Probabilistic Atomic Broadcast

Pascal Felber
Institut EURECOM
2229 route des Crêtes, BP 193
06904 Sophia Antipolis, France
pascal.felber@eurecom.fr

Fernando Pedone
Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA
Swiss Federal Institute of Technology
CH-1014, Lausanne, Switzerland
fernando.pedone@epfl.ch

Abstract

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).

1. Introduction

Message ordering abstractions, also known as group communication protocols, are very useful for the design of reliable distributed systems. 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 [16], 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 [2]) 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 enough 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 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 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, in ours both safety and liveness are probabilistic. Second, we present a protocol that implements probabilistic atomic broadcast. This protocol is resilient to message loses and f process failures, where f is a parameter of the protocol. Processes execute a sequence of rounds, during which 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. 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, and Section 5 discusses related work. Finally, Section 6 concludes the paper.

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*: (a) if p sends m to a correct process q an infinite number of times, q receives m from p an infinite number of times, (b) if p sends m to q a finite number of times, q receives m from p a finite number of times, and (c) if q receives m from p at time t, p sent m to q before t.

Even though 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. Let p and q be correct. 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 is correct and 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 [8] 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.

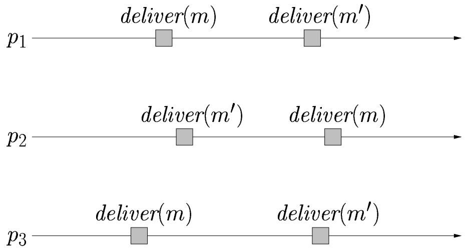

Figure 1. Agreement but no order

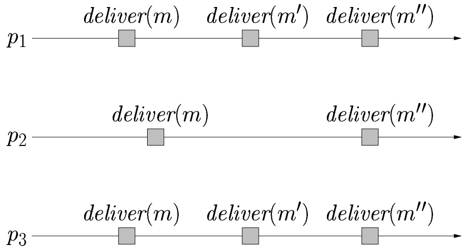

Figure 2. Order but no agreement

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 the run depicted in Figure 2, p_2 does not deliver m', but all three processes deliver m before m'', and p_1 and p_3 deliver m, m', and m'' in the same order, thus 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.

Basic Idea. Processes executing our PABCast algorithm proceed in a sequence of rounds $r_1, r_2, ...$ 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. Figure 3 depicts an execution of the algorithm (message reception is not shown). Moreover, p can only deliver a message broadcast in round $r+1$ after it has terminated round r.

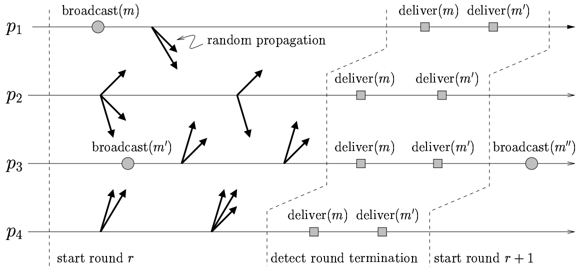

Figure 3. Algorithm execution

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. 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 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 unique identifier 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 (i.e., $list_q$ is empty), q initializes $list_q$ with $list_p$, chooses the message in $list_p$ with the smallest number of votes, and casts a vote for it.
- If q has already voted in round r (i.e., $list_q$ 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 $list_p$ set.

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 may prevent p from making progress because to terminate round r and proceed to round $r+1$, p may need messages from processes that are no longer in round r. 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.

