Lightweight Probabilistic Broadcast

P. Th. Eugster[1] R. Guerraoui[1] S. B. Handurukande[1] A.-M. Kermarrec[2] P. Kouznetsov[1]
[1] Federal Institute of Technology, Lausanne, Switzerland
[2] Microsoft Research, Cambridge, UK

Abstract

The growing interest in peer-to-peer applications has underlined the importance of scalability in modern distributed systems. Not surprisingly, much research effort has been invested in gossip-based broadcast protocols. These trade the traditional strong reliability guarantees against very good "scalability" properties. Scalability is in that context usually expressed in terms of throughput and delivery latency, but there is only little work on how to reduce the overhead of membership management at large scale.

This paper presents Lightweight Probabilistic Broadcast (lpbcast), a novel gossip-based broadcast algorithm which preserves the inherent throughput scalability of traditional gossip-based algorithms and adds a notion of membership management scalability: every process only knows a random subset of fixed size of the processes in the system. We formally analyze our broadcast algorithm in terms of scalability with respect to the size of individual views, and compare the analytical results both with simulations and concrete measurements.

1. Introduction

Large scale event dissemination. *Peer-to-peer* computing has recently received much attention, as shown by the success of large scale decentralized applications like *Gnutella* [22] or *Groove* [10]. In *peer-to-peer* computing, every process acts as client and server, and scalability is a major concern.

The scalability properties solicited from such applications have evolved from hundreds to thousands of participants, but adequate algorithms for reliable propagation of events at large scale are still lacking. Network-level protocols have turned out to be insufficient: *IP multicast* [6] lacks reliability guarantees, and reliable protocols do not scale well. The well-known *Reliable Multicast Transport Protocol* (RMTP) [17] for instance generates a flood of positive acknowledgements from receivers, loading both the network and the sender. Any form of *membership* [2, 12] is hidden by such network-level protocols, which makes them difficult to exploit with more dynamic dissemination (*filtering*, e.g., [15]), emphasizing the need for new forms of application-level broadcast.

Gossip-based broadcast algorithms. *Gossip*-based broadcast algorithms [4, 14, 19] appear to be more adequate in the field of large scale event dissemination, than the "classical" strongly reliable approaches [11]. Though such gossip-based approaches have proven good scalability characteristics in terms of throughput, 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 *memory resources*, let aside the *communication* required to ensure the consistency of the membership.

Partial view. Message routing and membership management are sometimes delegated to dedicated servers in order to relief application processes [1, 5, 20]. This only defers the problem, since those servers are limited in resources as well. To further increase scalability, the membership view should be *split*, i.e., every participating process should only have a *partial* view of 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.

Gossip-based membership. While certain systems rely on a deterministic scheme to establish the individual views [14, 21], we introduce here a new *completely randomized* approach. The local view of every individual member consists of a random process list which continuously evolves, but never exceeds a fixed size. In short, after adding new processes to a view, it is truncated to the maximum length by removing randomly chosen entries. To ensure 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 views. The membership protocol and the effective dissemination of events are thus dealt with at the same level.

Contributions. We present in this paper our strongly scalable decentralized algorithm for event dissemination, called *lpbcast*, which we have used to implement a static publish/subscribe. We convey our claim of scalability in two steps. First, we formally 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 delivery of an event. We similarly show that for a given view size, the probability of partition creation in the system decreases as the system grows in size. Second, we give some practical results that support the analytical approach, both in terms of simulation and prototype measurements.

It is important to notice that our membership approach is not intrinsically tied to our *Lightweight Probabilistic Broadcast (lpbcast)* algorithm. We illustrate this by applying our membership scheme to the well-known *pbcast* [4] algorithm.

Roadmap. Section 2 gives an overview of related gossip-based broadcast protocols. Section 3 presents our *lpbcast* algorithm and explains our randomized approach. Section 4 presents a formal analysis of our algorithm in terms of scalability and reliability. Section 5 gives some simulation and practical results supporting the formal analysis. Section 6 discusses the distribution of the views and also proves the general applicability of our membership approach by combining it with *pbcast* and contrasting the consolidated algorithm with *lpbcast*.

