Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems

Luís RODRIGUES  
Universidade de Lisboa, Portugal  
ler@di.fc.ul.pt

Michel RAYNAL  
IRISA, Université de Rennes 1, France  
raynal@irisa.fr

Abstract

Atomic Broadcast is a fundamental problem of distributed systems: it states that messages must be delivered in the same order to their destination processes. This paper describes a solution to this problem in asynchronous distributed systems in which processes can crash and recover.

A Consensus-based solution to Atomic Broadcast problem has been designed by Chandra and Toueg for asynchronous distributed systems where crashed processes do not recover. Although our solution is based on different algorithmic principles, it follows the same approach: it transforms any Consensus protocol suited to the crash-recovery model into an Atomic Broadcast protocol suited to the same model. We show that Atomic Broadcast can be implemented without requiring any additional log operations in excess of those required by the Consensus. The paper also discusses how additional log operations can improve the protocol in terms of faster recovery and better throughput.

1 Introduction

Atomic Broadcast is one of the most important agreement problems encountered in the design and in the implementation of fault-tolerant distributed systems. This problem consists in providing processes with a communication primitive that allows them to broadcast and deliver messages in such a way that processes agree not only on the set of messages they deliver but also on the order of message deliveries. Atomic Broadcast (sometimes called Total Order Broadcast) has been identified as a basic communication primitive in many systems [12]. It is particularly useful to implement fault-tolerant services by using software-based replication [6]. By employing this primitive to disseminate updates, all correct copies of a service deliver the same set of updates in the same order, and consequently the state of the service is kept consistent.

Solutions to the Atomic Broadcast problem in asynchronous systems prone to process crash (no-recovery) failures are now well known [3, 5, 13]. In this model process crashes are definitive (i.e., once crashed, a process never recovers), so, a failed process is a crashed process. Unfortunately, the crash-no recovery model is unrealistic for the major part of applications. That is why, in this paper, we consider the more realistic crash-recovery model. In this model, processes can crash and later recover. We assume that when a process crashes (1) it loses the content of its volatile memory, and (2) the set of messages that has been delivered while it was crashed is also lost. This model is well-suited to feature real distributed systems that support user applications. Real systems provide processes with stable storage that make them able to cope with crash failures. A stable storage allows a process to log critical data. But in order to be efficient, a protocol must not consider all its data as critical and must not log a critical data every time it is updated (the protocol proposed in this paper addresses these efficiency issues).

It has been shown in [3] that Atomic Broadcast and Consensus are equivalent problems in asynchronous systems prone to process crash (no-recovery) failures. The Consensus problem is defined in the following way: each process proposes an initial value to the others, and, despite failures, all correct processes have to agree on a common value (called decision value), which has to be one of the proposed values. Unfortunately, this apparently simple problem has no deterministic solution in asynchronous distributed systems that are subject to even a single process crash failure: this is the so-called Fischer-Lynch-Paterson’s (FLP) impossibility result [4]. The FLP impossibility result has motivated researchers to find a set of minimal assumptions that, when satisfied by a distributed system, makes Consensus solvable in this system. The concept of unreliable failure detector introduced by Chandra and Toueg constitutes an answer to this challenge [3]. From a practical point of view, an unreliable failure detector can be seen as a set of oracles: each oracle is attached to a process and provides it with information regarding the status of other processes. An oracle can make mistakes, for instance, by not suspecting a failed
process or by suspecting a not failed one. Although failure
detectors were originally defined for asynchronous systems
where processes can crash but never recover, the concept
has been extended to the crash-recovery model [1, 9].

Chandra and Toueg have also shown how to transform
any Consensus protocol into an Atomic Broadcast proto-
col in the asynchronous crash (no-recovery) model [3]. In
the present paper we follow a similar line of work and
we show how to transform a protocol that solves Consen-
sus into the crash-recovery model in a protocol that solves
Atomic Broadcast in the same model. Thus, our protocol
assumes a solution to the Consensus problem in the crash-
recovery model (such protocols are described in [1, 9]).
Our transformation owns several interesting properties. In
first place, it does not require the explicit use of failure de-
tectors (although those are required to solve the Consen-
sus problem). Thus, it is not bound to any particular fail-
ure detection mechanism. It relies on a gossip mechanism
for message dissemination, avoiding the problem of reliable
multicast in the crash-recovery model. Also, it allows re-
covering processes to skip over Consensus executions that
already have a decided outcome. Additionally, our solu-
tion is non-blocking [2], i.e., as long as the system allows
Consensus to terminate the Atomic Broadcast is live. Fi-
ally, but not the least, we show that Atomic Broadcast can
be implemented without requiring any additional log oper-
ations in excess of those required by the Consensus. Thus,
our protocol is optimal in terms of number of log operations.

