Atomic Probabilistic Broadcast

Patrick Th. Eugster
Sun Microsystems*

Abstract

Many applications, such as the dissemination of stock quote events, require reliable and totally ordered delivery of broadcast messages to a large number of processes. Recently, gossip-based broadcast algorithms have been praised as an interesting alternative to deterministic broadcast algorithms for such reliable information propagation, by providing an appealing tradeoff between reliability *and* scalability.

However, despite the importance of ordering guarantees, only little work exists on gossip-based broadcast algorithms providing guarantees stronger than (probabilistic) reliability. In fact, it is still not clear how (well) a probabilistic approach marries with such ordering guarantees.

In this paper, we present a novel algorithm, called Atomic Probabilistic Broadcast, for totally ordered delivery of broadcast messages to members of large groups. This algorithm is hybrid in the sense that it has both probabilistic and deterministic characteristics: while the propagation of broadcast messages and ordering information is probabilistic, this ordering information is computed deterministically in a decentralized manner, thereby providing availability *and* consistency. *We point out the advantages of such a hybrid approach, in terms of reliability, scalability, and consistency, and convey these through both analysis and simulation.*

1 Introduction

The success of publish/subscribe systems reflects the need of modern applications for group communication primitives deployable at large scale. Yet, group communication algorithms aiming at strong guarantees in the face of communication and process failures hardly scale. This statement is increasingly valid when considering algorithms additionally aiming at ordered, e.g., totally orderered, delivery of messages issued concurrently. Yet, many applications, such as the dissemination of stock quotes to a large number of stock brokers, typically require such guarantees on delivery order.

Atomic Broadcast. The "classic" solution of [18] to Atomic Broadcast (a.k.a. Total Order Broadcast) can be described as completely (1) *deterministic* and (2) *symmetric* (decentralized) with respect to both the propagation of broadcast messages, and ordering of these messages. The algorithm tolerates $\lfloor \frac{n-1}{2} \rfloor$ process failures within a group of n processes, but involves communication rounds of $O(n^2)$ messages for broadcasting messages, and furthermore requires each process to know every other process, leading to a membership knowledge of $O(n)$ for each process.

Asymmetric approaches. Literature is very rich in further solutions to the problem of Atomic Broacast (cf. [11]). A major principle in improving over the above-described approach consists in

*Former affiliation: Swiss Federal Institute of Technology in Lausanne, patrick.eugster@epfl.ch.

separating the propagation of messages from the computation of the delivery order for these messages. This aids in reducing the overhead of latter task — a predominant factor to the latency of message delivery [24].
This introduces a large family of algorithms, which can be characterized as (1) *deterministic* but (2) *asymmetric*. Typical representatives are algorithms relying on sequencers/tokens for determining the delivery order of messages (cf. [24]). Whilst incurring a reduced overhead for message ordering (possibly also by making assumptions on broadcasters, e.g., number, disposition), such algorithms naturally provide a lower degree of reliability (i.e., availability) than the approach of [18], as the failure of a sequencer (the loss of a token) does not go unnoticed, but conversely incurs a sensible failover time. Many publish/subscribe systems (e.g., [6],[1, 2]) carry asymmetry even further, by making use of an overlay graph of "message routers" for interconnecting group members much like a spanning tree. Such a tree can be used for propagating messages with low complexity, e.g., $O(n)$ by requiring every (router) process to know only a number of processes that can be upper bound by a constant. This graph can then also be exploited to order messages at "concentration points". However, it is not clear whether such approaches offer reliability (i.e., availability) in the face of (router) process failures, and if they do so (e.g., by replicating routers), whether and how consistency of the computed delivery order is ensured. [1, 2] for instance mentions the use of redundancy in the overlay graph, but the ordering algorithm of [1] does admittedly not cope with failures of processes broadcasting or ordering messages. Motivated by the desire of achieving delivery even despite temporary disconnections of group members, all broadcast messages are logged on stable storage, making persistence the mechanism of choice for reliability. Thus, it appears that the overhead of order determination (and message propagation) is somewhat traded against an overhead of message logging, and in particular, a loss of availability.

Probabilistic approaches. Another recent trend in the design of broadcast algorithms consists in using "gossips" [12] for achieving good scalability (by requiring typically $O(n \ ln \ n)$ messages for broadcasting a message [20]), and a high "degree" of reliability which gracefully degrades in the face of an increasing number of process failures [21].
However, only little work exists on gossip-based broadcast algorithms with more than (probabilistic) reliability guarantees. The seminal Probabilistic Broadcast (*pbcast*) algorithm of Birman et al. is originally described informally with an inherent scheme for achieving total order [3], and later on is presented as a Reliable Broadcast component which can be simply "plugged" into Atomic Broadcast [16]. In both cases, the guarantees achieved for the consistency of the delivery order are not clear.
Probabilistic Atomic Broadcast (*pabcast*) [15] is fully (1) *probabilistic*, and (2) *symmetric*. By mixing message ordering and propagation as in [18], but basing these on gossips, *pabcast* achieves lower complexity than [18] by sacrificing reliability in the sense of both order consistency and delivery reliability. This reflects through individual probabilities associated with its Validity, Agreement, and Total Order guarantees, which however gives the false impression that the respective probabities are independent: while the first two properties are defined with respect to a single message, the third property introduces dependencies between these messages. This manifests in [15] through the absence of an analytical evaluation of the probability associated with the third property. The behavior of *pabcast* is hence hard to grasp, as no information is given on the intersection of the different properties. Furthermore, *pabcast*, just like *pbcast*, imposes a membership knowledge of $O(n)$ on every process.

A hybrid approach. This paper presents Atomic Probabilistic Broadcast (*apbcast*), a novel algorithm implemented for publish/subscribe programming in the DACE platform [10]. *apbcast* is hybrid: its (1a) *deterministic* ordering of messages ensures the consistency of the delivery order of broadcast messages,

and its (1b) *probabilistic*, i.e., gossip-based, propagation of broadcast messages and order information provides a high level of reliability in the face of an increasing number of process failures.
apbcast achieves the best of both worlds, naturally, by relying on an inherent $L-$ recursive subdivision of process groups. This fully (2) *asymmetric* approach inherently provides "concentration points" for ordering broadcast messages. By making use of R such points which deterministically order broadcast messages for each primary subdivision group, an ideal tradeoff is achieved between ordering overhead (greatly keeping traffic "local" to such subgroups) and availability (by tolerating *at least* $\lfloor \frac{B-1}{2} \rfloor$ failures among the B broadcasting processes, *and* $\lfloor \frac{R-1}{2} \rfloor$ failures of ordering processes within *each* subgroup). And this tradeoff can be tuned through R. Furthermore, the dissemination of a broadcast message involves only $O(n\ ln\ n)$ network messages (typical for symmetric gossip-based broadcast), by imposing however only a membership knowledge of $O(n^{1/L})$ on processes.
The consistency of the total order delivery reflects in that Atomic Probabilistic Broadcast only introduces a probabilistic Agreement property (hence the emphasis on "Atomic" in constrast to *pabcast* [15]). This is particularly appealing since processes can detect missed messages, and are thus able to undertake more heroic efforts to recover these (e.g., by querying other processes, or making use of loggers) and deliver them without violating total order.
In this paper, we first provide our Atomic Probabilistic Broadcast specification, and then present our algorithm implementing it. We convey the scalability (e.g., in terms of message and time complexity, membership knowledge) and reliability of our algorithm through both analysis and simulation results, and discuss various ways of tuning the performance of *apbcast*.

Roadmap. The rest of this paper is organized as follows. Section 2 presents assumptions made on the system and the specification of Atomic Probabilistic Broadcast. Section 3 provides an overview of our *apbcast* algorithm. Section 4 presents the weakly consistent membership underlying *apbcast*. Section 5 presents the algorithm for message ordering in *apbcast*. Section 6 presents the broadcast message propagation in *apbcast*. Section 7 proves the correctness of our algorithm. Section 8 analyses our *apbcast* algorithm. Section 9 presents preliminary simulation results. Section 10 discusses various issues, such as tuning mechanisms. Section 11 presents related work in more detail. Section 12 draws final conclusions.

2 Model and Problem

In this section, we first present the assumptions made on the system, and then provide the specification of Atomic Probabilistic Broadcast.