2. Background: Probabilistic Algorithms

The achievement of strong reliability guarantees (in the sense of [11]) in practical distributed systems requires expensive mechanisms to detect missing messages and initiate retransmissions. Due to the overhead of message loss detection and reparation, protocols offering such strong guarantees do not scale over a couple of hundred processes [18].

2.1. Reliability vs Scalability

Gossip, or *rumor mongering* algorithms [7], are a class of *epidemiologic* algorithms, which have been introduced as an alternative to such "traditional" reliable broadcast protocols. They have first been developed for replicated database consistency management [7]. The main motivation is to trade the reliability guarantees offered by costly deterministic protocols against weaker reliability guarantees, but in return obtain very good scalability properties.

Their analysis is usually based on stochastics similar to the theory of epidemics [3], where the execution is broken down in steps. Probabilities are associated to these steps, and such algorithms are therefore sometimes also referred to as *probabilistic* algorithms. The degree of reliability is typically expressed by a probability; like the probability 1-α of reaching all processes in the system for any given message, or by a probability 1-β of reaching any given process with any given message. Ideally, α and β are precisely quantifiable.

2.2. Basic Concepts

Decentralization is the key concept underlying the scalability properties of gossip-based broadcast algorithms, i.e., the overall load of retransmissions is reduced by decentralizing the effort. More precisely, retransmissions are initiated in most gossip-based 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). Gossip protocols differ in the number of times the same information is gossiped, i.e., every process might gossip the same information only a limited number of times (*repetitions* are limited) and/or the same information might be forwarded only a limited number of times (*hops* are limited).

2.3. Membership Tracking in Gossip-Based Algorithms

Membership tracking in gossip-based algorithms is a challenging issue. Early approaches like [9] admit that the individual views of processes diverge temporarily, but assume that they eventually converge in "stable" phases. These views however represent the "complete" membership, which becomes a bottleneck at an increased scale. The *Bimodal Multicast* [4] and *Directional Gossip* [14] algorithms are representatives of a new generation of probabilistic algorithms – aware of the problem of scalable membership management.

Bimodal Multicast. *Bimodal Multicast* (also called *pbcast*) relies on two phases. A "classical" best-effort multicast protocol (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 membership problem is not dealt with in [4], but the authors refer to another paper which describes *Capt'n Cook* [21], a gossip-based resource location protocol for the Internet, which can in that sense be seen as a membership protocol. This protocol enables the reduction of the view of each individual process: each process has a precise view of its immediate neighbours, while the knowledge becomes less exhaustive at increasing "distance". The notion of distance is expressed in terms of host addresses. Capt'n Cook however only considers the propagation of membership information and it is thus not clear how this membership interacts with *pbcast*.

Directional Gossip. *Directional Gossip* is a protocol especially targeted at wide area networks. By taking into account the topology of the networks and the current processes, optimizations are performed. More precisely, a *weight* is computed for each neighbour node, representing the connectivity of that given node. The larger the weight of a node, the more possibilities exist thus for it to be infected by any node. The protocol applies a simple heuristic, which consists in choosing nodes with higher weights with a smaller probability than nodes 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 which 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.

In contrast to the deterministic hierarchical membership approaches in Directional Gossip or Capt'n Cook, our *lpbcast* algorithm has a probabilistic approach to membership: each process has a *random* 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 to disseminate notifications and to propagate digests of received events, but also to propagate membership information.

3. Lightweight Probabilistic Broadcast *(lpbcast)*

In this section, we present our completely decentralized lightweight probabilistic algorithm for event dissemination based on partial views. Though the parts concerning the event dissemination and the membership respectively can be considered as independent, we present our solution 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.