Chandra-Toueg’s approach and ours are similar in the
sense that both of them transform a Consensus protocol into
an Atomic Broadcast protocol. But, as they consider differ-
ent models (crash-no recovery and crash-recovery, respec-
tively), they are based on very different algorithmic prin-
ciples. This come from the fact we have to cope with pro-
cess crashes and message losses (that is why our protocol
requires a gossiping mechanism, which is not necessary in
a crash-no recovery + no message loss model). Actually,
when solving a distributed system problem, modifying the
underlying system model very often requires the design of
protocols suited to appropriate models [7].

The paper is organized as follows. Section 2 defines the
 crash-recovery model and the Atomic Broadcast problem
in such a model. Then, Section 3 presents the underlying
building blocks on top of which the proposed protocol is
built, namely, a transport protocol and a Consensus proto-
col suited to the crash-recovery model. A minimal version
of our Atomic Broadcast protocol for the crash-recovery
model is then presented in Section 4. The impact of ad-
ditional log operations on the protocol is discussed in Sec-
tion 5. Section 6 relates the Atomic Broadcast problem in
the crash-recovery with other relevant problems. Finally,
Section 7 concludes the paper.

2 Atomic Broadcast in the CR-Model

The Crash-Recovery Model
We consider a system consisting of a finite set of processes

\[ \Pi = \{p_1, \ldots, q\} \]

At a given time, a process is either up or down. When it is up, a
process progresses at its own speed behaving according to
its specification (i.e., it correctly executes its program text).
While being up, a process can fail by crashing: it then stops
working and becomes down. A down process can later re-
cover: it then becomes up again and restarts by invoking a
recovery procedure. So, the occurrence of the local event

\[ \text{crash} \quad \text{(resp. } \text{recover}) \]

generated by the local environment of a process, makes this process transit from up to down (resp. from down to up).

A process is equipped with two local memories: a
volatile memory and a stable storage. The primitives log
and retrieve allow an up process to access its stable stor-
age. When it crashes, a process definitely loses the content
of its volatile memory; the content of a stable storage is not
affected by crashes.

Processes communicate and synchronize by sending and
receiving messages through channels. We assume there is a
bidirectional channel between each pair of processes. Chan-
nels are not necessarily FIFO; moreover, they can duplicate
messages. Message transfer delays are finite but arbitrary.
Even if channels are reliable, the combination of crashes,
recoveries and arbitrary message transfer delays can entail
message losses: the set of messages that arrive at a process
while it is down are lost. Thus, the protocol must be pre-
pared to recover from messages losses.

Atomic Broadcast
Atomic Broadcast allows processes to reliably broadcast messages and to receive them in the same
delivery order. Basically, it is a reliable broadcast plus an
agreement on a single delivery order. We assume that all
messages are distinct. This can be easily ensured by adding
an identity to each message, an identity being composed of
a pair (local sequence number, sender identity).

At the syntactical level, Atomic Broadcast is composed of
two primitivies: 

\[ \text{A-broadcast}(m) \]

(used to send messages) and 

\[ \mu_p = \text{A-deliver-sequence()} \]

(used by the upper
layer to obtain the sequence of ordered messages). As in
[3], when a process executes 

\[ \text{A-broadcast}(m) \]

we say that it "A-broadcasts" 

\[ m \]

We also define a boolean predicate 

\[ \text{A-delivered}(m, \mu_p) \]

which evaluates to "true" if 

\[ m \in \mu_p \]

or "false" otherwise. We also say that some process 

\[ p \]

"A-delivers" 

\[ m \]

if 

\[ \text{A-delivered}(m, \text{A-deliver-sequence}()) \]

is "true" at 

\[ p \]

In the context of asynchronous distributed systems
where processes can crash and recover, the semantics1 of

\[ ^1 \text{We actually consider the definition of the Uniform Atomic Broadcast}

problem [8].\]
Atomic Broadcast is defined by the four following properties: Validity, Integrity, Termination and Total Order. This means that any protocol implementing these communication primitives in such a crash/recovery context has to satisfy these properties.