2.1 System Model and Assumptions

We consider an asynchronous (in the sense of [19]) system Π of uniquely identified processes $\{p_1, \ ...\}$ communicating over fair lossy channels.
Processes communicate with two pairs of primitives, namely (1) SEND and RECEIVE, which model unreliable communication associated with a probability $1-\epsilon$ of successful message transmission, and (2)RSEND and RRECEIVE, which model reliable communication, built on top of fair-lossy channels.
A small subset $\Psi = \{b_1, \ ...\} \subset \Pi$ of B processes are *broadcasters*. Each such $b_i \in \Psi$ is characterized by its known average broadcast rate $P[b_i]$. Without loss of generality, we assume that $P[b_1] \leq P[b_2] \leq P[b_3]$....
Increasing the number of broadcasters disproportionately does not invalidate the algorithm, yet, without taking measures (proposed in Section 10.2), may lead to performance degradation. For presentation,

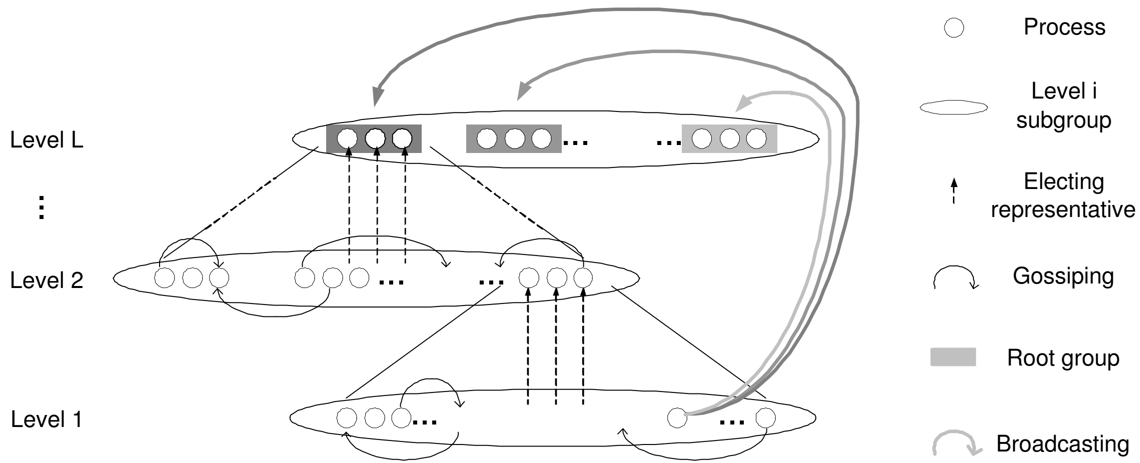


Figure 1: Overview of *apbcast*

we henceforth only more consider a single, large, group of processes, each of which can fail by halting prematurely only.

2.2 Atomic Probabilistic Broadcast

The problem of Atomic Probabilistic Broadcast (*apbcast*) within a group of processes is defined through two primitives APBCAST and APDELIVER as follows:

Integrity For any message m, every correct process APDELIVERs m at most once, and only if m was previously APBCAST by $sender(m)$.

Validity If a correct process p APBCASTs a message m then p eventually APDELIVERs m.

Probabilistic Agreement Let p and q be two correct processes. If p APDELIVERs a message m, then with probability α, q APDELIVERs m.

Global Total Order There exists a unique ordering $\{m_i\}_i$ of all APDELIVERed messages such that if any process APDELIVERs m_i before m_j then $i < j$.

In other terms, the only probabilistic property is Agreement. We refer to α as the *expected reliability degree*. Obviously, any algorithm with simply delivers a broadcast to the broadcaster itself and then stalls implements this specification. The main challenge is hence to achieve a high value for α. Note that this probabilistic notion of agreement also captures a weakly consistent membership, typical for large scale settings.

By associating a probability with the Agreement property, different messages are potentially delivered by different sets of processes. For that reason, the Total Order property known from Atomic Broadcast has been strengthened to the Global Total Order property from Global Atomic Multicast (cf. [19]).

3 Overview of *apbcast*

This section presents the recursive group subdivision underlying our *apbcast* algorithm, and an overview of the different algorithm parts.

3.1 Group Subdivision

With *apbcast*, a process group is subdivided recursively into subgroups. This subdivision is defined by the *addresses* of processes. With this respect, *apbcast* follows a similar approch as [14], but not an identical one (the motivation is slightly different, see Section 10.1).

Definition 1 (Addresses) Every process $p \in \Pi$ has a unique address, of the form $p_L \cdot ... \cdot p_1$ (big endian). $L > 2$ is constant for the entire group, and every address *component* is bound, i.e., $\exists\ a_k,\ k \in [1..L]$, such that $\forall\ p\ \in\ \Pi,\ 0\ \leq p_k\ \leq\ a_{k-1}$.

Hence, addresses are ordered, and the maximum number of addresses, and therethrough processes, in the system is given by $\prod_{k=1}^{L} a_k$. Ideally (but not necessarily), addresses are determined such that they reflect the network topology. This can be achieved by explicitly assigning such addresses to processes, or by approximating topology through some computational method (e.g., [23]).
A process p only knows all of its "immediate" group neighbors, i.e., every process q in the group whose address only diverges in the last position ($q_1 \neq p_1$), but coincides in the other positions ($q_k = p_k \forall k \in [2..L]$). This is illustrated by Figure 1, where the bottom oval represents such a set of immediate neighbors.

Definition 2 (Subgroups) All processes of a group whose addresses diverge in the last k positions, i.e., which share a prefix of length $L - k$, form a *subgroup of level* k. With respect to a process p, such a subgroup contains all processes which share $p_L \cdot ... \cdot p_{k+1}$. Consequently, there is only one (sub)group of level L. The *top-level* subgroups are however those of level $L - 1$.

Numbering the subgroup levels inversely to the number of subdivisions applied enables the entire structure to be viewed as "hierarchy", when focusing on the *representatives* of these subgroups (see Figure 1).

Definition 3 (Representatives) For all levels $k \in [2..L]$, a process p knows only a subset of processes in its corresponding subgroup, namely at most R ($= 3$ in Figure 1) for every subgroup of level $k - 1$ within $p's$ subgroup of level k. These *representatives of level* k are chosen deterministically, recursively, out of the representatives of the level $k - 1$ subgroups within that respective subgroup of level k, e.g., by picking those q with smallest q_{k-1}.

A process which is representative at level k remains representative at every level $k' \in [2..k-1]$ to preserve links between levels.

Definition 4 (Ranks) The *rank* of a process denotes the highest level k at which that process appears (as representative).

Definition 5 (Root processes) Processes with a *rank* of L are called *root processes*, or simply *roots*. Roots which represent the same subdivision, i.e., whose addresses coincide in the first position, form a *root group*.

Every process however knows all root processes. This is illustrated by Figure 1, where the processes in the bottom subgroup only know the excerpt of the system represented by the processes visible in that figure.
Similar hierarchies have been described in literature informally, e.g., [14, 17, 28].

3.2 Algorithm Breakdown

Our *apbcast* algorithm can be subdivided into the following three algorithm parts (cf. Figure 1):

Membership algorithm: This first algorithm is responsible for the L-recursive group subdivision, and provides every process p with a weakly consistent view of its respective subgroup of every level [14].

These views, which for a given process p consist in the representatives of the respective subgroups of p. are maintained through periodic gossip-baed membership knowledge exchanges between processes.

Ordering algorithm: This second algorithm is executed between broadcasters and roots. Messages from broadcasters are received by these roots, which order them deterministically based on the timestamps of these messages (and previous messages) and the identifiers of broadcasters. Every broadcast message is thereby associated a message sequence number. To ensure Global Total Order, this ordering algorithm requires a stronger consistency with respect to the membership of root groups than the one underlying the recursive group subdivision.

Dissemination algorithm: This third algorithm is performed by all processes in a group. It consists in proactively propagating messages to be delivered, along with their respective sequence numbers computed by the ordering algorithm. This dissemination procedure takes place by periodically gossiping *recursively*, i.e., following the recursive subgroup division, about messages to be delivered. Due to the probabilistic nature of this dissemination procedure, certain processes may miss certain messages, especially since received messages may be dropped if subsequent (according to the computed order) messages have been received and APDELIVERed already. Thanks to the consistency of the computed order however, a process can easily detect missed messages, and can hence undertake more heroic effort to recover such messages (see Section 10.2).

These three parts are detailed in the next three sections respectively.

4 Membership Algorithm

In this section, we present our weakly consistent membership algorithm related to the recursive group subdivision in more details.

4.1 Weakly Consistent Views

Every process p has an approximate view of its respective subgroup of level k ($1 \leq k \leq L$), represented by $view[k]$ (Figure 2). Every process q part of a process p's $view[k]$ is characterized by $q_L = p_L$, ...,$q_{k+1} = p_{k+1}$ (see Definition 2).

More precisely, every p's $view[k]$ is itself a set of elements ($infix$, $suffixes$, t), where $\forall suff \in suffixes$, $|suff| = k-1$, and $p_L \cdot ... \cdot p_{k+1} \cdot infix \cdot suff_{k-1} \cdot ... \cdot suff_1$ denotes a process in the group. t denotes the last time the corresponding entry was updated. Note that $\forall(infix, suffixes, t) \in view[1]$, $suffixes = \emptyset$ (Line 2, Figure 2).

4.2 Update Propagation

Periodically, every process p gossips about its view of the group (UPDATE). To that end, p picks a representative for each level k. For each such representative q, p sends q those parts of $p's$ view which are also relevant for q. (Since a representative of level k is a representative at every level $k' \in [2..k-1]$, this is not simply p's $view[k..L]$.)