3.1. System Model

We consider a system of processes $\Pi = \{p_1, p_2, ...\}$. Processes join and leave the system dynamically and have ordered distinct identifiers. We assume for presentation simplicity that there is not more than one process per node of the network.

Though our algorithm has been implemented in the context of topic-based publish/subscribe [8], 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.

3.2. Gossip Messages

Our *lpbcast* algorithm is based on non-synchronized periodical gossips, where a gossip message contains several types of information. To be more precise, a gossip message serves four purposes:

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

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

Unsubscriptions: A gossip message also piggybacks a set of unsubscriptions. This type of information enables the gradual removal of processes which have unsubscribed from local views. Unsubscriptions that are eligible to be forwarded with the next gossip(s) are stored in a variable *unSubs*.

Subscriptions: A set of subscriptions are attached to each message. These subscriptions are buffered in *subs*. A gossip receiver uses these subscriptions to update its view, stored in a variable *view*.

Note that none of the outlined data structures contains 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* ($|view|_m$) will be denoted l.

3.3. Procedures

The algorithm is composed of two parts. The first part is executed upon reception of a gossip message, and the second part is repeated periodically in 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 local view (*view*), and then added to the list of potentially forwarded unsubscriptions *unSubs*. This list 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 local 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 *subs*, are a random mixture of subscriptions which are present in the view after the execution of this phase, and subscriptions removed to respect the maximum size limit of *view*. Finally, *subs* is also truncated to respect the maximum size limit.

III. The third phase consists in delivering to the application 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 notifications in *eventIds*, as previously outlined. Delivered notifications are at the same time eligible for being forwarded with the next gossip.

Gossip sending. Each process periodically (every $T\ ms$) generates a gossip message – according to Section 3.2 – which it gossips to F other processes, randomly chosen among the local view (*view*) (Figure 1(b)). This is done even if the process has not received any new 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 fluctuations in terms of overall load due to gossip messages, as long as T and the number of processes inside Π and remain unchanged.

```
upon RECEIVE (gossip)
{Phase 1: Update view and unSubs with unsubscriptions}
for all unsub $\in$ gossip.unSubs do
  view $\leftarrow$ view $\setminus$ {unsub}
  unSubs $\leftarrow$ unSubs $\cup$ {unsub}
while $|$unSubs$| > |$unSubs$|_m$ do
  remove random element from unSubs
{Phase 2: Update view with new subscriptions}
for all newSub $\in$ gossip.subs $\wedge$ newSub $\neq$ $p_i$ do
  if newSub $\notin$ view then
    view $\leftarrow$ view $\cup$ newSub
    subs $\leftarrow$ subs $\cup$ newSub
while $|$view$| > l$ do
  target $\leftarrow$ random element in view
  view $\leftarrow$ view $\setminus$ {target}
  subs $\leftarrow$ subs $\cup$ {target}
while $|$subs$| > |$subs$|_m$ do
  remove random element from subs
{Phase 3: Update events with new notifications}
for all e $\in$ gossip.events do
  if e.id $\notin$ eventIds then
    events $\leftarrow$ events $\cup$ {e}
    LPB-DELIVER(e)
    eventIds $\leftarrow$ eventIds $\cup$ {e.id}
for all e.id $\in$ gossip.eventIds do
  if e.id $\notin$ eventIds then
    {Retrieving the notification}
    RETRIEVE(e)
    events $\leftarrow$ events $\cup$ {e}
    LPB-DELIVER(e)
    eventIds $\leftarrow$ eventIds $\cup$ {e.id}
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
gossip.subs $\leftarrow$ subs $\cup$ {$p_i$}
gossip.unSubs $\leftarrow$ unSubs
gossip.events $\leftarrow$ events
gossip.eventIds $\leftarrow$ eventIds
choose $F$ random members target$_1$, ... target$_F$ in view
for all $j \in [1..F]$ do
  SEND(target$_j$, gossip)
events $\leftarrow \emptyset$

upon LPB-CAST(e)
events $\leftarrow$ events $\cup$ {e}
```

(b) Gossip emission

Figure 1. *lpbcast* algorithm

3.4. Subscribing and Unsubscribing

For presentation simplicity we have not reported the procedures for subscribing/unsubscribing in Figure 1(a). In short, a process p_i which wants to subscribe must know a process p_j which 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, 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 local 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. It is important to notice that this scheme is not applied to subscriptions: these are continuously dispatched in order to ensure uniformly distributed views.

