscription request at time=0; it plots the number of members who came to know about the new member against time. It can be seen

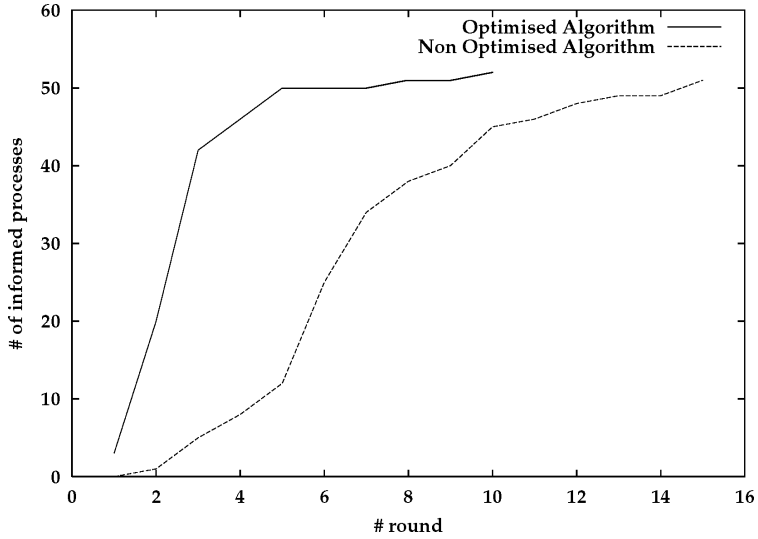


Fig. 15. Propagation delay for membership information for the original and optimized versions of the algorithm.

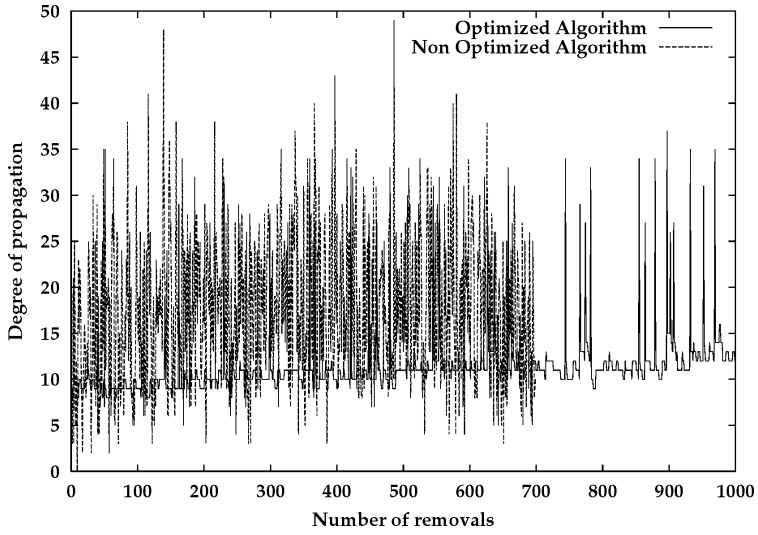


Fig. 16. Measurements: Degree of propagation of removed messages.

that information about new members propagates quickly with the optimized version.

Membership Management. Figure 16 presents the degree of propagation of process Ids that were removed from *subs* buffer with two versions of the algorithm. In the figure the y-axis represents the degree of propagation (i.e., the number of times a process heard about another process, which is equal to the value of *Frequency*) and the x-axis represents the number of removals

we considered. To make the difference clear, plotting was continued for the optimized version for 1000 removals while it was stopped after 700 removals for the other. This helps one to see the difference clearly. From Figure 16 it can be seen that in the non-optimized version process Ids are purged even when the degree of propagation is less than 5. Such a scenario is not experienced in the optimized scheme since it stops purging process Ids that have a lesser degree of propagation.

As a result, it can be seen that with the original version, members are removed even if the degree of propagation is less. But with the optimized version, if the degree of propagation is less, those members are kept in the *subs* buffer.

As seen in Figure 16, the lesser known membership information has a higher probability to survive in the buffers. As a result, isolation can be avoided. This is because when the information about a member is diminishing in the system, that information has a greater chance to be in the buffers.

In a real system, there would be members (subscribers as well as publishers) joining the system frequently, that is the membership is dynamic. An optimization that lowers the propagation delay of membership information will be very useful for a dynamic system.

