Probabilistic Atomic Broadcast

Pascal Felber[1], Fernando Pedone
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2002-69
March 20th , 2002*

E-mail: pascal@research.bell-labs@com, pedone@hpl.hp.com

Reliable distributed protocols, such as consensus and atomic broadcast, are known to scale poorly with large number of processes. Recent research has shown that algorithms providing probabilistic guarantees are a promising alternative for such environments. In this paper, we propose a specification of atomic broadcast with probabilistic liveness and safety guarantees. We present an algorithm that implements this specification in a truly asynchronous system (i.e., without assumptions about process speeds and message transmission times). Our algorithm tolerates a configurable number of crash failures, and delivers messages reliably and in order with high probability. Furthermore, even though some messages may be delivered out of order, processes can determine when this happens.

* Internal Accession Date Only Approved for External Publication
[1] Bell Laboratories, 600 Mountain Ave, 2B-303, Murray Hill, NJ 07920
© Copyright Hewlett-Packard Company 2002

1 Introduction

Message ordering abstractions, and more specifically group communication protocols, are very useful for the design of reliable distributed systems. Briefly speaking, message ordering abstractions ensure agreement on which messages are delivered in the system and on the order such messages are delivered. Many problems related to reliable and highly-available computation, such as active replication [Sch90], have been solved using one-to-many communication primitives with total-order guarantees.

Until recently, however, scalability has been the Achilles' heal of reliable one-to-many protocols. It has been shown (e.g., in [BHO+99]) that group communication protocols do not scale well past a couple of hundreds of processes and degrade rapidly when executed across wide-area networks. A promising approach for increasing scalability is to weaken the deterministic guarantees of the protocols to make them probabilistic. Provided that they are "adequately" high, probabilistic guarantees are sufficient for most applications. Actually, even deterministic protocols make implicit assumptions of probabilistic nature (e.g., failures are independent).

Several probabilistic protocols have been proposed to solve various group communication-related problems such as reliable broadcast and group membership. All the protocols we are aware of (e.g., [HB96, BHO+99]) are probabilistically live and deterministically safe. In this paper, we study the problem of probabilistic atomic broadcast and take into account not only probabilistic liveness but also probabilistic safety properties. We believe that many applications can take advantage of faster and more scalable algorithms without deterministic safety, if safety violations are infrequent and can be detected.

This paper makes the following contributions: First, we propose a probabilistic specification for atomic broadcast. Unlike other atomic broadcast specifications, ours provides probabilistic guarantees for both liveness and safety. Second, we present a protocol that implements probabilistic atomic broadcast. This protocol is resilient to message losses and f process failures, where f is a parameter of the protocol. Processes execute a sequence of rounds, and during a round they can vote for broadcast messages. Among the protocol features, messages that receive $f + 1$ votes in a round are delivered by all correct processes in the same order. We initially present a basic version of the protocol an then discuss how it can be extended in several ways. Finally, we analyze the probabilistic behavior of our protocol under various conditions.

Analytical and simulation results demonstrate that our protocol is highly reliable and scalable, and that the number of out-of-order messages is small in most scenarios.

The rest of this paper is organized as follows: Section 2 describes the system model. Section 3 defines the probabilistic atomic broadcast problem and presents an algorithm that solves it. Section 4 analyzes the probabilistic behavior of the protocol. Section 5 discusses related work, and Section 6 concludes the paper. Correctness proofs of all the propositions discussed in the paper are presented in the Appendix.

2 System Model

We consider a system composed of a finite set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate by message passing. The system is truly asynchronous, that is, there are no bounds on the time it takes for processes to execute operations, nor on the time it takes for messages to be transmitted. Processes can only fail by crashing (i.e., we do not consider Byzantine failures). A process that never fails is *correct*; processes that are not correct are *faulty*. For simplicity, we do not include process recovery in the model. We discuss this issue later in the paper (see Section 3.3).

Processes communicate using the primitives send(m) and receive(m). Communication links are *fair-lossy*, defined by the following properties:

- *Fair Loss*: If a process p sends a message m to a correct process q an infinite number of times, then q receives m from p an infinite number of times.
- *Finite Duplication*: If process p sends message m to process q a finite number of times, then q receives m from p a finite number of times.
- *No Creation*: If process q receives message m from process p at time t, then p sent m to q before t.

Fair-lossy links can lose messages; correct processes can construct reliable communication links on top of fair-lossy links by periodically retransmitting messages. If a correct process p keeps sending a message m to another correct process q, then q eventually receives m from p.

3 Probabilistic Atomic Broadcast

3.1 Problem Definition

In this section we introduce probabilistic atomic broadcast (PABCast). PABCast is defined by the primitives broadcast(m) and deliver(m), which guarantee *Agreement*, *Order*, *Validity*, and *Integrity*. The former three properties are probabilistic and the latter is deterministic. In the following, p and q are two processes in Π.

Probabilistic Agreement. If p delivers m, then with probability γ_a, q also delivers m.

Probabilistic Order. If p and q both deliver m and m', then with probability γ_o they do so in the same order.

Probabilistic Validity. If p broadcasts message m, then with probability γ_v, p delivers m.

Integrity. Every message is delivered at most once at each process, and only if it was previously broadcast.

PABCast generalizes the traditional atomic broadcast properties [HT93] to allow messages to be delivered by any subset of the processes (from probabilistic agreement), out of order (from probabilistic order), and not at all (probabilistic validity).[1] Probabilistic agreement and order are independent of each other, as illustrated in Figures 1 and 2.