Due to the evolving nature of the membership scheme, failed processes are gradually removed from all the views in the system.

4. Analytical Evaluation

This section presents a formal analysis of our *lpbcast* algorithm. The goal is to show the impact of the size l of the individual views of processes both (1) on the latency of delivery and (2) on the stability of our membership. The analysis differs from the one proposed in [4], precisely because our membership is not global and a same notification id is not forwarded only a limited number of times (hops are not limited), and can be forwarded several times by the same process (repetitions are not limited). We first introduce a set of assumptions without which the analysis becomes extremely tedious, but which have only 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). According to the terminology applied in epidemiology, a process which has delivered a given 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.

At each round, each process has an *independent uniformly distributed* random view of size l of known subscribers. In other terms, every combination of l 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 simplicity reasons, we will also refer to such views as *uniform views* (though this is a language abuse). The *expected* number of processes which know a given process is thus $\approx l$. These views are not constant, but continue 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 injected 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 which 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 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:

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

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, but conveys also the fact that the gain is not proportional. In fact, with a too high fanout, there will be more redundant messages received by each process, which limits performance (and the network would also drop more). 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 [13]

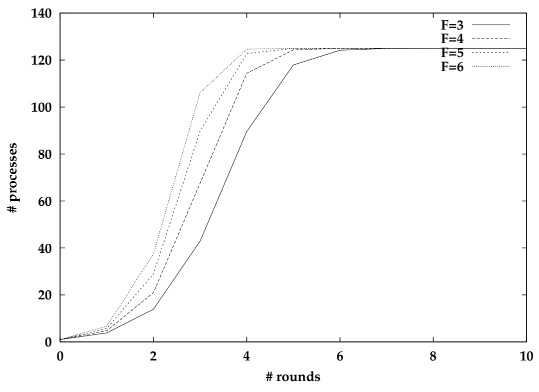


Figure 2. Analysis: expected number of infected processes for a given round with different fanout values

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 [3].

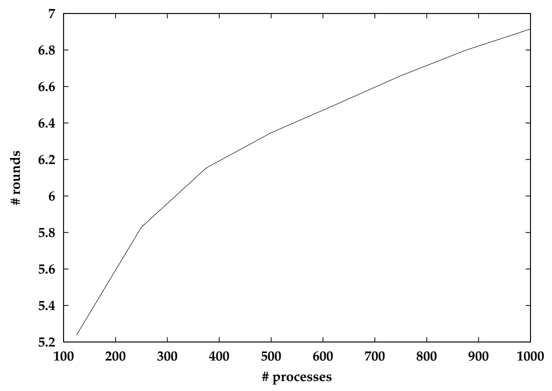


Figure 3. Analysis: expected number of rounds necessary to infect 99% of Π, given system size n

View size l. According to Equation 2, the view size l does not impact the time it takes for a 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 protocol 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. The algorithm shown in Figure 1(b) intuitively supports this hypothesis by two properties, namely (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 more in detail in Sections 5 and 6.

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

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

It can easily be shown that $\Psi(i, n, l)$ monotonically decreases when increasing n or l. 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.