8. DISCUSSION

This section discusses our *lpbcast* algorithm with respect to "perfectly" uniform views and compares it to the well-known *pbcast* algorithm [Birman et al. 1999], in particular by combining *pbcast* with our membership approach.

8.1 Towards "Perfect" Views

Simulations performed with artificially generated independent uniform views have shown that there is virtually no dependency between latency of delivery (and thus the degree of reliability) and the size of the individual views. The views obtained in practice with *lpbcast* however appear to not be completely uniform and independent.

One interpretation of the slight dependency between latency and l is that, despite the random truncating of views, there remains a correlation between individual views both in time ($view_i$ of process p_i at round r depends on $view_i$ at round $r-1$) and in space ($view_i$ of process p_i depends on $view_j$ of process p_j).

To avoid this effect, we have tried in a first attempt to reduce the frequency of the membership information gossiping (every k-th round only, $k > 1$). It has however turned out that this leads to the opposite effect: latency increases (and thus reliability decreases) further. In contrast, when the frequency for membership gossiping is increased (gossiping membership information more often than events), the views appear to come closer to ideal views, and the performance of our algorithm improves. This is however difficult to apply as an optimization, since T is usually already chosen to be very small to ensure a high throughput.

The precise analysis of the view distribution based on Markov chains seems intractable. However, under the assumption that $l \ll |subs|_m F$, the distribution

can be safely approximated as independently uniform. In making our assumptions, we basically rely upon the results of simulations held for different schemes of the initial view distribution. In particular, we considered a scheme in which one process is known initially to all ("star" topology) and a scheme in which each process is initially known to just two neighbors ("ring" topology). In both cases, the system eventually converges to the "uniform independent" scheme. Defining the exact relationship between the parameters of the algorithm that guarantees that, eventually, the view distribution can be regarded as uniform is an interesting research question.

8.2 Combining *lpbcast* Membership with *pbcast*

In this section we explore how some of the features of *lpbcast* can be combined with such similar algorithms as *pbcast*.

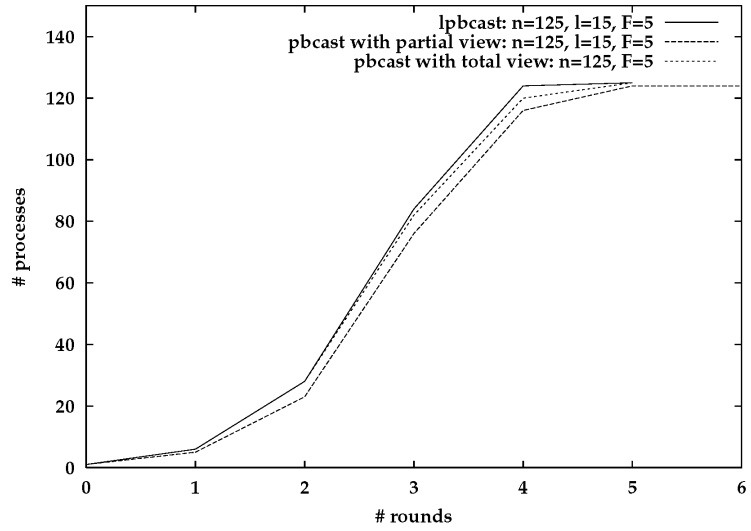
Aside from the memory management schemes (for membership and messages), the main difference between our *lpbcast* algorithm and *pbcast* [Birman et al. 1999] is that our approach melts the two phases of *pbcast* (dissemination of events and exchange of digests) into a single phase. We comment here on the integration of our membership approach with *pbcast*, and compare the resulting algorithm with our *lpbcast* algorithm.

Membership layer. We have presented our membership approach as an integral part of our *lpbcast* algorithm to ease presentation. As we mentioned earlier, our membership approach could be encapsulated as a membership layer, on top of many probabilistic broadcast algorithms, like *pbcast*. The layer would act by adding membership information to gossip messages, and would provide *quasi-independent* uniformly distributed views. Since probabilistic broadcast algorithms require a random subset of the system, theoretically the size of the view does not impact the probability of infection. Hence throughput and delivery latency of the broadcast algorithm would remain virtually unaffected.