Upon receiving such an update a process q compares the received view with its own, and updates those parts where the timestamps reflect more recent information. Once this has been done, q's view is traversed (ORCHESTRATE), and representatives are chosen deterministically (ELECT), such that local information is merged with the information obtained. Note that clocks need not be synchronized, but of course the smaller clock drifts are, the better the algorithm performs.

```
For every process p
 1: INIT
 2:   view[2..L] ← ∅, view[1] ← (p_1, ∅, current time)
 3: task UPDATE                                {every P milliseconds}
 4:   view[1] ← view[1] \ {(p_1, , )}
 5:   view[1] ← view[1] ∪ {(p_1, ∅, current time)
 6:   for all k ∈ [1..L] do
 7:     dest ←RANDOM(p_L · ... · p_{k+1}, view[k], ∅)
 8:     comm | p_L = dest_L, ..., p_{comm+1} = dest_{comm+1},
        p_comm ≠ dest_comm, ...
 9:     SEND(VIEW, view[comm..L]) to dest
10: task MONITOR                               {remove obsolete entries}
11:   for all k ∈ [1..L] do
12:     for all (infix, , t) ∈ view[k] |
        current time − t > timeout do
13:       view[k] ← view[k] \ {(infix, , t)}
14:   ORCHESTRATE
15: function RANDOM(pref, horizon, except)
16:   return pref_{|pref|} · ... · pref_1 · infix · suff_{|suff|} · ... · suff_1
      ∉ except | (infix, { , suff, }, ) ∈ horizon
17: JOIN(help) occurs as follows:  {possibly several instances}
18:   while |view[1]| ≤ 1 do
19:     SEND(JOIN) to help
20:     wait until RECEIVE(JREP, view′[comm..L]) from
        help
21:       view[comm..L] ← view′[comm..L]
22:       if comm > 1 then
23:         help ←RANDOM(p_L · ... · p_{comm+1}, view[comm], ∅)
24: when RECEIVE(LEAVE) from q
25:   view[1] ← view[1] \ {q_1, , }
26: when RECEIVE(VIEW, view′[comm..L])
27:   for all k ∈ [comm..L] do
28:     for all (infix′, suffixes′, t′) ∈ view′[k] do
29:       if ∄ (infix′, , t) ∈ view[k] | t > t′ then
30:         view[k] ← view[k] \ {(infix′, , )}
31:         view[k] ← view[k] ∪ {(infix′, suffixes′, t′)}
32:   ORCHESTRATE
33: procedure ORCHESTRATE
34:   for all k ∈ [2..L] do
35:     view[k] ← view[k] \ {(p_k, , )}
36:     suffixes ←ELECT(view[k − 1])
37:     view[k] ← view[k] ∪ {(p_k, suffixes, current time)}
38: function ELECT(horizon)                    {|horizon| ≥ R}
39:   sel ← ∅, sub ← 0, suff ← 0
40:   while |sel| ≤ R do
41:     sub ← (infix, suffixes, ) ∈ horizon |
        (|sel| mod |horizon| + 1)th smallest infix
42:     suff ← (|sel| div |horizon|+1)th smallest ∈ suffixes
43:     sel ← sel ∪ {suff}
44:   return sel
45: LEAVE occurs as follows:
46:   for all (infix, , ) ∈ view[1] do
47:     SEND(LEAVE) to p_L · ... · p_2 · infix
48: when RECEIVE(JOIN) from q
49:   comm | q_L = p_L, , q_{common+1} = p_{common+1},
      q_common ≠ p_common,
50:   SEND(JREP, view[comm..L]) to q
51:   if comm = 1 then
52:     view[1] ← view[1] ∪ {q_1, ∅, current time}
```

Figure 2: Weakly consistent membership algorithm for recursive group subdivision

4.3 Suspicion and Failure Propagation

The same procedure is performed when nothing has been recently received from an immediate neighbor, and that process is suspected to have crashed after some time. A process p namely updates the t value for each $(infix, \emptyset, t)$ in its $view[1]$ every time it receives a membership update from the process with address $p_L \cdot \ldots \cdot p_2 \cdot infix$. Once p has not received any update for *timeout* milliseconds from an immediate neighbor, it suspects that process and removes it from its $view[1]$. By re-arranging processes (MONITOR, Line 13) any appearances of that suspected process as representative are removed.

4.4 Joining and Leaving

A joining process p must contact an arbitrary process p in the group until it receives a reply (JOIN). That reply consists of those parts of p's view that are "shared" with p. From there, p can extract information about a representative for a subgroup of lower level. This can be repeated (successively, and also in parallel), until immediate neighbours are reached.

Leaving processes (LEAVE) attempt to contact their immediate neighbours with simple, unreliable, communication. If this communication fails, processes may terminate anyway, as they will be handled just like failed processes.

Observation 1 For every process p and every k, p never appears in any q's $view[k]$ unless $q_{k'} = p_{k'} \forall k' \in [k+1..L]$.

Hence, by the absence of byzantine failures, no two processes ever appear as roots for a same process while being from distinct subgroups of level $L-1$.

5 Ordering Algorithm

This section presents the algorithm guiding the interaction between broadcasters and roots performing the message ordering.

5.1 Deterministic Merge

Our ordering algorithm is inspired by the Bias algorithm of Aguilera and Strom [1], but extends it to tolerate failures of roots or broadcasters for increasing its reliability, (i.e., availability, vs message logging in [1], see Section 11.1). In the Bias algorithm, a root decides from which broadcaster to take the next message by weighting the average production rates of broadcasters with the timestamps of the last messages chosen from them. Ties between two broadcasters are broken by chosing the broadcaster with smaller identifier. Broadcaster clocks need not be synchronized, but of course the performance of the algorithm becomes the better the closer clocks are synchronized. The strength of the Bias algorithm is its proven minimal delaying of messages produced by broadcasters according to memory-less random processes [1], and hence its minimal space required for buffering messages prior to ordering and delivering them.

5.2 Root Group Consistency

Though [1] presents many variants of the Bias algorithm (e.g., with bound delaying), root groups always consist of single roots. In *apbcast*, a root group consists of replicated roots.

Reliable Broadcast. Messages are broadcast from broadcasters to individual root groups by making use of a virtual synchrony Reliable Broadcast algorithm (cf. [8]):

Validity If a correct process executes BCAST$^{vseq}(m)$, then it eventually DELIVERS m (in view *vseq* or a subsequent view).

Termination If a correct process executes BCAST$^{vseq}(m)$, then eventually (1) every process in the view *vseq* executes DELIVER$^{vseq}(m)$ or (2) every correct process in *vseq* INSTALLS a new view.

View Synchrony If process p belongs to two consecutive views *vseq* and $vseq+1$, and executes DELIVER$^{vseq}(m)$, then every process q in $vseq \cap vseq+1$ that INSTALLS $vseq+1$, also executes DELIVER $vseq(m)$.

Sending View Delivery A message BCAST in view *vseq*, if DELIVERed, has to be DELIVERed in view *vseq*.

Integrity For any message m, every correct process DELIVERS m at most once, and only if m was previously BCAST.

By relying on group membership, such a primitive incurs certain "costs" regarding its implementation but also its use (e.g., program forced crash). These have been largely discussed in literature (e.g., [26, 7]), and will hence not be repeated here. The main consequence to keep in mind in the present context is that an indeed correct process can be excluded from a group at some *vseq*, and can fail to DELIVER all messages BCAST in $vseq-1$.

The resulting membership views complement the weakly consistent view information (*infix*, *suffixes*) of the membership algorithm outlined in the previous section (cf. Lines 13 and 16 in Figures 4 and 5 resp.).

Being employed to broadcast to individual root groups, information on the targeted root group will be added, where relevant, to primitives, through the L-th address component common to roots of the corresponding root group (e.g., ^{infix}). In the following, “broadcast”, as well as “deliver”, will hence refer to the use of this primitive (vs APBCAST and APDELIVER respectively). The term “broadcaster” still refers to processes APBCASTing, as those are the only ones to BCAST.

Views. Without loss of validity we assume that a view *vseq* for a root group *infix* contains a set of roots, as well as a set of broadcasters. Furthermore, latter processes are clearly distinguishable from the roots. Their identifiers are chosen such as to convey information on (1) their respective broadcast rates, and (2) the respective timestamps of their first broadcast messages to the root group *infix*. Hence, a root group *infix* is joined by a broadcaster b through $\text{JOIN}^{infix}(P[b],\ f[b]_{infix})$. Any root in a root group *infix*, can access this knowledge then through $P[b]$ and $f[b]$ (accessing $f[b]_{infix}$ in latter case).

5.3 Broadcast Message Subsequences

The use of virtual synchrony Reliable Broadcast cuts the sequence of broadcast messages into subsets *delivered* in between subsequent view changes. The messages delivered in a view *vseq* will be said to belong to that view *vseq*. Similarly, these views cut the sequence of messages into those *ordered* in between subsequent views. However, not necessarily all messages delivered up to view *vseq* (including those delivered in *vseq*) can be effectively ordered up to *vseq*.
For a process which acts as root in a given view *vseq* to be able to order deterministically the messages delivered in that view, it has certain knowledge prerequisites. Being inspired by the Bias algorithm, our ordering algorithm relies on similar information to achieve deterministic ordering:

Definition 6 (Ordering configuration) For a given root, the (1) timestamps of messages of each broadcaster last ordered by that root, and (2) the rates of these broadcasters form its *ordering configuration*, or simply *configuration*.

These are stored as tuples of the form (b, *freq*, *last*) by each broadcaster b at each root (Figure 4). At the beginning of a view *vseq*, the *initial* ordering configuration refers to the values of the configuration after ordering by considering all messages delivered up to *vseq*, but without those of subsequent views. To this specific configuration, we also count (cf. Section 10.2: the messages, and) the timestamps of messages delivered up to *vseq* but not ordered (*buf*), as well as the messsage sequence counter *mseq*.
Figure 3 illustrates these principles. $m_{2.2}$ is for instance delivered in view $vseq = 1$, but only ordered in view 2, e.g., because the next broadcaster from which a message is to be taken is different from b_2. A new root joining at view 1, must be passed the configurations of the other root processes after ordering in view 1, i.e., O_1. This will include message $m_{2.2}$, or at least the timestamp information of that message. In contrast, if message $m_{4.2}$ was not ordered in view 1, it could be dropped when ordering in view 2, since broadcaster b_4 is not within that view anymore.

5.4 Asynchronous Replicated Deterministic Merge

A root acting in a given view *vseq*, though already delivering broadcast messages for that view, can still lack information pertaining to the initial configuration for that view (or even for the view $vseq - 1$ before that, etc.). This can occur if that root was not in the previous view but just joined (possibly after an erroneous removal following a false suspicion), or a new broadcaster has been added. Hence, the algorithm given in Figure 4 can proceed asynchronously, which manifests in that both configurations (*bcers*) as well as the sets of *roots* are indexed by view sequence numbers. Though not possible in the simplified algorithm of Figure 4, a root can in the extreme case simultaneously order messages pertaining to different views *vseqs* (see Section 10.2).