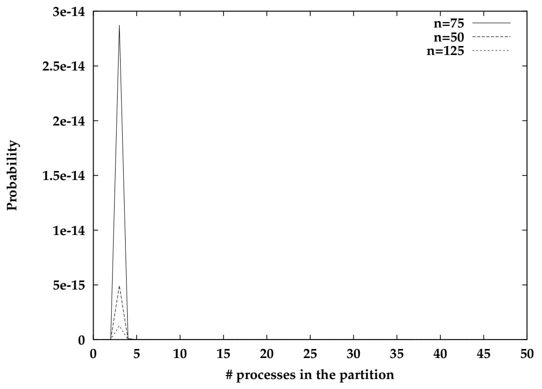


Figure 4. Analysis: probability of partitioning in systems of different sizes

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:

$$\begin{aligned} \phi(n, l, r) &= \left(1 - \sum_{l+1 \leq i \leq \lfloor n/2 \rfloor} \Psi(i, n, l)\right)^r \\ &\approx 1 - r \cdot \sum_{l+1 \leq i \leq \lfloor n/2 \rfloor} \Psi(i, n, l) \end{aligned} \quad (5)$$

This probability decreases very slowly with r. It takes $\approx 10^{12}$ rounds to end up with a partitioned system with a probability of 0.9 with $n = 50$ and $l = 3$.

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 (and also for bootstrapping). Alternatively, we could use a set of dedicated processes to collaborate in keeping track of the total number of processes.

5. Practical 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 a notification, the 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 [16]).

5.1. Simulation

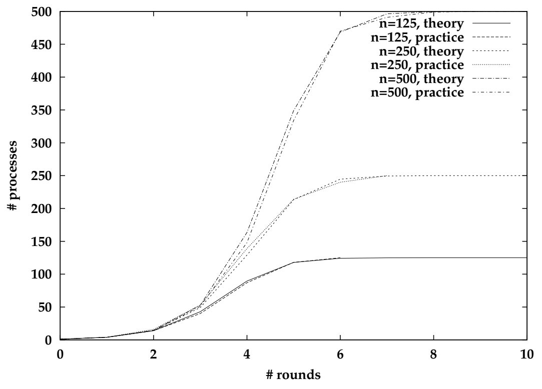
In a first attempt we have simulated the entire system in a single machine. More precisely, we have simulated synchronous gossip rounds in which each process gossips once. The results obtained from these simulations support the validity of our analysis.

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.

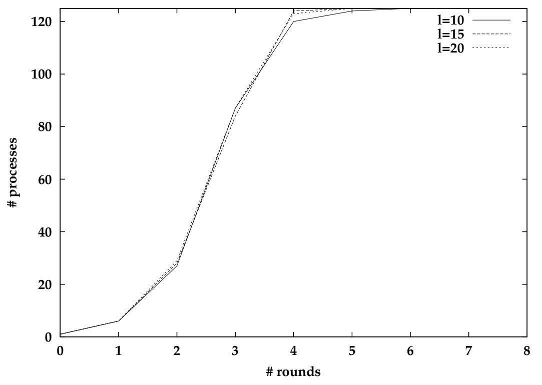
Impact of l. According to the analysis presented in the previous section, the size l of the individual views on the other hand has no impact on the number of gossip rounds it takes to infect every process in the system. Figure 5(b) reports the simulation results obtained for different values for l in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



(a) Analysis vs simulation



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

Figure 5. Simulation results

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 (100Mbit/s). The measurements we present here were obtained with all 125 processes; each publishing 40 events per gossip round. To conform to our simulations, F was fixed to 3.

Impact of a 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 to deliver any given notification ($1-\beta$, cf. Section 2). The reliability of the system seems to deteriorate slightly with a decreasing value for l. Intuitively this seems understandable, since our simulation results have already shown that latency *does* increase slightly by decreasing l. And with an increased latency, the probability that a given message is purged from all buffers before all processes have been infected becomes higher.