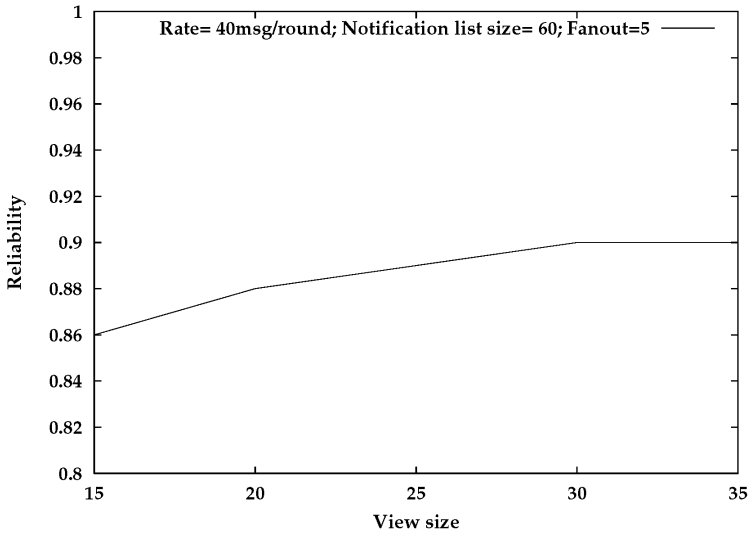
Evaluation. We simulated the behaviour of a *pbcast* version instrumented with our membership approach. Figure 17(a) illustrates the process of event propagation with a partial view membership for *pbcast* and *lpbcast*, comparing it with the case of the original *pbcast* based on a complete view.

Figure 17(b) presents the reliability degree measured with different values for l (in every round, each of $n = 125$ processes published 40 events). The results are similar to the ones obtained with *lpbcast* (Figure 6). A direct comparison of the two algorithms is however not a useful measure, since there are different parameters involved. In fact, because repetitions and hops are limited in the case of *pbcast*, a higher fanout is required to obtain similar results than with *lpbcast* ($F = 5$ here vs $F = 3$ in Figure 6). In fact, *lpbcast* reaches a higher reliability degree when simulated in the same setting, since its latency is smaller.

In practice, and at a high load of the system however, performance can be expected to drop faster with *lpbcast*, since the first phase of (complete) *pbcast* ensures a high throughput, while gossip messages in *lpbcast* will transport a large number of event notifications, which might become a bottleneck.



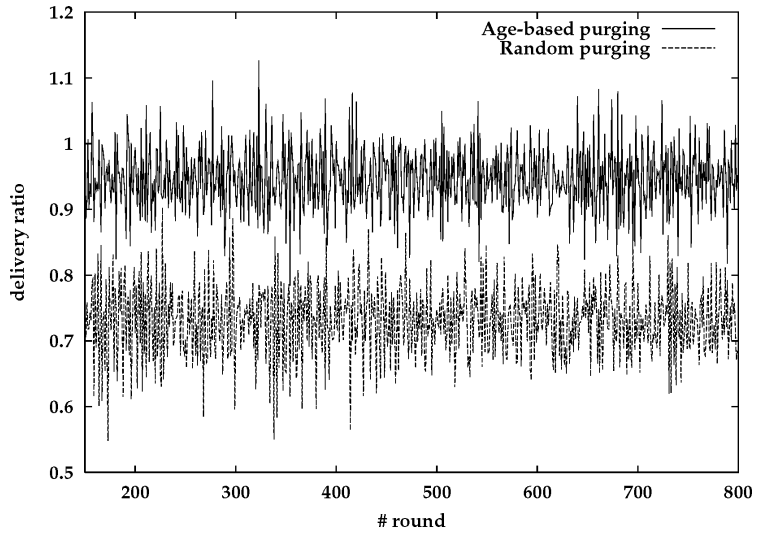
(a) Comparison: number of infected processes in a given round.



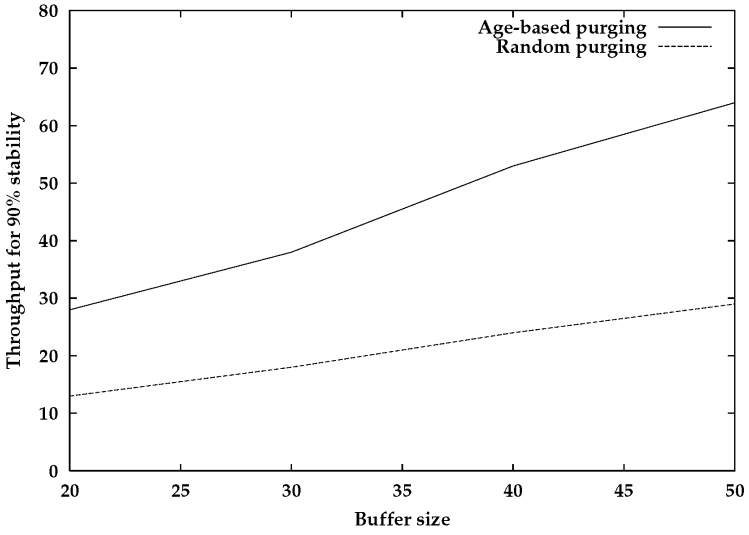
(b) Delivery reliability of *pbcast* with a random partial view.