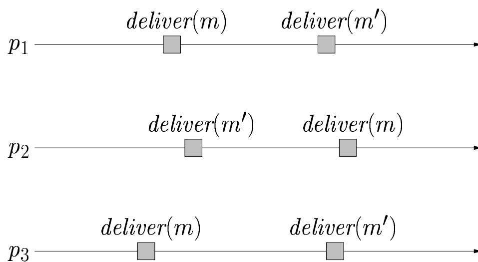

Figure 1: Run with agreement but no order

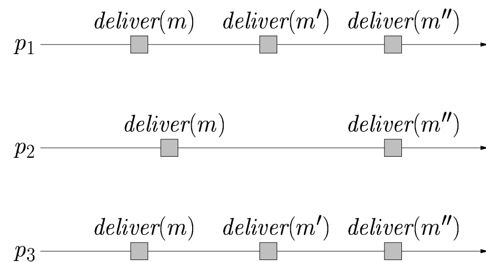

Figure 2: Run with order but no agreement

In the run depicted in Figure 1, all processes deliver messages m and m', but p_1 and p_3 deliver m before m' and p_2 delivers m' before m, thus, agreement is satisfied but order is not. In Figure 2, p_2 does not deliver m', but all processes deliver m before m'', and p_1 and p_3 deliver m, m', and m'' in the same order, so order is satisfied but agreement is not.

[1] Each one of our probabilistic properties is associated with a probability γ. When $\gamma = 1$, we actually have a deterministic property. Therefore, when we refer to deterministic agreement, for example, we have $\gamma_a = 1$.

3.2 Solving Probabilistic Atomic Broadcast

We present our PABCast algorithm incrementally. In this section we introduce a simple, but not very efficient, version of the algorithm. In Section 3.3, we discuss various improvements to the basic algorithm.

3.2.1 Basic Idea

Processes executing our PABCast algorithm proceed in a sequence of rounds $r_1, r_2, \dots$ Each process starts in round 0 and can broadcast at most one message per round. If p has broadcast a message in round r and wants to broadcast another message, p has to wait until round r has terminated (see Figure 3). Moreover, p can only deliver a message broadcast in round $r+1$ after it has terminated round r.

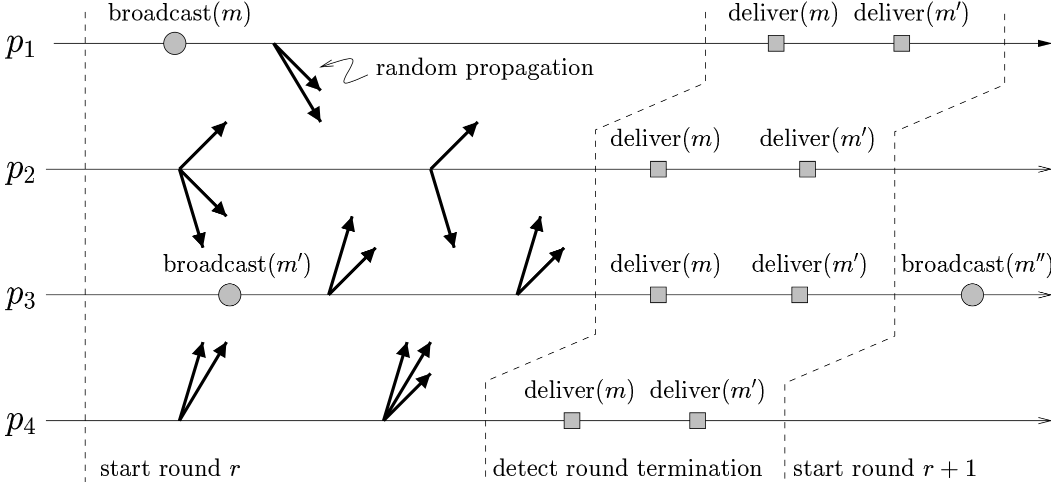

Figure 3: Algorithm execution (message reception not shown)

During the execution of a round, processes vote for broadcast messages. Each process can cast only one vote per round, either for the message it broadcast in the round or for the message some other process broadcast in the round. Each process p keeps an list of message votes per round (hereafter $list_p$). Each item in $list_p$ is a pair $(m, vSet)$, where m is a message broadcast during the current round and $vSet$ is the set of processes that p knows have voted for m. To simplify the algorithm, we assume that messages in $list_p$ can be ordered according to a unique identifier associated with each message. The message unique identifier is generated by the process broadcasting the message, which uses its process unique identifier (e.g., its IP address)

and a local sequential number, associated to each broadcast message.

When p starts round r and wants to broadcast m, it initializes $list_p$ with $\{(m, \{p\})\}$. Process p periodically chooses a random subset of processes to which it will send $list_p$. When process q receives $list_p$ from p, it updates its own list as follows:

- If q has not cast a vote in round r, that is, $list_q$ is empty, q initializes $list_q$ with $list_p$; then, it chooses the message m in $list_p$ with the smallest number of votes and casts a vote for m.
- If q has already voted in round r, that is, $list_q$ is not empty, q updates its list with the items in $list_p$: all votes in $list_p$ that are not in $list_q$ are copied into $list_q$.

Process p starts the termination of round r after it receives *directly* or *indirectly* $n - f$ votes cast in round r (remember that f is the number of processes that may fail in the system): q's vote is received directly by p if q sends a message to p with its vote; q's vote is received indirectly by p if p learns q's vote from some other process. To terminate round r, p delivers all messages received during r in a deterministic order, based on the unique identifier associated with each message. Then, p_i starts round $r + 1$ with an empty list of message votes ($list_p = \emptyset$).