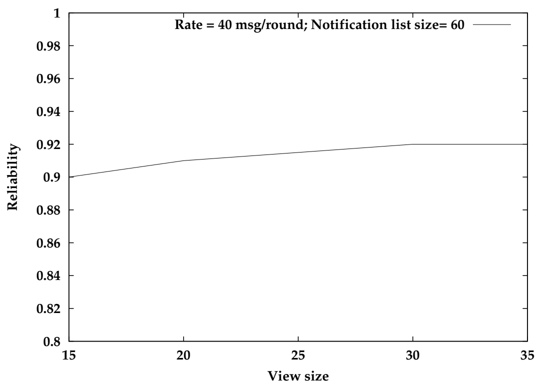


Figure 6. Measurements: degree of reliability

6. Discussion

This section discusses our *lpbcast* algorithm with respect to "perfectly" uniform views and compares it closer with the well-known *pbcast* algorithm [4], in particular by combining *pbcast* with our membership approach.

6.1. 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* thus appear to not be completely uniform and independent.

Dependency. 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). Intuitively, such dependencies negatively affect the latency of delivery of an event: since every process appends part of its *view* to each outgoing gossip message, a process p_i, which receives a gossip message from p_j, has a certain probability of gossiping to processes that have received the same gossip message from p_j (p_i updates its *view* according to the *subs* it received from p_j, possibly including processes which have been gossiped to in the same round by p_j).

To avoid this effect, we have tried in a first attempt to reduce the frequency for the gossiping of membership information (every k-th round only, $k > 1$). It has however turned out that this sanction leads to the opposite effect, i.e., latency increases (and thus reliability decreases) further. In contrast, when the frequency for membership gossiping is increased (gossiping membership information more often that 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 chosen already very small to ensure a high throughput.

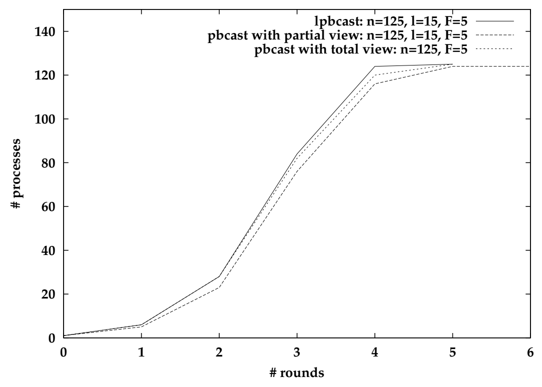
Weighted views. As already mentioned, every process should ideally be known by *exactly* l other processes. This is however difficult to ensure without relying on any form of agreement or counting. We propose an optimization to get the distribution of views closer to this ideal case. It consists in adding a *weight* to every entry in the view, which gives a measure about how well a given process is known. Unlike the weights used in Directional Gossip [14] however, which represent the connectivity of processes, weights in our case represent the *level of awareness* for a given process.

When a process p_i learns about another process p_j which is in p_i's view through the *subs* attached to an incoming gossip message, the weight of p_j is increased. When truncating the *view*, a simple heuristic is applied, consisting in removing entries with a high weight, since these are more probable of being known by many other processes. Furthermore, when constructing *subs*, a process preferably adds entries from its *view* with a small weight. A similar scheme could also be applied to *events* and *eventIds*.

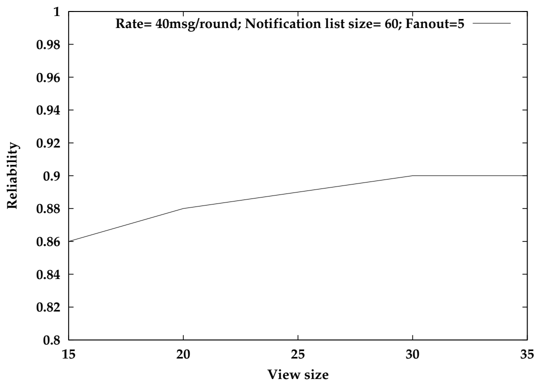
6.2. Comparison with *pbcast*

Aside from the membership scheme, the main differences between our *lpbcast* algorithm and *pbcast* are (1) that the latter algorithm limits the number of hops as well as (2) repetitions for a given message, and (3) 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 *pbcast* with our membership approach, and compare it with our *lpbcast* algorithm.