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.

```
 1: Initialization:

 2:     $r_p \leftarrow 0$; $list_p^{r_p} \leftarrow \emptyset$                    {processes start in round 0}
 3:     $broadcast_p \leftarrow \epsilon$                                            {sequence of locally broadcast messages}
 4:     $delivered_p \leftarrow \epsilon$                                            {sequence of all delivered messages}

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

 6:     $broadcast_p \leftarrow broadcast_p \oplus \langle m \rangle$               {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 $r_q \geq r_p$   {when receive a message from $q$}
 9:         for each $m$ in $delivered_q \setminus delivered_p$, in sequence order do  {for each $m$ delivered only by $q$:}
10:             deliver($m$)                                                         {deliver $m$ and...}
11:             $delivered_p \leftarrow delivered_p \oplus \langle m \rangle$       {...keep track of it}

12:         if $r_q > r_p$ then $r_p \leftarrow r_q$; $list_p^{r_p} \leftarrow \emptyset$   {if $q$ is ahead of $p$, start new round}

13:         if $list_p^{r_p} = \emptyset$ then                                      {if hasn't cast a vote in $r_p$:}
14:             if $broadcast_p \setminus delivered_p \neq \epsilon$ then          {if has a message to broadcast:}
15:                 let $m$ be the first message in $broadcast_p \setminus delivered_p$   {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:             $list_p^{r_p} \leftarrow \{(m, \{p\})\}$                           {vote for message $m$}

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

25:         if $all\_votes(p, r_p) \geq n - f$ 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 $m \notin delivered_p$ then                                  {...that hasn't been delivered:}
28:                     deliver($m$)                                                 {deliver it and...}
29:                     $delivered_p \leftarrow delivered_p \oplus \langle m \rangle$   {...keep track of it}
30:             $r_p \leftarrow r_p + 1$; $list_p^{r_p} \leftarrow \emptyset$      {start next round}

31: Random propagation of messages:                                                  {Task 3}

32:     periodically do
33:         $fwdSet \leftarrow$ some random subset of $\Pi$ 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)

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 14–15 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:

$$all\_votes(p, r) \overset{def}{=} \sum_{(-, vSet) \,\in\, list_p^r} |\, vSet \,|.$$

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).

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. In the following, we provide only the proposition statements; the proofs can be found in [5].

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 improvements to PABCast.

#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. First, the delivery latency of m is increased 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 mean time 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.

#2: Increasing Throughput. PABCast only allows processes to broadcast one message per round, reducing the throughput of the system. This limitation can be addressed in several ways. First, processes can bundle several broadcast messages and vote for all of them as if they were a single 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 m' has no vote at the end of round r, p will broadcast it 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 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 $last_p$; if so, p delivers the messages in $delivered_q$ and terminates round $last_q$. When p receives a message for a round it has not yet voted, it votes for some message in the round. As before, to deliver messages in a round, p has to wait until each previous round has terminated.

#3: Coping with Process Recovery. Process recovery requires processes to have access to stable storage (e.g., disk). Once a process votes for a message in a round, it should not forget for which message it voted and vote for a different one 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.

#4: Reducing the Message Size. To prevent processes from systematically sending the sequence of all the messages they have previously delivered, a mechanism similar to the one described in [2] can be used: If process p executing in round r receives a message from process q related to round $r' > r$, p requests to q the messages in $delivered_q$, or a subset of them. Therefore, processes do not always need to propagate the messages they have previously delivered.