Due to the asynchrony of the system and the possibility of message losses, it may happen that some process p executes in round r, while other processes execute in round $r' > r$. This situation may prevent p from terminating round r and proceeding to round $r + 1$, since p may need messages from processes that are no longer in round r. Thus, to ensure progress, processes also include in the messages they exchange, a sequence with the messages they have delivered in previous rounds. Whenever p in round r receives a message from q in round $r' > r$, p delivers the messages delivered by q that it has not yet delivered and jumps to round r'. We discuss ways to avoid sending all previously delivered messages in Section 3.3.

3.2.2 Detailed Algorithm

Figure 4 depicts the PABCast algorithm. Tasks 1, 2, and 3 execute concurrently, but there is only one instance of each task executing at a time. We assume that the task scheduler is fair, that is, all tasks get equal chances to execute. Moreover, each line is executed atomically. For example, the operations in line 12 cannot be interrupted.

Processes start a new round by setting the round number and creating an empty list of message votes for the round (lines 2, 12, and 30). To broadcast a message m, p includes m in its $broadcast_p$ sequence (line 6). Messages are appended to sequences with the concatenation operator $\oplus$. Messages in $broadcast_p$ are eventually gossiped to the other processes in the system.

When p receives a message from q (line 8), it delivers all messages delivered by q that it has not yet delivered (lines 9–11). If the message p receives from q is related to some round ahead of the round in which p executes, p jumps to this round (line 12). If p has just started the current round (i.e., $list_p^{r_p}$ is empty) (line 13), it can vote for a message: If p has broadcast some message m that has not been yet delivered, p will vote for m (lines 15 and 18); otherwise, p chooses the message in q's list of message votes that has received the smallest number of votes and votes for it (lines 17–18).

Then, p merges its message votes list with q's list (lines 19–24). Process p detects the end of round r_p by evaluating predicate $all_votes(p, r_p)$ (line 25), defined as:

$$all_votes(p,r) \stackrel{def}{=} \sum_{(-,vSet)\ \in\ list_p^r} |\, vSet\,|.$$

When p reaches the end of round r_p, it iterates through all the messages of $list_p$ and delivers each of them, if it has not yet done so (lines 26–29); p then starts a new round (line 30). Processes periodically choose a random subset of Π of size k (a parameter of the algorithm) and send to these processes their list of message votes for the current round (lines 32–34).

3.2.3 PABCast Algorithm Properties

We characterize next the PABCast algorithm by presenting some of its properties. Propositions 3.1 and 3.2 show that acquiring $f+1$ votes for two messages m and m' is a sufficient condition for having them delivered in the same order by all processes. Proposition 3.3 shows that eventually, a single vote cast for a message is enough to guarantee its delivery. Proposition 3.4 proves that the PABCast algorithm eventually becomes deterministic. These last two results hold after all faulty processes have crashed.[2] In the following, we provide only the proposition statements; the proofs can be found in the Appendix.

[2]Notice that this result is not only of theoretical interest but also practical: it shows that if only n-f processes participate in the execution of a round, broadcast messages are delivered in the same order by all processes that execute the round.

```
1: Initialization:

2:    r_p ← 0; list_p^{r_p} ← ∅                                   {processes start in round 0}
3:    broadcast_p ← ε                                              {sequence of locally broadcast messages}
4:    delivered_p ← ε                                              {sequence of all delivered messages}

5: To execute broadcast(m):                                        {Task 1}

6:    broadcast_p ← broadcast_p ⊕ ⟨ m ⟩                            {include m in broadcast_p sequence}

7: deliver(m) occurs as follows:                                   {Task 2}

8:    when receive (r_q, list_q^{r_q}, delivered_q) from q and r_q ≥ r_p   {when receive a message from q}
9:      for each m in delivered_q \ delivered_p, in sequence order do      {for each m delivered only by q:}
10:       deliver(m)                                               {deliver m and...}
11:       delivered_p ← delivered_p ⊕ ⟨m⟩                          {...keep track of it}

12:     if r_q > r_p then r_p ← r_q; list_p^{r_p} ← ∅              {if q is ahead of p, start new round}

13:     if list_p^{r_p} = ∅ then                                   {if hasn't cast a vote in r_p:}
14:       if broadcast_p \ delivered_p ≠ ε then                    {if has a message to broadcast:}
15:         let m be the first message in broadcast_p \ delivered_p   {select this message}
16:       else                                                     {else:}
17:         let (m, vSet) be the item in list_q^{r_q} with the smallest vSet   {select a message in list_q^{r_q}}
18:       list_p^{r_p} ← {(m, {p})}                                {vote for message m}

19:     for each (m_q, vSet_q) in list_q^{r_q} do                  {update list with votes}
20:       if (m_q, vSet_p) ∈ list_p^{r_p} then                     {if has seen message m_q:}
21:         list_p^{r_p} ← list_p^{r_p} \ {(m_q, vSet_p)}          {join votes for m_q}
22:         list_p^{r_p} ← list_p^{r_p} ∪ {(m_q, vSet_q ∪ vSet_p)}  {done}
23:       else                                                     {if hasn't seen m_q:}
24:         list_p^{r_p} ← list_p^{r_p} ∪ {(m_q, vSet_q)}          {make new entry in list of votes}

25:     if all_votes(p, r_p) ≥ n − f then                          {if collected enough votes to terminate:}
26:       for each m in list_p^{r_p}, in ID order do               {for each m in list...}
27:         if m ∉ delivered_p then                                {...that hasn't been delivered:}
28:           deliver(m)                                           {deliver it and...}
29:           delivered_p ← delivered_p ⊕ ⟨m⟩                      {...keep track of it}
30:       r_p ← r_p + 1; list_p^{r_p} ← ∅                          {start next round}

31: Random propagation of messages:                                {Task 3}

32:   periodically do
33:     fwdSet ← some random subset of Π of size k                 {choose set of receivers}
34:     for each q in fwdSet do send (r_p, list_p^{r_p}, delivered_p) to q   {forward list with votes}
```