Membership layer. We have presented our membership approach as integral part of our *lpbcast* algorithm to ease presentation. As we have mentioned earlier, our membership approach is nevertheless not inherently coupled with our *lpbcast* algorithm, but can be separated from the event dissemination process. It could thus be encapsulated as a membership layer, on top of which many gossip-based algorithms, like *pbcast*, could be deployed. It would act by adding membership information to gossip messages, and would provide *quasi-independent* uniformly distributed views. Since gossip-based protocols require a random subset of the system, theoretically the size of the view does not impact the probability of infection and hence throughput and delivery latency of the broadcast algorithm would remain virtually unaffected.



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



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

Figure 7. Simulations and measurements with *pbcast*

Evaluation. We simulated the behaviour of a *pbcast* version instrumented with our membership approach. Figure 7(a) illustrates the process of an event propagation in such a partial view membership for *pbcast* and *lpbcast*, comparing with the original *pbcast* based on a complete view. The advantage of our *lpbcast* over *pbcast* can be explained by the fact that hops and repetitions are not limited with the former algorithm.

Figure 7(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 *pbcast* ensures a high throughput, while gossip messages in *lpbcast* will transport large numbers of notifications, which might become a bottleneck.

Acknowledgements

We are very grateful to Ken Birman and Robert van Renesse for affording us an insight into the subtle approach of probabilistic reliable broadcast.

References

[1] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, Nov. 1999.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Mahlki. Membership algoritms for multicast communication groups. In *6th Intl. Workshop on Distributed Algorithms proceedings (WDAG)*, pages 292–312, Nov. 1992.

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

[4] K. Birman, M. Hayden, O.Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[5] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Dec. 1998.

[6] S. Deering. Internet multicasting. In *ARPA HPCC 94 Symposium*. Advanced Research Projects Agency Computing Systems Technology Office, Mar. 1994.

[7] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Aug. 1987.

[8] P. Eugster, R. Guerraoui, and J. Sventek. Distributed Asynchronous Collections: Abstractions for publish/subscribe interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 252–276, June 2000.

[9] R. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California at Santa Cruz, Dec. 1992.

[10] Groove Networks, *Introducing Groove*. http://www.groovenetworks.com/, 2000.

[11] V. Hadzilacos and S. Toueg. *Distributed Systems*, chapter 5: Fault-Tolerant Broadcasts and Related Problems, pages 97–145. Addison-Wesley, 2nd edition, 1993.

[12] M. Hiltunen and R. Schlichting. Properties of membership. In *Proceedings of the 2nd IEEE Symposium on Autonomous Decentralized Systems*, pages 200–207, Apr. 1995.

[13] A.-M. Kermarrec, L. Massoulie, and A. Ganesh. Reliable probabilistic communication in large-scale information dissemination systems. Technical Report MSR-TR-2000-105, Microsoft Research Cambridge, Oct. 2000.

[14] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. Technical Report CS1999-0622, University of California, San Diego, Computer Science and Engineering, June 1999.

[15] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP Multicast in content-based publish-subscribe systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*, pages 185–207, Apr. 2000.

[16] J. Orlando, L. Rodrigues, and R. Oliveira. Semantically reliable multicast protocols. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS 2000)*, Oct. 2000.

[17] S. Paul, K. Sabnani, J. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, Apr. 1997.

[18] R. Piantoni and C. Stancescu. Implementing the swiss exchange trading system. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS '97)*, pages 309–313, June 1997.

[19] Q. Sun and D. Sturman. 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, July 2000.

[20] TIBCO. *TIB/Rendezvous White Paper*. http://www.rv.tibco.com/, 1999.

[21] R. van Renesse. Scalable and secure resource location. In *Proceedings of the IEEE Hawaii International Conference on System Sciences*, 2000.

[22] Wego.com Inc., *What Is Gnutella?* http://gnutella.wego.com/, 2000.