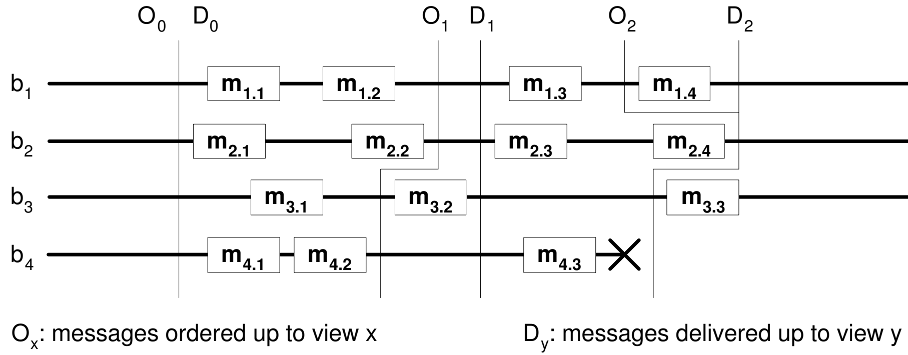


Figure 3: Ordering and delivering of messages

The algorithm in Figure 4 proceeds as follows: procedure MERGE orders the available messages. To ensure that messages coming from the same broadcaster are taken in the same order by all roots, a FIFO order is imposed on these (Line 35), reflecting in that messages delivered from broadcasters carry also the timestamps of the previous messages from the respective broadcaster (Line 30). After successfully ordering all messages delivered up to a given view *vseq*, TRANSIT computes the initial configuration for the following view $vseq + 1$. Once that new view is installed (Line 13), the initial configuration can be computed based on (1) the *terminal* configuration of the previous view, and (2) the broadcast rates and first messages of new broadcasters. This information is sent to all new roots of that view (Line 55).
Since the ordering of messages of a view *vseq* can be delayed by lacking information on the initial configuration for that view (and possibly all views up to a $vseq' << vseq$), MERGE can be triggered either when such information is received (Line 12), when new broadcast messages are delivered in that view *vseq* (Line 32), or after a TRANSITion to that view after successful termination of the previous one (Line 56). Such a TRANSITion itself can be triggered by the installation of a new view (Line 20) or successful ordering of all messages pertaining to the previous view (Line 44), if that task was delayed until after the view change.

5.5 Broadcasting

As conveyed by the algorithm in Figure 5, a broadcaster stores various data structures indexed by the *infix* of the root groups. These data structures represent the composition of all root groups, i.e., (1) the *roots*, and (2) the broadcasters (*bcers*). Furthermore, (3) the respective view sequence numbers (*vseqs*), and (4) the timestamps of the *last* messages broadcast.
A process willing to broadcast must first join the root groups (START, Figure 5). This procedure is also repeated whenever a broadcaster is falsely suspected to have crashed by a given root group *infix* (INSTALL), and hence is removed from that root group. For presentation simplicitly, the time indicated for the first broadcast (Line 30) is supposed to be sufficiently far away in the future for the joining to succeed. This means that for any root group *infix* joined at some *vseq* by a broadcaster b_i, there must be at least one message from a broadcaster b_j that was broadcasting already in $vseq - 1$ within that root group *infix* to be ordered before the first one of b_i through the Bias comparison at Line 35. For presentation simplicity, this is not integrated in the algorithm in Figure 4.
Note that a broadcaster may only APBCAST if it is member of its own root group. This is required for Validity (see Section 7).

```
For every root process r
 1: INIT
 2:   roots ← ∅                                   {root processes}
 3:   bcers ← ∅              {broadcasters and production rates}
 4:   mseq ← 0                          {message sequence number}
 5:   msgs ← ∅                                {ordered messages}
 6:   buf ← ∅                               {unordered messages}
 7: when RRECEIVE(WELCOM, bcers', buf', vseq, mseq)
 8:   if bcers[vseq] ≠ bcers' then
 9:     bcers[vseq] ← bcers'
10:     buf ← buf'
11:     mseq ← mseq'
12:     MERGE(vseq)
13: when INSTALL^{r_L}_{vseq}({r_1, ..., r_x, b_1, ..., b_y})
14:   if r ∈ {r_1, ..., r_x} then
15:     roots[vseq] ← {r_1, ..., r_x}
16:     for all b ∈ {b_1, ..., b_y} do
17:       if ∄(b, , ) ∈ bcers[vseq] then
18:         bcers[vseq] ← bcers[vseq] ∪ {(b, ⊥, ⊥)}
19:     if r ∈ roots[vseq − 1] and
        ∄(b, ⊥, ⊥) ∈ bcers[vseq − 1] then
20:       TRANSIT(vseq)
21:   else if ∃r_{L−1}·...·r_1 ∈ suffixes | (r_L, suffixes) ∈ view[L]
      then
22:     buf ← ∅
23:     JOIN^{r_L}
24: when (r_L, suffixes) ∈ view[L] changes to (r_L, suffixes')
25:   if r_{L−1} · ... · r_1 ∈ suffixes' then
26:     if r_{L−1} · ... · r_1 ∉ suffixes then
27:       JOIN^{r_L}
28:   else if r_{L−1} · ... · r_1 ∈ suffixes then
29:     LEAVE
30: when DELIVER^{r_L}_{vseq}(BCAST, b, m, curr, prev)
31:   buf ← buf ∪ {(b, m, curr, prev)}
32:   MERGE(vseq)
33: procedure MERGE(vseq)
34:   if ∄(b, ⊥, ⊥) ∈ bcers[vseq] and roots[vseq] ≠ ∅ then
35:     while ∃ (b, m, curr, prev) ∈ buf and
        ∃ (b, freq, prev) ∈ bcers | prev + BIAS(freq, vseq) =
        min_{(b_j, freq_j, last_j)∈bcers[vseq]}(last_j+BIAS(freq_j, vseq)) do
36:       buf ← buf \ {(b, m, curr, prev)} {smallest such b}
37:       gossips[L−1] ← gossips[L−1] ∪ {(m, mseq, 0)}
38:       if b_L = r_L then
39:         RSEND(ACK, m, mseq) to b
40:       mseq ← mseq + 1
41:       bcers ← bcers \ {(b, freq, prev)}
42:       bcers ← bcers ∪ {(b, freq, curr)}
43:     if roots[vseq + 1] ≠ ∅ then
44:       TRANSIT(vseq + 1)
45: procedure TRANSIT(vseq)
46:   for all (b, ⊥, ⊥) ∈ bcers[vseq] do
47:     bcers[vseq] ← bcers[vseq] \ {(b, ⊥, ⊥)}
48:     if ∃(b, freq, last) ∈ bcers[vseq − 1] then
49:       bcers[vseq] ← bcers[vseq] ∪ {(b, freq, last)}
50:     else
51:       bcers[vseq] ← bcers[vseq] ∪ {(b, P[b], f[b])}
52:   for all (b, m, , ) ∈ buf | ∄(b, , ) ∈ bcers[vseq] do
53:     buf ← buf \ {(b, m, , )}
54:   for all roots[vseq] \ roots[vseq − 1] do
55:     RSEND(WELCOM, bcers[vseq], buf, vseq, mseq)to r_k
56:   MERGE(vseq)
57: function BIAS(freq, vseq)
58:   ref ← max_{(b_i, freq_i, ) ∈ bcers[vseq]}(freq_i)
59:   return log(freq) / log(1 − ref)
```

Figure 4: Ordering: root process algorithm

6 Dissemination Algorithm

This section presents the gossip-based propagation of broadcast messages and their sequence numbers computed by the ordering algorithm.

6.1 Recursive Subgroup Gossiping

When disseminating a broadcast message, *apbcast* gossips recursively, i.e., following the underlying recursive group subdivision (GOSSIP&DELIVER and REPOSITION in Figure 6). To ensure that a gossip message passes from one level to the next, it is crucial that a process at level i gossips about received broadcast messages in any level $j < i$, and thus also remains in the view of any of these lower levels. Processes have a gossip buffer for each level, and each gossiped message is attached, besides the sequence number determined by the ordering algorithm presented in the previous section, the level at which the message is currently being gossiped about, as well as the number of times it has been forwarded at that level already.
Note that a broadcaster participates in the dissemination of any message it issues at any level $k \in [1..L-1]$ within its own top-level subgroup, regardless of its own rank.

```
For every broadcaster b
 1: INIT
 2:   for all root group infix do
 3:     roots[infix] ← ∅                          {root processes}
 4:     bcers[infix] ← ∅                          {broadcasters}
 5:     vseqs[infix] ← ⊥                  {view sequence numbers}
 6:     last[infix] ← ⊥                           {last messages}
 7: APBCAST(m) at time t occurs as follows:
                               {iff vseqs[b_L] ≠ ⊥ and last[b_L] < t}
 8:   for all infix | last[infix] < t and vseqs[infix] ≠ ⊥ do
 9:     BCAST^infix(BCAST, b, m, t, last[infix])          {b_L first}
10:     last[infix] ← t
11: STOP occurs as follows:
12:   for all infix do
13:     LEAVE^infix
14:     vseqs[infix] ← ⊥
15: when RRECEIVE(ACK, m, mseq) for first time
16:   gossips[L − 1] ← gossips[L − 1] ∪ {(m, mseq, 0)}
16: when INSTALL^infix_vseq({r_1, ..., r_x, b_1, ..., b_y})
17:   if b ∈ {b_1, ..., b_y} then
18:     if b ∈ bcers[vseq − 1] then
19:       last[infix] ← last[vseq − 1]
20:     else
21:       last[infix] ← f[b]_infix
22:     bcers[infix] ← {b_1, ..., b_y}
23:     roots[infix] ← {r_1, ..., r_x}
24:     vseqs[infix] ← vseq
25:   else
26:     vseqs[infix] ← ⊥
27:     START
28: START occurs as follows:
29:   for all infix | vseqs[infix] = ⊥ do
30:     f[b]_infix ← estimated time of first
31:     JOIN^infix(P[b], f[b]_infix)
```