#5: Deterministic Guarantees. Propositions 3.1 and 3.2 show that all what it takes for messages to be delivered in the same order is to gather $f+1$ votes. Thus, before propagating messages to the whole system, processes could make sure that they will get so many votes. One way of doing this is to divide the system in groups 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. Thus, all processes in g a-deliver messages in the same order, and can cast their vote for the same messages. After a-delivering and casting a vote for a message, the protocol continues as the basic PABCast protocol: processes propagate their votes and as soon as $n - f$ votes are received for a round, the round terminates. 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 one described in the basic algorithm, allowing for deterministic and probabilistic guarantees (e.g., only some subsets of processes can 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

The diffusion of a message using gossiping follows complex mathematical models well studied in Epidemiology (see for instance [1]). In the following, we focus only 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 including in $fwdSet$.

4.2. Agreement

Probabilistic agreement states that, with a given 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 assuming that periodic gossiping (lines 32–34 in Figure 4) is performed synchronously, i.e., all processes gossip at the same time. We call the synchronous sending of gossip messages by all process 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.

Note that since all gossip messages contain the list of all messages delivered by a process (*delivered*), the probability of agreement will eventually converge to 1. In practice, however, *delivered* will be bounded and older messages will be deleted after a number of gossip steps. The probabilistic analysis of γ_a can help determine when to perform such a garbage collection.

Informally, a gossip message sent by some correct process p is received by another process q if (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:

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

Let $Q = 1 - P$ be the probability that q does *not* receive m during any 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.

We conservatively assume that initially $N(0) = 0$ (in fact the sender of m has a copy of m in $s = 0$). After the first step, $P(1) = P$, $Q(1) = 1 - P$, and $N(1) = nP$. To compute the probabilities for subsequent steps, we note that for a process not to receive a message m after s steps, it must not receive m in s nor in any previous step. We derive the following recursive relation for step s:

$$\begin{array}{rcl} Q(s) & = & Q^{N(s-1)}Q(s-1) \\ P(s) & = & 1 - Q(s) \\ N(s) & = & nP(s) \end{array} \qquad (2)$$

Figures 5 and 6 show the expected behavior of message diffusion with $n = 100$, $P_{loss} = 0.05$, and $P_{fail} = 0.05$. 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, 5 gossip steps are sufficient for any $k \geq 5$.

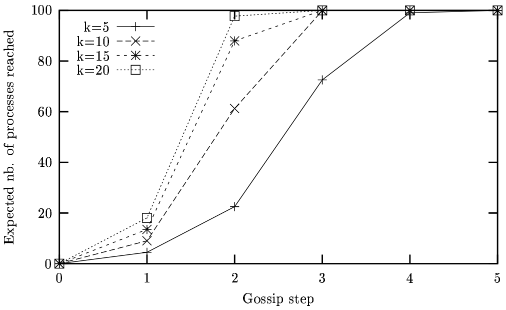

Figure 5. Number of processes that received a message after s gossip steps

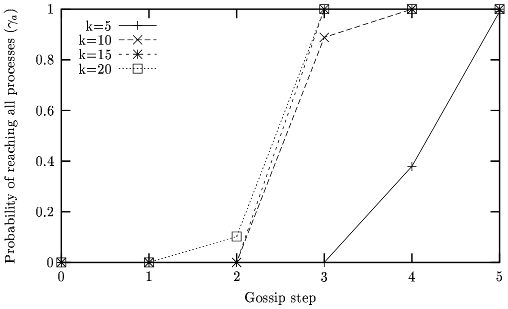

Figure 6. Probability that all processes received a message after s gossip steps

4.3. Validity

In PABCast, the only scenario where p may not deliver a message m is if the round r during which m is broadcast never terminates.

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 complement of the probability that p does not receive such a vote in s steps:

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

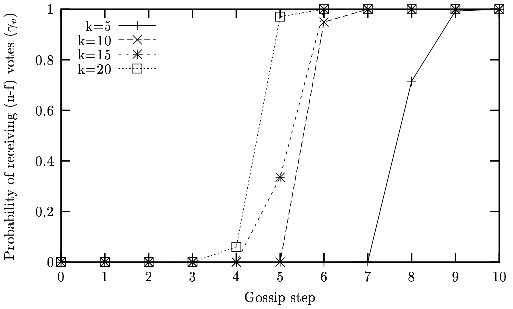

Figure 7. Probability that a processes receives $n - f$ votes after s gossip steps

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 send concurrently and a process can terminate a round without waiting for $n - f$ messages,.

4.4. Order

Messages can be delivered at lines 10 and 28 in PABCast. 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 order is violated at line 28.

Processes use a deterministic function to order messages (line 26), independent of the number of votes associated with the messages. So, for messages to be delivered in a different order by 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 in $list_p$ but not in $list_q$ (and vice-versa).[2] Hence, the probability γ_o directly depends on the maximum number of failures f and on 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 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 $n = 100$, $P_{loss} = 0.05$, and $f = 0$. We did not consider 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.

4.5. Scalability

In order to analyze how our protocol scales, we 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