Fig. 17. Simulations and measurements with *pbcast*.

The age-based message purging scheme, which was discussed in Section 6, is also applicable to *pbcast*. The two versions of pbcast with random message purging (using the original algorithm of *pbcast*), and age-based message purging, were used to show this. The simulation results of these two versions are depicted in Figure 18(a) and Figure 18(b). The age-based message purging performance of the *pbcast* algorithm is clearly higher.



(a) Delivery ratio of pbcast with random purging and age-based purging.



(b) Throughput of pbcast with random purging and age-based purging (message stability level 90%).

Fig. 18. Performance improvement of pbcast by age-based message purging.

The optimization techniques presented in Sections 6 and 7 improve the algorithm without compromising scalability or reliability. These optimization techniques do not reduce the number of messages gossiped. In gossip-based algorithms, messages are gossiped more than once: while some messages

are gossiped many times, others are gossiped comparatively less. That is, there is a high variance. Our approach reduces this variance to maximise the utilization of memory for message storage as well as the bandwidth for message transportation. These optimization techniques can also be applied together with other gossip-based approaches such as the schemes described in Rodrigues et al. [2003].

In WANs. *Lpbcast* is an application level broadcast scheme that does not depend on network level functions and can be easily deployed in WANs. In other terms, it can perform broadcasting in a true peer-to-peer model. In fact because of the highly scalable membership scheme it is better suited to environments like WANs with a huge number of participants.

Lpbcast does not however recognize the "locality" in the network, for example when retrieving missed messages or gossiping, but can be combined with other schemes such as those in Xiao et al. [2002] and Xiao and Birman [2001], which exploit the "locality".

A couple of algorithms were introduced to reduce the memory requirement for buffers [Xiao et al. 2002; Xiao and Birman 2001]. These are especially useful in WANs and arrange receivers into a hierarchical structure of regions. They are built on top of Birman et al. [1999], which is similar to *lpbcast*. As a result these algorithms [Xiao et al. 2002; Xiao and Birman 2001] can make better use of the advantages provided by *lpbcast* in terms of membership management and message purging than Birman et al. [1999].

9. CONCLUDING REMARKS

Probabilistic broadcast algorithms have become very attractive for large scale information dissemination because of their nice combination of scalability and reliability. They seem to constitute ideal candidates to support emerging peer-to-peer applications. Though the reliability guarantees they offer are weaker than traditional ones ([Hadzilacos and Toueg 1993]), their degree of reliability is satisfactory in a practical context. In return, probabilistic broadcast algorithms excel in terms of scalability. Probabilistic broadcast algorithms are scalable because each process sends only a *fixed* number of messages; they achieve fault-tolerance because a process receives copies of a message from *several* processes. However, as we pointed out, the problem of memory management has been neglected, in particular membership management and message purging issues.

This paper addresses precisely the problem of scalable memory management in a probabilistic broadcast algorithm. We present an algorithm called *lpbcast* that is completely decentralized. In our algorithm the membership is handled in a probabilistic manner: a process only knows a *fixed* number of processes obtained randomly, and fault-tolerance can be preserved if each process is known by *several* processes. This idea is intuitively supported by the fact that gossip messages are only sent to a fixed number of processes. Besides the scalability properties of our *lpbcast* algorithm, we have shown that, in practice, there is very little dependency between its reliability and the size of the views, and this view size can be very small compared to the total size of the system.

Purging messages from buffers is also an important issue in probabilistic broadcast. Messages that have been disseminated "enough" in the system (i.e., stable messages) should be removed from the buffers while keeping the others. Due to the decentralized nature of the system, it is non trivial to detect the stability of the messages. In this paper we also presented some optimization techniques that improve the message purging. A similar optimization is also shown that further improves the membership management.

ACKNOWLEDGMENTS

We are very grateful to Ken Birman and Robert van Renesse for affording us an insight into probabilistic reliable broadcast. We would also like to thank the reviewers for their helpful comments on an earlier revision of this manuscript.