Figure 5: Ordering: broadcaster algorithm

```
For every process p
 1: INIT
 2:   last ← ⊥                         {sequence nb of last msg}
 3: procedure REPOSITION(m, mseq, rnd, k)
 4:   if k > 1 then
 5:     gossips[k−1] ← gossips[k−1] ∪ {(m, mseq, 0)}
 6:   else if mseq > last and
      ∄ (m', mseq', ) ∈ gossips | mseq > mseq' then
 7:     last ← mseq
 8:     APDELIVER(m)
 9:   else
10:     gossips[1] ← gossips[1] ∪ {(m, mseq, rnd)}
11: task GOSSIP&DELIVER                     {repeat periodically}
12:   for all k ∈ [L − 1..1] do                 {from top to down}
13:     for all (m, mseq, rnd) ∈ gossips[k] by increas. mseq do
14:       gossips[k] ← gossips[k] \ {(m, mseq, rnd)}
15:       if rnd < c ln |view[k]| then            {limit rounds}
16:         gossips[k] ← gossips[k] ∪ {(m, mseq, rnd + 1)}
17:         SEND(GOSSIP, m, mseq, rnd, k) to F random
            processes ∈ view[k]
18:       else
19:         REPOSITION(m, mseq, rnd, k)
20: when RECEIVE(GOSSIP, m, mseq, rnd, k)
21:   if mseq > last and ∄ (m, mseq, ) ∈ gossips then
22:     gossips[k] ← gossips[k] ∪ {(m, mseq, rnd)}
```

Figure 6: Dissemination algorithm

6.2 Bound Gossiping

At each level, the expected number of rounds necessary to propagate a gossip among the processes in the considered process p's subgroup of that level is approximated, based on the size of the respective subgroups (Line 15 in Figure 6). This enables the limiting of the number of messages involved in the dissemination of a broadcast messages.
Ways of improving reliability by adding to this simple scheme will be discussed in Section 10.2.

6.3 Message Delivery

According to the above scheme, broadcast messages are namely mainly buffered for the duration of the time they are being gossiped about. After that, a broadcast message is only further stored/its delivery delayed (Line 10), if a message with a smaller sequence number (i.e., to be delivered earlier) has been received meanwhile and is still being gossiped about (Line 6). Through the absence of explicit synchronization on gossip rounds between processses, this might indeed occur, albeit with small probability.

To respect total order, a message RECEIVEd with a sequence number smaller than that of the last APDELIVERed message however can not be delivered. This is reflected in Figure 6, by dropping such messages immediately when they are RECEIVEd. Alternatively, such a message could be delivered to the application by indicating that it is out of order. It is however more likely that the application would want to undertake explicit efforts to acquire a relevant message when detecting the absence of that message (see Section 10.2).

7 Correctness

In this section, we prove the correctness of our *apbcast* algorithm with respect to the specification given in Section 2.2.

Lemma 1 (**Merge Completion**) If all correct roots of a root group *infix* in view *vseq* achieve the same initial configuration for *vseq*, they all associate the same sequence numbers to messages in *vseq*, i.e., in MERGE(*vseq*).

Proof. Suppose all roots achieve identical, non-empty, *buf*, *bcers*[*vseq*], *roots*[*vseq*], and *mseq*, before starting to MERGE for *vseq*. Hence, no matter how triggered, the procedure MERGE can only choose the same messages and the same *mseq*s, as each decision is made deterministically based only on *bcers*[*vseq*], which reflects previous choices. Since all roots in a root group DELIVER the same messages in *vseq* itself, and all had the same messages DELIVER in previous views and not ordered stored in *buf*, they either all eventually can assign a sequence number to a given m in *vseq* or none can. That they all effectively do so is ensured by the fact that every DELIVER trigger a new MERGE (Line 32 in Figure 4), and by the assumption that they all at some point reach the initial configuration for *vseq*: either (1) a root was in $vseq - 1$ already, and hence must have executed TRANSIT($vseq - 1$), leading to a subsequent MERGE (Line 56) to deal with pending messages already DELIVERed in *vseq*, or (2) the root joined in *vseq*, meaning that it RRECEIVEd its initial configuration at Lines 9-11, after which MERGE is similarly performed.

Lemma 2 (**Transit Completion**) A correct root of a root group *infix* in views *vseq* and $vseq+1$ which can order messages in *vseq* eventually achieves the initial configuration for $vseq + 1$.

Proof. When view $vseq + 1$ is installed, a correct root in *vseq* and $vseq + 1$ executes TRANSIT at Line 20 in Figure 4. In case the root had reached the initial configuration for *vseq*, it has ordered all messages up to *vseq*, as by that time no more messages can be DELIVERed in *vseq*. In TRANSIT, every broadcaster for $vseq + 1$ is initialized. More precisely, for every entry (b, *freq*, *last*) in *bcers*[*vseq*], broadcaster b will only more broadcast messages with a timestamp larger than *last* in $vseq + 1$. Hence, this timestamp can be taken as initial value for $vseq + 1$ (Line 49). Any other broadcaster is new in $vseq + 1$, and its timestamp is hence initialized to the first expected broadcast (Line 51).

Lemma 3 (**Local Root Ordering**) All correct roots in a root group assign the same sequence numbers to messages.

Proof. Suppose this is true for all messages up to a given *vseq* (i.e., in MERGE($vseq' \leq vseq$)). By Lemma 2, all correct roots reach the initial configuration for $vseq + 1$. As these roots in all cases assign the same sequence numbers to messages in *vseq*, their ordering configurations at $vseq + 1$ must be identical. Since to reach this configuration, they must execute TRANSIT, they execute Line 55 in Figure 4, meaning that all new roots in $vseq+1$ reach this configuration as well. By Lemma 1, all roots in $vseq + 1$ then MERGE the same messages, and hence assign the same sequence numbers to these, in $vseq+1$. Showing that the algorithm bootstraps correctly is trivial as long as no broadcaster is the first member of a root group.

Lemma 4 (**Global Root Ordering**) There exists a unique ordering $\{m_i\}_i$ on all broadcast messages such that if any root APDELIVERs m_i before m_j, then $i < j$.

Proof. In the case where all root groups DELIVER the same broadcast messages, i.e., from the same set of broadcasters, the FIFO order with respect to broadcasters imposed at Line 35 in Figure 4 and Lemma 3 leads to the same situation as in the original Bias algorithm (see [1]). Global Total Order is also ensured by the sequence numbers assigned to messages when roots receive messages from (only) overlapping sets of broadcasters, due to the use of timestamps. The only delicate case, i.e., the joining of new broadcasters in one or several root groups has been discussed in Section 5.5.

Theorem 1 *apbcast* ensures Global Total Order.

Proof. By Lemmas 3 and 4. Global Total Order is ensured by the sequence numbers associated by correct roots to messages. Since any process only APDELIVERs messages with larger sequence number than the last APDELIVERed one (Line 6 in Figure 6), that sequence number is retained (Line 7), no antecedent message (according to the sequence numbers) can be APDELIVERed.

Lemma 5 (**Eventual Ordering**) A message broadcast to a root group by a correct broadcaster is eventually ordered and APDELIVERed by every correct root in that group.

Proof. A message m broadcast by b to a root group *infix* at time t in *vseq* is DELIVERed by all correct roots of *infix* in *vseq*. Hence an entry $(b,\ m,\ t,\ prev)$ is added to the respective *buf* at Line 31 in Figure 4, where *prev* is the timestamp of the last message APDELIVERed from b (or the planned initial message). As long as the corresponding broadcaster b remains correct, i.e., remains in the root group *infix*, every correct root in *infix* will have an entry $(b,\ freq,\ prev)$ in *bcers*, since it is passed along from any *vseq* to its followup $vseq+1$ at Line 49 by any correct root, and is passed to any new root at Line 55. Similarly, m will remain in *buf*. By Lemma 1 the roots keep executing MERGE. Since broadcasters which LEAVE or crash are not kept in configurations after the view at which they are excluded (Line 16), new messages are continuously DELIVERed from correct broadcasters broadcasting continuously. Thus, there is a time and a view $vseq'$ at which the condition at Line 35 is satisfied, and m is APDELIVERed.

Theorem 2 *apbcast* ensures Validity.

Proof. By Lemma 5, a message broadcast by a correct broadcaster b is eventually assigned a sequence number by all correct roots in the root group of that broadcaster. b is sent the corresponding sequence number at Line 39 in Figure 4 by each of these root processes. When received by b at Line 16 in Figure 5, the message is inserted into *gossips*. Eventually, Line 6 in Figure 6 is reached, and the message APDELIVERed by b.

Theorem 3 *apbcast* ensures Integrity.