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

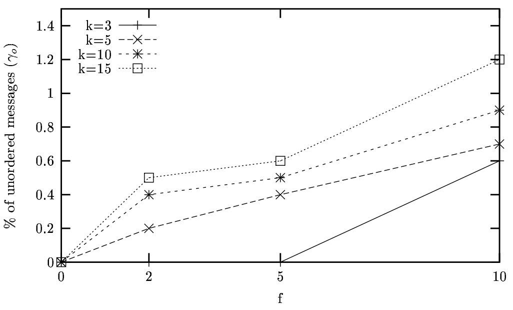

Figure 8. Unordered messages as a function of the number of failures f

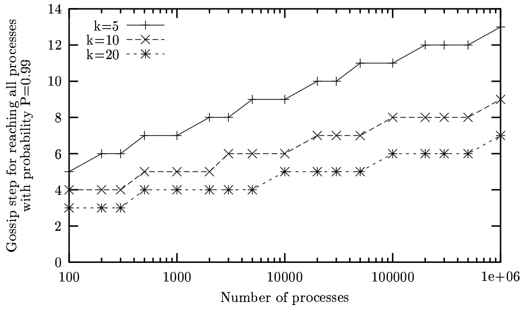

Figure 10. Number of gossip steps to reach all processes (with probability 0.99)

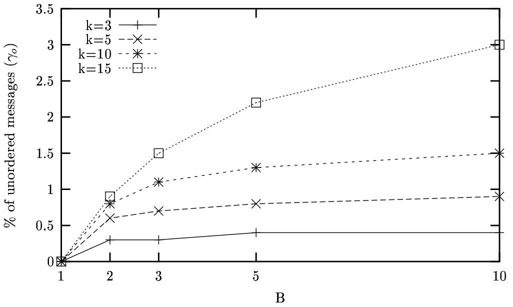

Figure 9. Unordered messages as a function of concurrent messages B

0.99 for various fanout value k as a function of the number of processes (represented on a logarithmic scale), with $P_{loss} = 0.05$ 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.

5. Background and Related Work

Epidemic protocols, also known as gossip protocols, were introduced in [3] in the context of replicated database consistency management. More recently, the idea has been used to build failure detection mechanisms [7, 15], garbage collection [6], leader election algorithms [10], and group communication protocols, as we review next.

In [2], a gossip-based mechanism is proposed to implement reliable broadcast in large networks. The protocol proceeds in two phases: in the first phase, processes use an unreliable gossip-based dissemination of information to transmit messages; in the second phase, messages losses are detected and repaired with re-transmissions. Several other papers have considered probabilistic approaches to solve reliable broadcast [13, 17].

The work in [14] discusses ways to reduce the number of gossip messages exchanged between processes. Processes communicate according to a pre-determined graph with minimal connectivity to attain a desired level of reliability. More recently, [12] 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 discusses in [4] and [11]. 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 [4] and [11] 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 [9]. As in [2], the execution proceeds in rounds—the notion of round in [9] is that of a gossip-like propagation of messages, and so, it differs from the PABCast rounds. The protocol assumes that processes can determine the number of rounds needed for messages to reach all correct 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. Once a process determines that a round has terminated, it delivers all messages broadcast in the round in timestamp order.

Our work is different from the one in [9] in several aspects. First, we solve probabilistic atomic broadcast in a truly asynchronous model and discuss how to integrate recovering processes in the algorithm. Second, our algorithm allows for probabilistic and deterministic message delivery in the same execution. Finally, our protocol exhibits the unique property that eventually it becomes deterministic—even though such a property is more of theoretical than practical interest, since it only holds after all faulty processes have crashed.

6. Conclusion

This paper addresses the scalability of message-ordering group communication protocols. We propose a specification of probabilistic atomic broadcast with probabilistic safety and liveness properties, present a basic probabilistic atomic broadcast protocol, and extend it to overcome some shortcomings. The probabilistic behavior of our protocol is analyzed under various conditions. Analytical and simulation results demonstrate that high reliability and scalability can be achieved. More specifically, results show that the number of out-of-order messages is small in most scenarios.

References

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

[2] 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.

[3] 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, Aug. 1987.

[4] 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, Aug. 2001.

[5] P. Felber and F. Pedone. Probabilistic atomic broadcast. Technical report, Bell Labs, Lucent, Dec. 2001. Also appears as Hewlett-Packard Technical Report HPL-2002-69, 2002.

[6] 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, Dec. 1997.

[7] 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, Aug. 2001.

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

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

[10] K. P. B. 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, Oct. 2000.

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

[12] P. Kouznetsov, R. Guerraoui, S. 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, Oct. 2001.

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

[14] 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, Nov. 1999.

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

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

[17] 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.