Figure 4: Probabilistic atomic broadcast algorithm (for process p)

Proposition 3.1. *If message m has received $f+1$ votes in round r, then m is delivered by every process that terminates r.*

Proposition 3.2. *If m and m' are two messages that have received $f+1$ votes in rounds r and r', respectively, then all processes that deliver m and m' do so in the same order.*

Proposition 3.3. *After f processes fail, every message that receives a vote in round r is delivered by all correct processes.*

Proposition 3.4. *After f processes fail, every broadcast message is delivered by all correct processes and in the same order.*

3.3 Improving the PABCast Algorithm

We discuss next how to improve our PABCast algorithm in several aspects. In all cases, the basic idea of the algorithm, described in the previous section, remains the same.

Improvement #1: Reducing Propagation Delays

Each process p votes for a message by updating its list of message votes. This list is periodically sent by p to a random subset of processes upon execution of Task 3. Assuming that Task 3 is executed every δ milliseconds, it takes on average $\delta/2$ milliseconds between the time p casts a vote for a message m and the time this vote is propagated to other processes.

There are two problems with this propagation delay. First, it increases the delivery latency of m by $\delta/2$ on average, because processes will only have a chance to vote for m after they receive it. Second, the more a process waits to propagate the vote for m, the lower the chances that m will receive $f+1$ votes—the condition for deterministic agreement and ordering, as stated by Properties 3.1 and 3.2—since in the meanwhile processes may receive and vote for other messages.

The delay in the propagation of the votes can be suppressed by having processes execute Task 3 right after they vote for a message (line 18), in addition to the task's periodic execution.

Improvement #2: Increasing Throughput

The PABCast protocol only allows processes to broadcast one message per round. This limitation, which reduces the throughput of the system, can be addressed in several ways. First,

processes can bundle together several broadcast messages and vote for this sequence of messages as if it were a regular message.

The second approach to increase throughput is to let processes insert more than one message per round in their list of message votes: if p broadcasts a message m in round r and wants to broadcast another message, say m', before r is finished, p can add m' to $list_p$ with an empty vote set. If this happens early enough in the round, there are good chances that m' collects enough votes to be delivered by all processes at the end of round r. If there are no votes for m in $list_p$ at the end of round r, p will broadcast m again during round $r + 1$. Note that, while this approach increases the throughput of the PABCast algorithm, it also increases the changes of having out of order messages (see Section 4).

A third alternative to increase throughput is for processes to overlap round executions. Instead of executing rounds sequentially, processes can participate in multiple rounds at the same time. This requires each process to maintain a distinct list of message votes per round and to keep track of the last round ($last$) that has been terminated. Processes should also embed $last$ in every message sent to other processes. When receiving a message from process q, process p checks if $last_q$ is greater than $last_p$, and if so, p delivers the messages in $delivered_q$ and terminates round $last_q$. When p receives a message for a round in which it has not yet voted, it votes for some message in the round. As before, to deliver messages in round r, p has to wait until each previous round has terminated.

Improvement #3: Coping with Process Recovery

Process recovery can be integrated into the system as long as processes have access to some form of stable storage (e.g., disk). The key requirement is that once a process votes for a message in a round, it cannot forget for which message it voted and vote for a different message in the same round after recovery. So, in order to accommodate process recovery, before voting for a message, processes have to store their vote on stable storage. Moreover, to guarantee that messages are delivered at most once (Integrity property of PABCast), processes also have to "remember" which messages they have previously delivered after recovering from a crash.

Improvement #4: Reducing the Message Size

To prevent systematic gossipping of the *delivered* message sequence, processes can embed into gossip messages only the message identifiers of the messages delivered during the last few rounds. As in [BHO+99], a process p executing in round r that receives a message from process q related to round $r' \geq r$ can use the message identifiers to detect missing messages and request them from q. Therefore, processes do not always need to propagate the messages they have previously delivered.

Improvement #5: Deterministic Guarantees

Propositions 3.1 and 3.2 presented in Section 3.2.3 show that all it takes for messages to be delivered by all processes in the same order is to gather $f + 1$ votes. This suggests that before propagating messages to the whole system, processes could make sure that they will get so many votes. One way of doing that is to divide the system in groups $g_1, g_2, ...$ of size greater than f and equip processes in each group with a deterministic atomic broadcast protocol. The atomic broadcast, defined by the primitives a-broadcast and a-deliver, is only executed by the members of the group it belongs to.

To broadcast a message to the whole system, processes in group g a-broadcast m in g. This guarantees that all processes in g a-deliver messages in the same order, and can all cast their vote for the same messages. After a-delivering and casting a vote for a message, the protocol can continue as the basic PABCast protocol: processes propagate their votes and as soon as $n - f$ votes are received for a round, the round can be terminated. Since every message has at least $f + 1$ votes, it will be delivered by all processes in the same order. This scheme can co-exist with the original one described in the basic algorithm, allowing for deterministic and probabilistic guarantees in the system. For example, only some subsets of processes may be able to broadcast messages with deterministic guarantees.

This solution increases the delivery latency of messages—even though only for those messages with deterministic guarantees—but it is a powerful one since it does not depend directly on the size of the system (although one might argue that as n grows, f should grow as well). For a large-scale system, it also shows how local interactions can have an effect on the overall system.