Proof. By the absence of Byzantine failures (see Section 2.1), and the specification of virtual synchronous Reliable Broadcast, no spurious message is ever DELIVERed. A message m BCAST by a broadcaster (which happens at most once) to a root group is DELIVERed at most once by any root in that group, and hence added once to the buffer for incoming messages *buf* at Line 31 in Figure 4. Since messages are only assigned sequence numbers after being removed from *buf* at Line 36, no message can be APDELIVERed twice by a root. The case of the broadcaster of is trivial by Theorem 2.

Theorem 4 *apbcast* ensures Probabilistic Agreement with $\alpha = 1$ whenever q is a root and p is a root as well, or a broadcaster.

Proof. When a correct broadcaster broadcasts to all root groups, Lemma 5 is satisfied for all those groups, and hence a message is eventually ordered by all root groups, and hence inserted into *gossips* at each correct root. The case of the broadcaster is covered by Theorem 2.

The next section provides (among other things) a global, analytical, evaluation of Probabilistic Agreement, i.e., the expected reliability degree α when including the gossip-based propagation of messages with their sequence numbers also to non-root/broadcasting processes.

8 Analysis

In this section, we analyze the ordering and broadcasting procedures, and provide bounds on the complexity of our *apbcast* algorithm. We are interested in the (1) memory overhead of the membership, the (2) "time"- and and (3) message overhead of a broadcast, the (4) resulting reliability of the algorithm, and last but not least, (5) the fault tolerance of our algorithm.

8.1 Model

We consider a snapshot of an *apbcast* group composed of n processes, arranged according to a *regular* group subdivision of constant depth L: every subgroup of level $k \in [1..L]$ contains the same number $A = n^{\frac{1}{L}}$ of subgroups of level k ($\forall k \in [1..L], A \leq a_k$). We consider a stable phase, i.e., all memberships views are initialized. We similarly assume for analysis only that processes gossip in synchronous rounds (cf. [12, 3, 27, 15]), and that there is an upper bound on the network latency which is smaller than a gossip period P. P is constant and identical for each process, just like the fanout $F < A$. Failures are stochastically independent. The probability of a network message loss does not exceed a predefined $\epsilon > 0$, and the probability of a process (neither broadcaster nor root) crash during a run is τ.
The overhead of root-broadcaster algorithm is not reported below when its complexity with respect to the considered measures is smaller than that of the gossiping as long as $L \geq 2$.

8.2 Membership Memory Overhead

Theorem 5 For a fixed A, the number of processes a process knows in an *apbcast* group is $O\left(n^{\frac{1}{L}}\right)$.

Proof. For each level, every process must know the representatives of its subgroup at that level (all processes in its lowest level subgroup), without those already known from the lower level (if any). For any level $i \in]1..L]$ in a regular subdivision, it is easy to see that $M_i = R\ (A-1)$, and $M_1 = A$.

$$\sum_{i=1}^{L} M_i = R\ (A-1)\ (L-1) + A = R\ (n^{\frac{1}{L}} - 1)\ (L-1) + n^{\frac{1}{L}}$$

This is asymptotically valid for very large groups, i.e., assuming that the growth of the group is "distributed" over the entire subdivision (but L is constant).

8.3 Broadcast Time Overhead

Theorem 6 The number of gossip rounds required for disseminating a messsage with *apbcast* is around $O(ln\ n)$.

Proof. In terms of gossip rounds, a broadcast message has an overhead of $T_i = c\ ln(A\ R)$ rounds at each level $i \in]1..L-1]$ and simply $T_1 = c\ ln\ A$ at the lowest level (see Line 15 in Figure 6). This yields the following total number of gossip rounds for disseminating a message with *apbcast*:

$$\sum_{i=1}^{L-1} T_i = (L-2)\ c\ ln(A\ R) + c\ ln\ A = (L-2)\ c\ \left(\frac{1}{L}\ ln\ n + ln\ R\right) + \frac{c}{L}\ ln\ n$$

8.4 Broadcast Message Overhead

Theorem 7 The dissemination of a broadcast message with *apbcast* requires $O(n \; ln \; n)$ messages.

Proof. The message overhead due to BCAST incurs an overhead of $O\left((B+R)^2\right)$ messages in each of the A root groups. Considering Ψ at a given moment, the total overhead to the ordering algorithm is of $O\left(n^{\frac{1}{L}}\right)$. The number of processes which gossip at a given level $i \in \;]1..L[$ on the other hand is given by $D_i = R \; A^{L-i+1}$, while $D_1 \; = \; A^L$. Hence, the expected total number of gossip messages for a given broadcast message is given by (pessimistic)

$$F \sum_{i=1}^{L-1} D_i \; T_i \; = F \sum_{i=2}^{L-1} \; R \; A^i \; c \; ln(R \; A) + F \; A^L \; c \; ln \; A$$

$$= F \; R \; c \left(ln \; R + \frac{1}{L} \; \frac{n - n^{1-\frac{2}{L}} - n^{\frac{1}{L}} + 1}{n^{\frac{1}{L}} - 1} \; ln \; n \right) + F \; c \; n \; \frac{1}{L} \; ln \; n$$

This message complexity hence largely prevails over that of the ordering algorithm (which involves $O\left(A \; (B+R)^2\right)$ messages for APBCASTing a message). Both broadcast time and message overhead achieve hence orders typical for (unordered) gossip-based broadcast (in a non-hierarchical group), cf. [20] (just like in the case of the message overhead, the time overhead of the ordering has a lower complexity than that of the gossiping).

8.5 Expected Reliability Degree

Lemma 6 $1 - \alpha$ is of $\Theta\left(n^{-\frac{1}{L}}\right)$ in stable phases.

Proof. Considering stable phases means that every broadcaster is in all root groups, and that when a message m is APDELIVERed by a process, any message with a smaller sequence number that will ever be received by that process has been received until then. The opposite leads to simply ignoring such messages, and can occur in practice, albeit with a very small probability.
The most common model for approximating the spreading of a disease in a heterogenous population of size n without considering births, depths, or immunity [13], uses the following differential equation

$$\frac{dI}{dt} = \frac{b}{n} I(n - I), \text{ and hence } I = \frac{n}{1 + ne^{-bt}}$$

where I denotes the number of infected entities at time t, and b is the number of “contacts” of each infected entity per time unit. In our case, $b = F(1-\tau)(1-\epsilon)$. With this, the probability that an entity at level i is infected after gossiping at that level for T_i rounds, is given by $g_i = 1 - \left(1 - \frac{1}{1+RAe^{-bT_i}}\right)^R$ for a level $i \in \;]1..L[$ (at least one of R roots), while $g_1 = \frac{1}{1+Ae^{-bT_i}}$. Since T_i is given by $c \; ln \; AR \; \forall i \in]1..L]$ and T_1 is $c \; ln \; A$, the expected reliability degree is given by:

$$\prod_{i=1}^{L-1} g_i = \frac{1}{1 + A^{-(b \; c-1)}} \left(1 - \left(1 - \frac{1}{1 + (RA)^{1-b \; c}}\right)^R\right)^{L-2} \approx \left(1 - A^{1-b \; c}\right)\left(1 - (L-2)(R \; A)^{R(1-b \; c)}\right)$$

$$\geq 1 - A^{1-b \; c} - (L-2)(R \; A)^{R(1-b \; c)} \qquad = 1 - \frac{1}{n^{\frac{b \; c-1}{L}}} - \frac{L-2}{n^{\frac{R(b \; c-1)}{L}}}$$

Theorem 8 In stable phases, the expected reliability degree achieved by *apbcast* comes close to 1, and increases as n increases.

Proof. Follows immediately from Lemma 6.

8.6 Fault Tolerance

Through the use of a virtually synchronous Reliable Broadcast for communication between broadcasters and roots our *apbcast* algorithm inherits the "majority-requirement" of that primitive, and can hence tolerate *at least* $\lfloor\frac{R+B-1}{2}\rfloor$ root/broadcaster failures, since broadcasters become part of root groups. By considering broadcasters and roots separately, our algorithm can tolerate $\lfloor\frac{B-1}{2}\rfloor$ broadcaster failures, *and* $\lfloor\frac{R-1}{2}\rfloor$ root failures within *each* root group *at least* (since $\lfloor\frac{R-1}{2}\rfloor + \lfloor\frac{B-1}{2}\rfloor \leq \lfloor\frac{R+B-1}{2}\rfloor$). Increasing R might be hence tempting, but care must be taken as this also increases the overhead of root-broadcaster communication, which involves $O\left(A\ (B+R)^2\right)$ messages for APBCASTing a message.
With respect to failures of processes not belonging to those two categories, the performance of *apbcast* degrades as gracefully as any gossip algorithm (based on recursive subdivision, cf. [14]).

9 Simulation Results

This section presents simulation results obtained with *apbcast*.

9.1 Setting

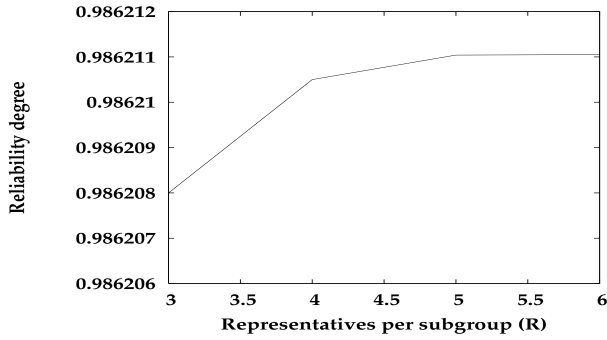
Simulation results were computed by simulating up to 100 processes on each of 107 Sun Ultra 10 machines (Solaris 2.7, 256 Mb RAM, 9 Gb harddisk). The individual stations, which were communicating via Fast Ethernet (100 Mbit/s), were arranged according to a $L = 3$-subdivison.
Crash failures were simulated by stopping a process with probability $\tau = 0.1$ at the beginning of the simulation, and message sends were omitted with a probability $\epsilon = 0.1$ to simulate unreliable communication. Through the use of UDP for communication between remote processes, further losses were incurred. A message was broadcast at every gossip period by a random process. Default values (i.e., unless varied) for the simulations were $R = 3, F = 3, A = 20$.