REFERENCES

AGUILERA, M., STROM, R., STURMAN, D., ASTLEY, M., AND CHANDRA, T. 1999. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*.

BAILEY, N. 1975. *The Mathematical Theory of Infectious Diseases and its Applications (second edition)*. Hafner Press.

BIRMAN, K., HAYDEN, M., OZKASAP, O., XIAO, Z., BUDIU, M., AND MINSKY, Y. 1999. Bimodal multicast. *ACM Trans. Comput. Syst. 17*, 2 (May), 41–88.

CARZANIGA, A., ROSENBLUM, D., AND WOLF, A. 2000. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC 2000)*. 219–227.

DEERING, S. 1994. Internet multicasting. In *ARPA HPCC 94 Symposium*. Advanced Research Projects Agency Computing Systems Technology Office.

DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., AND TERRY, D. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*. 1–12.

EUGSTER, P. T., FELBER, P., GUERRAOUI, R., AND KERMARREC, A.-M. 2001a. The many faces of publish/subscribe. Tech. Rep. DSC/2001/004, Swiss Federal Institute of Technology, Lausanne, http://dscwww.epfl.ch/EN/publications/.

EUGSTER, P. T., GUERRAOUI, R., AND SVENTEK, J. 2000. Distributed Asynchronous Collections: Abstractions for publish/subscribe interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*. 252–276.

EUGSTER, P. T., GUERRAOUI, R., HANDURUKANDE, S. B., KERMARREC, A.-M., AND KOUZNETSOV, P. 2001b. Lightweight probabilistic broadcast. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2001)*.

EUGSTER, P. T., GUERRAOUI, R., KERMARREC, A.-M., AND MASSOULIE, L. 2003. From epidemics to distributed computing. *IEEE Comput.*

GOLDING, R. 1992. Weak consistency group communication for wide-area systems. In *Proceedings of the Second Workshop on the Management of Replicated Data*.

HADZILACOS, V. AND TOUEG, S. 1993. *Distributed Systems*, 2nd ed. Addison-Wesley, Chapter 5: Fault-Tolerant Broadcasts and Related Problems, 97–145.

KERMARREC, A.-M., MASSOULIÉ, L., AND GANESH, A. 2003. Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. Parallel Distrib. Syst. 14*, 3 (March).

KOUZNETSOV, P., GUERRAOUI, R., HANDURUKANDE, S. B., AND KERMARREC, A.-M. 2001. Reducing noise in gossip-based reliable broadcast. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS 2001)*.

LIN, M.-J. AND MARZULLO, K. 1999. Directional gossip: Gossip in a wide area network. In *European Dependable Computing Conference (EDCC)*. 364–379.

LIN, M.-J., MARZULLO, K., AND MASINI, S. 2000. Gossip versus deterministically constrained flooding on small networks. In *Proceedings of the International Conference on Distributed Computing (DISC 2000)*. 253–267.

ORLANDO, J., RODRIGUES, L., AND OLIVEIRA, R. 2000. Semantically reliable multicast protocols. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS 2000)*.

PAUL, S., SABNANI, K., LIN, J., AND BHATTACHARYYA, S. 1997. Reliable multicast transport protocol (RMTP). *IEEE J. Selected Areas Comm. 15*, 3 (Apr.), 407–421.

PIANTONI, R. AND STANCESCU, C. 1997. Implementing the Swiss exchange trading system. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS '97)*. 309–313.

RODRIGUES, L., HANDURUKANDE, S., PEREIRA, J., GUERRAOUI, R., AND KERMARREC, A.-M. 2003. Adaptive gossip-based broadcast. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2003)*.

SUN, Q. AND STURMAN, D. 2000. A gossip-based reliable multicast for large-scale high-throughput applications. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN2000)*. New York, USA.

TIBCO. 1999. *TIB/Rendezvous White Paper*. http://www.rv.tibco.com/.

VAN RENESSE, R. 2000. Scalable and secure resource location. In *Proceedings of the IEEE Hawaii International Conference on System Sciences*.

XIAO, Z. AND BIRMAN, K. 2001. Randomized error recovery algorithm for reliable multicast. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*.

XIAO, Z., BIRMAN, K., AND VAN RENESSE, R. 2002. Optimizing buffer management for reliable multicast. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN2002)*.

Received August 2001; revised January 2003; accepted February 2003