4 Analysis

Our protocol provides probabilistic agreement, ordering, and validity. In the rest of this section, we discuss these aspects individually and present both theoretical and experimental results obtained from simulation. The diffusion of a message using gossiping follows complex mathematical models well studied in Epidemiology (see for instance [Bai57]). In the following, we focus on the probabilistic analysis of the asymptotic behavior of our protocol.

4.1 Probabilistic Model

For the probabilistic analysis of our algorithm, we assume that failures are independent. The probability of a message loss is smaller than the constant $P_{loss} > 0$ and not more than $f < n$ processes can fail. The probability of some process crashing is thus not higher than $P_{fail} = f/n$. The processes in $fwdSet$, the subset of Π to which a process gossips a message, are chosen randomly according to a uniform distribution. Since k, the size of $fwdSet$, is a parameter of the algorithm, each process has a probability k/n of being included in $fwdSet$.

4.2 Agreement

Probabilistic agreement states that, with probability γ_a, two correct processes deliver the same set of messages. To compute γ_a, we are interested in finding the scenarios where agreement is violated. We simplify the analysis by considering a single round and assuming that periodic gossiping (lines 32–34 in Figure 4) is performed synchronously, i.e., all processes gossip at about the same time. We call the synchronous sending of gossip messages by all processes a *gossip step*.

A message m sent by a process p during round r can be received by another process q in two ways: (1) as part of $list_p^r$ during round r, or (2) as part of $delivered_p$ during round $r' \geq r$. Both cases are triggered by the reception of a gossip message (line 8). We are therefore interested in computing γ_a as a function of the number of gossip steps after m has been sent.

Informally, a gossip message sent by some correct process p is received by another process q if the following conditions hold: (1) q is part of $fwdSet_p$, (2) the message is not dropped by the network, and (3) q does not fail. Thus, the probability P that q receives a message m during any step can be calculated as:

$$P = \frac{k}{n}(1 - P_{loss})(1 - P_{fail}) \tag{1}$$

Let $Q = 1 - P$ be the probability that q does *not* receive m during any given step. We denote by $P(s)$ the probability that some process has received a message m after s gossip steps, $Q(s)$ the probability that it did *not* receive m, and $N(s)$ the expected number of processes that have received m after s gossip steps.

For simplification, we conservatively assume that initially $N(0) = 0$ (in fact the sender of a message m already has a copy of m). After the first step, $P(1) = P$, $Q(1) = 1 - P$, and $N(1) = nP$. To compute the probabilities for subsequent steps, we should first note that for a process p not to receive a message m after s steps, p must not receive m neither during s, nor during any previous step. We can easily derive the following recursive relation for step s:

$$\begin{aligned} Q(s) &= Q^{N(s-1)}Q(s-1) \\ P(s) &= 1 - Q(s) \\ N(s) &= nP(s) \end{aligned} \tag{2}$$

Figures 5 and 6 show the expected behavior of message diffusion with $n = 100$, $P_{loss} = 0.05$, and $P_{fail} = 0.05$ computed using expressions in (1) and (2). The expected number of processes reached by a message m after s gossip steps converges to 100 at different speeds depending on the fanout value k. Similarly, the probability that all processes have received a message converges to 1 as the number of gossip steps grows.

As expected, the agreement probability γ_a eventually converges to 1, because processes keep on gossiping each message forever. In practice, a process p can stop sending some message m (i.e., garbage collect the messages in $delivered_p$) after m has been gossiped a certain number of times. In our example above, 5 gossip steps are sufficient for any $k \geq 5$.

4.3 Validity

Probabilistic validity requires that, with probability γ_v, a correct process p delivers each message m that it broadcasts. In the PABCast algorithm, the only scenario where p may not deliver m is if it never terminates round r during which m is broadcast.

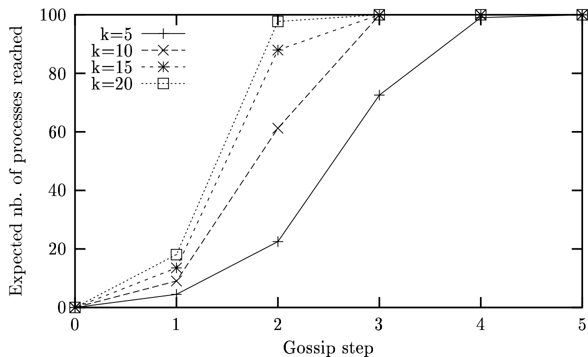

Figure 5: Expected number of processes that have received a message after s gossip steps

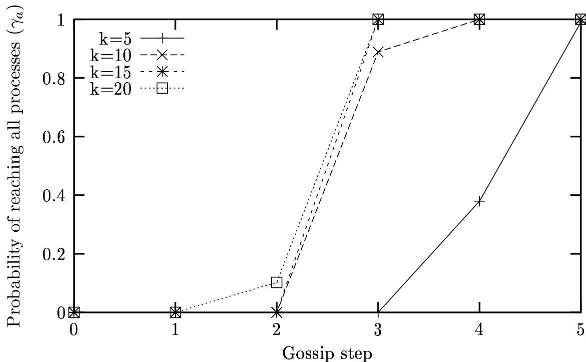

Figure 6: Probability that all processes have received a message after s gossip steps

A process p terminates round r when it receives $n-f$ votes during that round, or any message from round $r' > r$. To simplify, we pessimistically concentrate only on the first case and assume that a single message m is being broadcast during round r. For p to receive $n-f$ votes, $n-f$ processes must first receive m, and p must then receive the vote of all these processes. Similarly to the analysis of probabilistic agreement, we can compute a lower bound for γ_v as a function of the number of gossip steps after m has been sent.