9.2 Reliability

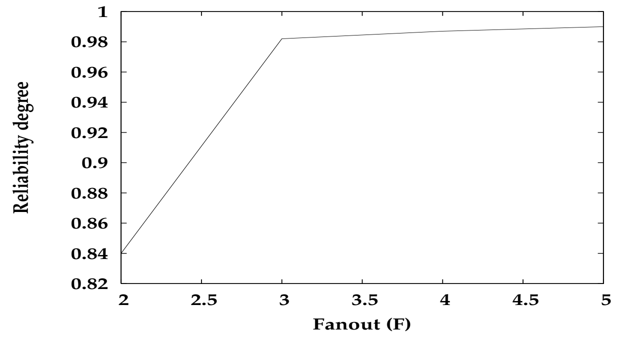
Various parameters have an impact on the reliability of our *apbcast* algorithm.

Redundancy. Figure 7(a) confirms that the reliability degree increases also when increasing the number of representatives for each subgroup R. This makes intuitively sense, as this increases the number of processes gossiping at each level, and also the probability that at each such level at least one process receives a broadcast message. This in turn strongly decreases the probability that an entire subgroup of any level is isolated. The figure also illustrates that this improvement stagnates when R further increases. The expected downside of increasing R, i.e., an increased latency of ordering, could not be directly measured. This effect is probably masked by the asynchrony of the system within the considered range of values.

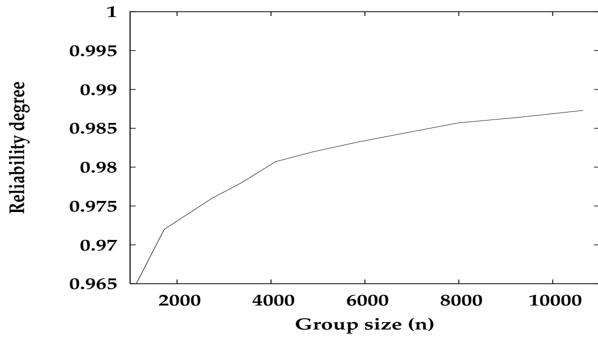
Fanout. Figure 7(b) shows that the reliability degree can obviously be pushed further than visible in the previous figures by increasing the fanout F. However, also here, after a given value, only little more is gained. This stems from the fact that the number of gossip rounds is computed at each level based also on this fanout ($c \in O(\frac{1}{F})$ in Figure 6).



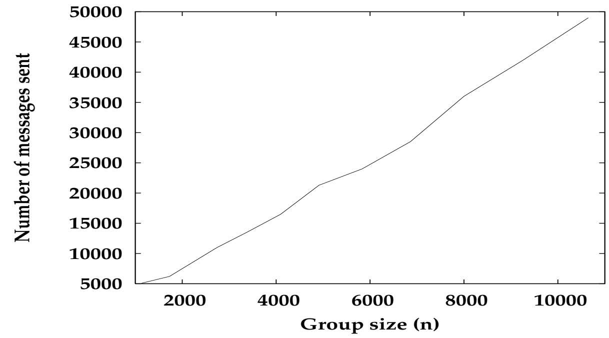
(a) Reliability (redundancy)



(b) Reliability (fanout)



(c) Scalability (reliability)



(d) Scalability (message complexity)

Figure 7: Simulation results

9.3 Scalability

Reliability. Figure 7(c) illustrates that, according to analysis, the reliability degree, i.e., the probability of delivery achieved with *apbcast*, increases slightly with an increasing system size (see Section 8.5). A is varied from 11 to 22, meaning that the group size increases by a factor of 8. In a realistic setting, it would hence make sense to decrease the fanout F in function of the size for a given subgroup, in order to keep α closely constant. The effect of increasing A on the latency of ordering, according to analysis even smaller than that of increasing R, could indeed not be observed in our setting.

Message complexity. Figure 7(d) confirms analysis by showing how the number of messages required for propagating a message with *apbcast* increases slightly sur-linearly. However, this traffic is greatly dispersed, thanks to the recursive subgrouping of proceses. With a smart scheme for determining addresses reflecting network topology [23], this benefit can be strongly exploited.

10 Discussion

This section discusses various issues, including possible ways of tuning our *apbcast* algorithm.

10.1 Atomic Multicast

Besides reducing membership knowledge stored at each process in a group, and naturally embracing the deterministic ordering of messagess, the recursive subdivision advocated by our approach can also be easily used to perform selective gossiping and hence to implement Atomic (Probabilistic) *Multicast* by reducing the amount of messages sent to processes without them being interested in those messages (cf. [14]).
By associating with each process p_i an interest predicate $\triangleleft_{p_i} : m \rightarrow [true,\ false]$ (with corresponding entries in views) evaluated against multicast messages to determine which messages that a process is interested in, the (potential) set of destinations for a given multicast message is *implicitly* given. A representative for a set of processes $\{p_i,\ p_j,\ ....\}$ then manifests interest in any message m such that $\triangleleft_{p_i}(m) \vee \triangleleft_{p_j}(m) \vee ... = true$. At each level then, gossiping takes place selectively, i.e., only among "interested" processes.
[1] presents an ordering algorithm for solving (Global) Atomic Multicast, which could be adapted to be used by root processes in *apbcast* instead of the one presented in Section 5. Given the emphasis on large scale of our approach, practical settings have however manifested a very high likelihood that for each multicast message m, *at least one* process in every subgroup of level $L-1$ is interested in m. In such scenarios, the ordering algorithm in Figures 5 and 4 can be retained. Messages are then broadcast to *all* root groups, and selective propagation takes place only at subsequent levels.

10.2 Tuning *apbcast*

Various tuning mechanisms have been considered in *apbcast*, but have not been reported so far for presentation simplicity.

Digest forwarding. Quite obviously, the forwarding of *buf* to new roots joining a root group at a given *vseq* at Line 55 in Figure 4 can incur a considerable overhead if *buf* contains many messages DELIVERed but not APDELIVERed up to *vseq*. This can be easily alleviated by transferring only the timestamp information corresponding to the messages in *buf*, i.e., by replacing the effective messages by $\perp$. Any process of course refrains from APDELIVERing such a message. Through the the redundancy of gossips, it is however very likely that processes receive such messages (indirectly) from any of the other roots already in $vseq-1$.

Increasing asynchrony in message ordering. To similarly avoid transferring messages to roots joining at some *vseq* which have been erroneosly excluded at some $vseq' < vseq$, roots can keep track of the view during which messages are DELIVERed, and only transfer relevant messages to joining roots. Latter processes would when then also refrain from purging their *buf* when being excluded. If this happens repeatedly to a same root, it can occur that that process first orders messages from some $vseq_2$, and only after that those for some $vseq_1 < vseq_2$. Through the subsequent asynchronous gossiping, processes in the subgroups of such roots can nevertheless receive the ordered messages from $vseq_1$ (possibly also from another root) early enough to APDELIVER them before those of $vseq_2$.

Broadcasters. To deal with a larger number of broadcasters without overloading roots, one can use processes at level $L-1$ for performing a primary ordering, and "feeding" level-L processes which compute the final order. The number of levels involved in the ordering can be even further increased to cope with a growing number of broadcasters. This however tends to increase the complexity in terms of message overhead.

Buffering. To increase the average probability of delivery α, broadcast messages gossiped about can be stored longer before delivery, in case broadcast messages with smaller sequence numbers have still not been received. This however comes at the expense of memory overhead.

Logging. As outlined previously, the problem of Atomic Broadcast has the nice property that it introduces a *continuum*, i.e., it produces an ongoing stream of messages, where missed messages can be immediately detected by observing sequence numbers associated with delivered messages. By logging messages à-la [1, 2], e.g., at roots, processes could then easily retrieve corresponding messages at these loggers. This implicitly introduces a *feedback loop* (cf. [25]), as high message loss rates (e.g., due to congestioned paths along the abstract hierarchy) lead to an increasing number of retransmission requests converging at roots, which can be turned into a positive effect by slowing these processes down *unintentionally* (they must handle these requests), and even *intentionally* (as high rates of retransmission requests can motivate them to take further actions, e.g., by alerting broadcasters).

Gossiping. Considering proven bounds on complexities of various gossip interaction styles [20], the dissemination algorithm presented in Section 6 might appear to be non-optimal. It could be improved by mixing the current *push* transmission with a *pull* scheme (gossip receivers query gossip senders for missed messages based on digests piggybacked by gossip messages) [20]. We have however refrained from modifying our prototype accordingly, as such a scheme does not marry well with selective gossiping (i.e., multicast, see Section 10.1).
For the same reason, the approach consisting in making use of a lightweight spanning-tree dissemination of messages and limiting gossiping to the propagation of message digests (as advocated by *pbcast* [3]) — indeed interesting in a broadcast setting — has not been adopted.

11 Related Work

We overview closest related work. Thereby, we distinguish between deterministic algorithms for atomic broadcast, and probabilistic ones.

11.1 Deterministic Algorithms

As a fundamental problem in distributed computing, much effort has been invested in solving atomic broadcast. Early work such as [9, 4, 22, 5] mostly focuses on stronger notions of Agreement (and also membership) than the one discussed in this paper. A good overview is given by [11].
As mentioned, our ordering algorithm is inspired by the one devised by Aguilera and Strom [1] in the context of the Gryphon publish/subscribe system. *Merger* nodes (similarly to our root processes) in an overlay graph deterministically merge broadcast message streams from concurrent broadcasters. The main difference between the pioneering work of Aguilera and Strom and ours, is that their work focuses on the deterministic merging itself, and such merging algorithms which are *economical* (no control messages sent from consumers to producers) and *message-oblivious* (the effective content of messages is not used for order determination). With assumptions such as FIFO reliable channels between parents and children, logging of messages, correct broadcasters/mergers, and recovery of crashed processes in the overlay graph, these properties are easily achieved, just like Agreeement and Total Order.
Our approach is also message-oblivious, but requires control messages to be sent to processes joining root groups. This occurs however rather seldom, and we have presented techniques to reduce the amount of this control information.

11.2 Probabilistic Algorithms

Our Atomic Probablistic Broadcast has been also strongly inspired by the many recent groundbreaking work on gossip-based broadcast algorithms.

Probabilistic Broadcast. The first probabilistic scheme for providing total order delivery is presented in the context of the almost legendary *pbcast* [3]. The algorithm assumes that processes can determine the number of gossip rounds needed for messages to reach all correct processes and the time it takes to execute such a round. To achieve total order, processes assign timestamps to the messages they broadcast and delay message delivery until any earlier messages have been received and delivered. Once a process determines that a round has terminated, it delivers all messages broadcast in the round in timestamp order.
Another mention of the use of *pbcast* in implementing ordered broadcast is made in [16], which presents a virtual synchronous (atomic) broadcast built on top of *pbcast* (*pbcast* is used to replace Reliable Broadcast). Unfortunately, both [3] and [16] lack precise definition of the guarantees achieved, and the scalability of the membership overhead is not addressed (by relying on a, though weakly consistent, complete membership).
The Reliable Probabilistic Broadcast (*rpbcast*) [27] algorithm has been devised, based on *pbcast*, in the context of the Gryphon project. *rpbcast* roughly adds message logging to *pbcast* to improve reliability, but, as its name suggests, does not consider ordering guarantees.

Probabilisitic Atomic Broadcast. Probabilistic Atomic Broadcast (*pabcast* [15]) is a more recent approach, which inherently aims at ordered delivery. In *pabcast* processes proceed in fully asynchronous rounds, during which each of them can cast a vote for a single received broadcast messsage. Once $n-f$ votes have been cast for a given message (f being the number of faulty processes tolerated), that message can be delivered. The basic algorithm described in [15] relies on a full membership, the assumption that processes broadcast at most one message per round, and on protocol messages piggybacking all previously delivered messages. Ways to circumvent the latter two restrictive assumptions are discussed, yet not included in the analysis. Furthermore, no buffer purging is described, meaning that broadcast messages which do not receive enough votes accumulate.
The specification of Probabilistic Atomic Broadcast associates individual probabilities ϕ_a, ϕ_v, and ϕ_o with the Agreement, Validity, and Total Order properties of Atomic Broadcast respectively. This is useful whenever one wants to know with what probability any *single* probability is fulfilled. However, no information is for instance given on the probability that Validity and Agreement are *both* fulfilled. Neither the probability of a perfect run (i.e., the satisfaction of all properties), nor that of an average run (i.e., the average fraction of processes in a group which behave as in a perfect run) is specified. These are inherently hard to specify, as a run does not define the behavior for a single message, since (the) Total Order (property) introduces dependencies between messages. This awakes serious doubts as to the usefulness of the specification of *pabcast*. This is somewhat confirmed by the fact that Validity and Agreement are both analytically captured in [15], but the probability associated with the Total Order property is only quantified through simulation results.

12 Concluding Remarks

In this paper, we have presented Atomic Probabilistic Broadcast (*apbcast*), a novel gossip-based algorithm for total order delivery of broadcast messages to processes in large groups, which is hybrid: it

combines the benefits of probabilistic, gossip-based, message propagation for achieving *scalability* and a high degree of *reliability* despite high rates of process failures, with those of deterministic ordering of messages for ensuring *consistency* with respect to the (total) delivery order of messages. This order is computed in a way offering *availability*, which can be balanced against overhead. These assets are mainly a consequence of the recursive group subdivision underlying *apbcast*, which also helps reducing the membership knowledge stored at individual processes.
We have presented evidence of the scalability and reliability of our *apbcast* algorithm through both analysis and simulation, which can also be used to adapt parameters of the algorithm. Last but not least, we have suggested several ways of tuning our basic algorithm for special application needs.
We are currently investigating the achievement of further guarantees (e.g., causal order) commonly associated with broadcast. Ultimately, our goal is to come up with a broadcast framework which is first of all modular, and in which algorithms, as illustrated by *apbcast*, provide only probabilistic guarantees where these are both necessary and useful.

13 Acknowledgements

We are very grateful to Patrick Berard, Patrick Bizeau, David Mayor, and Swati Rastogi for implementing the prototype of *apbcast*, as well as for conducting the experiments with that prototype.

References

[1] M.K. Aguilera and R.E. Strom. Efficient Atomic Broadcast Using Deterministic Merge. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC 2000)*, pages 209–218, July 2000.

[2] S. Bhola, R.E. Strom, S. Bagchi, Y. Zhao, and J.S. Auerbach. Exactly-once Delivery in a Content-based Publish-Subscribe System. In *Proceedings of the 2002 IEEE International Conference on Dependable Systems and Networks (DSN 2002)*, pages 7–16, June 2002.

[3] K.P. 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.

[4] K.P. Birman and T.A Joseph. Reliable Communication in Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[5] K.P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Multicast. *Transactions on Computer Systems*, 9(3):272–314, August 1991.

[6] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. 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)*, pages 219–227, July 2000.

[7] B. Charron-Bost, X. Défago, and A. Schiper. Broadcasting Messages in Fault-Tolerant Distributed Systems: the Benefit of Handling Input-Triggered and Output-Triggered Suspicions Differently. In *Proceedings of the 21st IEEE Symposium On Reliable Distributed Systems (SRDS'02)*, pages 244–249, October 2002.

[8] G.V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 4(33), December 2001.

[9] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Proceedings of the 15th IEEE International Symposium on Fault-Tolerant Computing (FTCS '85)*, pages 200–206, June 1985.

[10] Distributed Asychronous Computing Environment (DACE). http://www.d-a-c-e.com.

[11] X. Défago, A. Schiper, and P. Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. Technical Report IC/2003/56, Swiss Federal Institute of Technology in Lausanne, 2003.

[12] 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 ACM Symposium on Principles of Distributed Computing (PODC '87)*, pages 1–12, August 1987.

[13] O. Diekmann and J.A.P. Heesterbeek. *Mathematical Epidemiology of Infectious Diseases: Model Building, Analysis and Interpretation*. Wiley and Associates, 2000.

[14] P.Th. Eugster and R. Guerraoui. Probabilistic Multicast. In *Proceedings of the 2002 IEEE International Conference on Dependable Systems and Networks (DSN 2002)*, pages 313–323, June 2002.

[15] P. Felber and F. Pedone. Probabilistic Atomic Broadcast. In *Proceedings of the 21st IEEE Symposium On Reliable Distributed Systems (SRDS'02)*, pages 170–179, October 2002.

[16] I. Gupta, K.P. Birman, and R. van Renesse. Fighting Fire with Fire: Using Randomized Gossip to Combat Stochastic Scalability Limits. *Special Issue Journal Quality and Reliability Engineering International: Secure, Reliable Computer and Network Systems*, 18(3):165–184, May/June 2002.

[17] I. Gupta, R. van Renesse, and K.P. Birman. Scalable Fault-Tolerant Aggregation in Large Process Groups. In *Proceedings of the 2001 IEEE International Conference on Dependable Systems and Networks (DSN 2001)*, pages 433–442, June 2001.

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

[19] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcast and Related Problems. Technical report, Cornell University, Computer Science, May 1994.

[20] R. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking. Randomized Rumor Spreading. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2000)*, pages 565–574, November 2000.

[21] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus Deterministically Constrained Flooding on Small Networks. In *Proceedings of the 14th International Conference on Distributed Computing (DISC 2000)*, pages 253–267, October 2000.

[22] S.W. Luan and V.D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):271–285, 1990.

[23] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC-1771, http://www.ietf.org/rfc/rfc1771.txt, 1995.

[24] L. Rodrigues, H. Fonseca, and P. Verissimo. Totally Ordered Multicast in Large-Scale Systems. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems (ICDCS '96)*, pages 503–510, May 1996.

[25] L. Rodrigues, S.B. Handurukande, J. Pereria, R. Guerraoui, and A.-M. Kerrmarrec. Adaptive Gossip-Based Broadcast. In *Proceedings of the 2003 IEEE International Conference on Dependable Systems and Networks (DSN 2003)*, pages 47–56, June 2003.

[26] A. Schiper and A. Ricciardi. Virtually Synchronous Communication Based on a Weak Failure Suspector. In *Proceedings of the 23rd IEEE International Symposium on Fault-Tolerant Computing (FTCS '93)*, pages 534–543, June 1993.

[27] Q. Sun and D.C. Sturman. A Gossip-Based Reliable Multicast for Large-Scale High-Throughput Applications. In *Proceedings of the 2000 IEEE International Conference on Dependable Systems and Networks (DSN 2000)*, pages 347–358, July 2000.

[28] R. van Renesse. Scalable and Secure Resource Location. In *Proceedings of the 33rd IEEE Hawaii International Conference on System Sciences (HICSS-33)*, January 2000.