Let $P(s)$ be the probability that some process p receives a gossip message from another process q after s steps—we have calculated this value in Section 4.2. The probability $P_v(s)$ that p receives a vote for m s steps after m has been broadcast is the probability that (1) some process has received m in less than s steps, and that (2) p has received the vote from that process during the remaining steps. This can be computed recursively as follows, with $P_v(0) = 0$:

$$P_v(s) = \sum_{i=0}^{s} P(i)P(s-i) - \sum_{i=0}^{s-1} P_v(i) \quad (3)$$

The probability $P_t(s)$ that p receives $n-f$ votes (i.e., that p terminates the current round) s steps after a single message m has been broadcast is thus $P_v(s)^{n-f}$. Figure 7 shows the values of $P_t(s)$ as a function of the number of gossip steps s, with $n = 100$, $P_{loss} = 0.05$, and $f = 2$. The probability of receiving $n-f$ votes converges to 1 at different speeds depending on the fanout value k. Note that $P_t(s)$ is a lower bound for γ_v: In practice γ_v will converge to 1 significantly faster because several messages can be sent concurrently and a process can terminate a round without waiting for $n-f$ votes (when receiving a message for a future round).

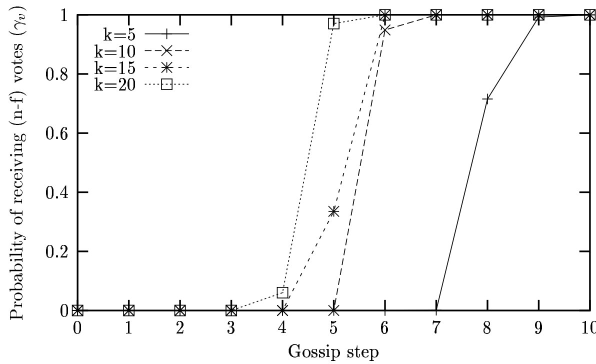

Figure 7: Probability that a processes receives $n - f$ votes after s gossip steps

4.4 Order

Probabilistic order specifies that, with probability γ_o, two processes p and q that both deliver two messages m and m' do so in the same order. In the algorithm of Figure 4, messages are delivered at lines 10 and 28. It is easy to see that if all the processes that execute line 28 during round r deliver the same messages in the same order, then no process can deliver these messages at lines 10 in a different order. Therefore, we are interested in computing the probability that ordering is violated at lines 28.

Processes use a deterministic function to order messages in $list$ (line 26), independent of the number of votes associated with each message. Thus, for messages to be delivered in a different order by two processes p and q, $list_p$ and $list_q$ must contain a different set of messages when they execute line 26. Since each process can only cast one vote, messages are guaranteed to be ordered if both p and q receive n votes. With $n - f$ votes however, up to f messages can be present in $list_p$ but not in $list_q$ (and vice versa).[3] Hence, the probability γ_o directly depends on the maximum number of failures f, as well as the number of messages B broadcast concurrently during a given round. In addition, the fanout k also influences γ_o, as the number of gossip steps required to obtain $n-f$ votes decreases when k grows, and fewer gossip steps increase the probability of having unordered messages.

We have built a simulation model of our protocol and conducted experiments to evaluate the probability of having out of order messages with different values for f, B, and k. Our simulator

[3] Note that, in that case, there are still chances that messages get "spontaneously" delivered in the same order.

models a distributed system with fair-lossy communication links. Processes are implemented as concurrent tasks, and gossip messages are sent at random intervals according to a uniform distribution. In the experiments, we set $n = 100$ and $P_{loss} = 0.05$. We did not inject failures when measuring γ_o because the probability of having out-of-order messages decreases when processes fail.

Figure 8 shows the simulation results obtained for different values of f and k, with $B = 2$. As expected, the number of unordered messages increases with the maximal number of failures. We also observed more unordered messages with larger fanout values (i.e., fewer gossip steps per round). In Figure 9, we have varied B and k, with $f = 5$. We observed a significant increase in the number of unordered messages with high values of B and k, reaching approximately 3% when broadcasting 10 messages simultaneously with a fanout of 15.

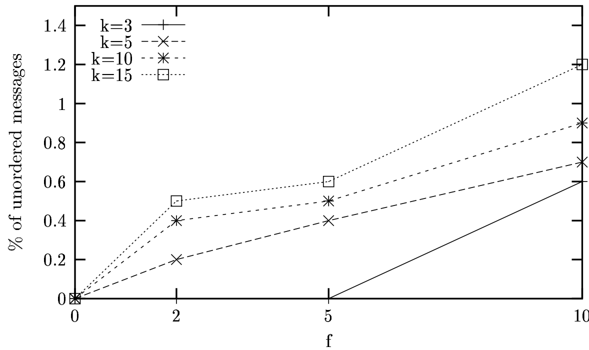

Figure 8: Unordered messages as a function of the number of failures f

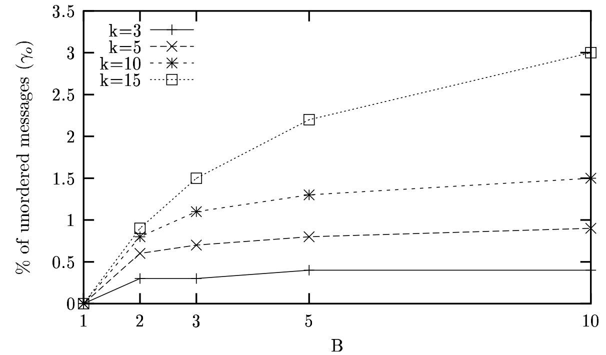

Figure 9: Unordered messages as a function of the number of concurrent messages B

These simulation results show that the number of unordered messages remains small when k and B are small. However, order seems to be more sensitive to the fanout than to the number of messages broadcast.

4.5 Scalability

Probabilistic protocols are known to generally scale better than deterministic protocols. In order to analyze how our protocol scales to large number of processes, we have computed the expected number of gossip steps required to reliably broadcast a message when increasing the number of processes in the system. For that purpose, we used the same diffusion model as in Section 4.2.

Figure 10 shows the number of gossip steps required to reach all processes with a probability of 0.99 as a function of the number of processes (represented on a logarithmic scale) for various fanout value. We have considered $P_{loss} = 0.05$ and $P_{fail} = 0.05$. The number of steps increases linearly with the logarithm of the number of processes, which demonstrates that our probabilistic broadcast algorithm scales well to very large numbers of processes.

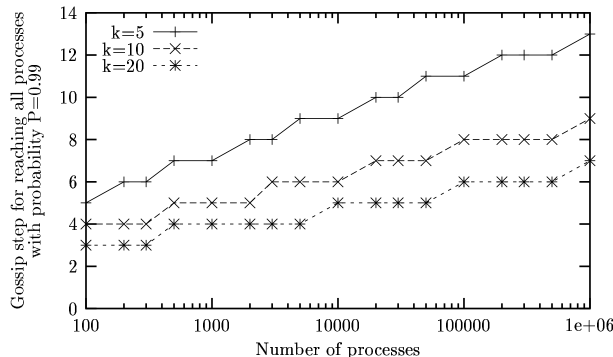

Figure 10: Expected number of gossip steps necessary to reach all processes (with a probability of 0.99) as a function of the number of processes n

5 Related Work

Epidemic protocols, also known as gossip protocols, were introduced in [DGH$^+$87] in the context of replicated database consistency management. More recently, the idea has been used to build failure detection mechanisms [RMH98, GCG01], garbage collection [GHvR$^+$97], leader election algorithms [IG00], and group communication protocols, as we review next.

In [BHO$^+$99], a gossip-based mechanism is proposed to implement reliable broadcast in large networks. The protocol proceeds in two phases: in the first phase, processes transmit messages using unreliable gossip-based dissemination; in the second phase, message losses are detected and, if present, repaired with re-transmissions. Several other papers have considered probabilistic approaches to solving reliable broadcast [LM99, SS00].

The work in [LMM99] discusses ways to reduce the number of gossip messages exchanged between processes. The idea is to make processes communicate according to a pre-determined graph with minimal connectivity guarantees to attain a desired level of reliability. Other than

message overhead, these protocols present similar characteristics to traditional gossip-based approaches. More recently, [KGHK01] has presented heuristics to garbage collect messages in gossip-based broadcast algorithms. The approach aims to identify "aging" buffered messages.

Group membership issues in a gossip-based reliable broadcast protocol are discussed in [KMG01] and [EHG+01]. The idea is to provide processes with a partial view of the membership of the system, which will be used to propagate the broadcast messages in the gossip phase of the algorithm. The problem solved in [KMG01] and [EHG+01] is orthogonal to the problem addressed in this paper; an interesting open question is how one could adapt the PABCast algorithm to run on top of such a membership service.

The only probabilistic atomic broadcast algorithm we are aware of is the one presented in [HB96]. As in [BHO+99], the execution proceeds in rounds.[4] The protocol assumes that processes can correctly estimate the number of rounds needed for messages to reach all non-faulty processes and the time it takes to execute such a round. To achieve total order, processes delay delivering a message until any earlier messages have been delivered. Processes assign timestamps to the messages they broadcast and, once a process determines that some round has terminated, it delivers all messages broadcast in the round following the timestamp order.

Our work is different from the one in [HB96] in two aspects. First, contrary to [HB96], we solve probabilistic atomic broadcast in a truly asynchronous model—with no assumptions on transmission delays and round duration—and we discuss how to integrate recovering processes in the algorithm. Second, our PABCast algorithm provides probabilistic guarantees for both message diffusion and ordering, with the option of making ordering deterministic, although at a higher cost.

6 Conclusion

In this paper, we have addressed the problem of scalable message-ordering abstractions for distributed systems. Recent research has shown that probabilistic protocols can scale significantly better than protocols that offer deterministic guarantees. We have initially proposed a specification of probabilistic atomic broadcast. Unlike previous work in this field, our specification does not only consider probabilistic liveness properties, but also probabilistic safety properties.

[4] Note that rounds in [HB96] are actually gossip steps in our terminology.

We have presented a protocol that implements probabilistic atomic broadcast in a truly asynchronous system. This protocol tolerates a configurable number of crash failures, and delivers messages reliably and in order with high probability. We have proposed extensions to the original protocol to overcome some of its shortcomings, and have analyzed its behavior under various conditions. Analytical and simulation results demonstrate that high reliability and scalability can be achieved and that the number of out-of-order messages is small in most scenarios.

Probabilistic protocols are a promising approach to increasing the scalability of distributed systems. While offering weaker guarantees than deterministic protocols, they are still of practical interest if these guarantees can be quantified and shown to be small. We believe that by carefully balancing between probabilistic liveness and safety guarantees, we can build other lightweight probabilistic protocols of broad interest for highly-scalable distributed computing.

References

[Bai57] N.T.J. Bailey. *The Mathematical Theory of Epidemics.* Charles Griffin & Company Limited, 1957.

[BHO+99] 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.

[DGH+87] 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*, pages 1–12, Vancouver, BC, Canada, August 1987.

[EHG+01] P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, Newport, Rhode Island, USA, August 2001.

[GCG01] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, Newport, Rhode Island, USA, August 2001.

[GHvR+97] K. Guo, M. Hayden, R. van Renesse, W. Vogels, and K. P. Birman. GSGC: An efficient gossip-style garbage collection scheme for scalable reliable multicast. Technical Report TR97-1656, Cornell University, Computer Science, December 1997.

[HB96] M. Hayden and K. Birman. Probabilistic broadcast. Technical Report TR96-1606, Cornell University, Computer Science, September 1996.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.

[IG00] K. P. Birman I. Gupta, R. van Renesse. A probabilistically correct leader election protocol for large groups. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 89–103, Toledo, Spain, October 2000.

[KGHK01] P. Kouznetsov, R. Guerraoui, S.B. Handurukande, and A.-M. Kermarrec. Reducing noise in gossip-based reliable broadcast. In *Proceedings of the 20th International Symposium on Reliable Distributed Systems*, pages 186–189, New Orleans, LA, USA, October 2001.

[KMG01] A.-M Kermarrec, L. Massoulie, and A.J. Ganesh. Probabilistic reliable dissemination in large-scale systems. Technical report, Microsoft Research, June 2001.

[LM99] 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.

[LMM99] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministic flooding: Low message overhead and high reliability for broadcasting on small networks. Technical Report CS1999-0637, University of California, San Diego, Computer Science and Engineering, November 1999.

[RMH98] R. Van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report TR98-1687, Cornell University, Computer Science, May 1998.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SS00] Q. Sun and D. Sturman. A gossip-based reliable multicast for large-scale high-throughput applications. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, New York (USA), June 2000.

Appendix: Proofs

Proposition 3.1. *If message m has received $f+1$ votes in round r, then m is delivered by every process that terminates r.*

PROOF (SKETCH): There are two ways p can terminate round r: (a) p receives $n-f$ votes related to r; and (b) p receives a message from some process in round $r' > r$. Case (a): Processes can only vote once in a round. So, if m receives $f+1$ votes in r, and p receives $n-f$ votes in r, p receives at least one vote for m, $(m,-) \in list_p^r$, and p delivers m (line 28). Case (b): Let q be the process p that receives the message from. To move from round r to round $r+1$, at least one process has to execute line 30, after receiving $n-f$ votes related to round r. Without loss of generality, assume q is that process. From case (a), q delivered m and included it in $delivered_q$. So, after p receives message $(r', list_q^{r'}, delivered_q)$ from q and before it moves to round r', p delivers all the messages in $delivered_q$ that it has not yet delivered. Thus, p delivers m. □

Proposition 3.2. *If m and m' are two messages that have received $f+1$ votes in rounds r and r', respectively, then all processes that deliver m and m' do so in the same order.*

PROOF (SKETCH): There are two cases to be considered: (a) $r = r'$ and (b) $r \neq r'$. Case (a): Let p and q be two processes that terminate r after receiving $n-f$ votes. Since m and m' have received $f+1$ votes and using Proposition 3.1, p and q have received at least one vote for m and m' and will order them deterministically according to their identifiers, include them in their *delivered* sequence, and deliver them in that order. Assume now that p terminates round r after receiving some message from q related to round $r'' > r$. In that case, p delivers m and m' according to their ordering in $delivered_q$, i.e., in the same order as q and all processes that have received $n-f$ votes in round r (if q did not receive $n-f$ votes in round r, then we can apply the same proof recursively to demonstrate that q has delivered m and m' in the same order as some process that has received $n-f$ votes in round r). Case (b): Assume without loss of generality that $r < r'$. If m and m' have received $f+1$ votes and using Proposition 3.1, m will be delivered by all processes that terminate r and m' by all processes that terminate r'. From the algorithm, round r' cannot terminate before round r terminates, and thus all processes that deliver m' at the end of round r' must have previously delivered m at the end of round r. □

Proposition 3.3. *After f processes fail, every message that receives a vote in round r is deliv-*

ered by all correct processes.

Proof (sketch): Let $t' < t$ be the time after which f processes have failed. Let r be a round that starts at time t. Since there are only $n - f$ processes in execution, all correct, a process p can only terminate r after receiving a vote from each correct process. Thus, every message that has received a vote in r is received by p and p delivers all such messages. Let q be a process that terminates r by receiving from p a message for round $r' > r$. Before jumping to r', q delivers all messages delivered by p during round r. □

Proposition 3.4. *After f processes fail, every broadcast message is delivered by all correct processes and in the same order.*

Proof (sketch): Let $t' < t$ be the time after which f processes have failed. From Proposition 3.3, any two messages m and m' that receive at least one vote each during round r and r', respectively, are delivered by all correct processes. If $r = r'$, since there are only $n - f$ processes in execution, m and m' are delivered in the same order by all processes that receive $n - f$ votes during round r, as well as processes that terminate r when receiving a message for round $r'' > r$. If $r' \neq r$, since all messages for round r (r') are delivered before moving to round $r + 1$ ($r' + 1$), m and m' are delivered in the same order by all correct processes that terminate both round r and r'. □

Proposition 3.5. *(Integrity.) Every message is delivered at most once at each process, and only if it was previously broadcast.*

Proof (sketch): It follows immediately from the algorithm that only broadcast messages can be delivered. In addition, before delivering any messages (lines 10 and 28), processes make sure that these messages are not already in their *delivered* sequence (lines 9 and 27). Thus, messages are never delivered more than once. □