- **Validity**: If a process A-delivers a message m, then some process has A-broadcast m.
- **Integrity**: Let \( \mu_p \) be the delivery sequence at a given process p. A message m appears at most once in \( \mu_p \).
- **Termination**: For any message m, (1) if the process that issues A-broadcast(m) eventually remains permanently up, or (2) if a process A-delivers a message m, then all processes that eventually remain up A-deliver m.
- **Total Order**: Let \( \mu_p \) be the sequence of messages A-delivered to process p. For any pair \((p, q)\), either \( \mu_p \) is a prefix of \( \mu_q \) or \( \mu_q \) is a prefix of \( \mu_p \).

The validity property specifies which messages can be A-delivered by processes: it states that the set of A-delivered messages can not contain spurious messages. The integrity property states there are no duplicates. The termination property specifies the situations where a message m has to be A-delivered. The total order property specifies that there is a single total order in which messages are A-delivered. This is an Agreement property that, joined to the termination property, makes the problem non-trivial.

### 3 Underlying Building Blocks

The protocol proposed in Section 4 is based on two underlying building blocks: a Transport Protocol and a protocol solving the Uniform Consensus problem. This section describes their properties and interfaces.

**Transport Protocol** The transport protocol allows processes to exchange messages. A process sends a message by invoking a \texttt{send} or \texttt{multisend} primitive\(^2\). Both primitives are unreliable: the channel can lose messages but it is assumed to be fair, \textit{i.e.}, if a message is sent infinitely often by a process \( p \) then it is received infinitely often by its receiver [10]. When a message arrives at a process it is deposited in its input buffer that is a part of its volatile memory. The process will consume it by invoking a \texttt{receive} primitive. If the input buffer is empty, this primitive blocks its caller until a message arrives.

\(^2\)The primitive \texttt{multisend} is actually a macro that allows a process \( p \) to send (by using the basic \texttt{send} primitive) a message to all processes (including itself).

**Consensus Interface** In the Consensus problem each process proposes a value and all correct processes have to decide on some value \( v \) that is related to the set of proposed values [4]. The interface with the \texttt{Consensus} module is defined in terms of two primitives: \texttt{propose} and \texttt{decided}. As in previous works (e.g., [3]), when a process \( p \) invokes \texttt{propose}(\( w \)), where \( w \) is its proposal to the Consensus, we say that \( p \) “proposes” \( w \). A process proposes by logging its initial value on stable storage; this is the only logging required by our basic version of the protocol. In the same way, when \( p \) invokes \texttt{decided} and gets \( v \) as a result, we say that \( p \) “decides” \( v \) (denoted \texttt{decided}(\( v \))).

The definition of the Consensus problem requires a definition of a “correct process”. As the words “correct” and “faulty” are used with a precise meaning in the crash (no-recovery) model [3], for clarity purpose, we define their equivalents in the crash-recovery model, namely, “good” and “bad” processes (we use the terminology of [1]).

**Good and Bad Processes** A good process is a process that eventually remains permanently up. A bad process is a process that is not good. So, after some time, a good process never crashes. On the other hand, after some time, a bad process either permanently remains crashed or permanently oscillates between crashes (down periods) and recoveries (up periods). From a practical point of view, a good process is a process that, after some time, remains up long enough to complete the upper layer protocol. In the Atomic Broadcast problem for example, this means that a good process that invokes A-broadcast(m) will eventually terminate this invocation (it is possible that this termination occurs only after some (finite) number of crashes). It is important to note that, when considering a process, the words “up” and “down” refer to its current state (as seen by an external observer), while the words “good” and “bad” refer to its whole execution.

**Consensus Definition** The definition of the Consensus problem in the crash-recovery model is obtained from the one given in the crash (no-recovery) model by replacing “correct process” by “good process”. Each process \( p \) has an initial value \( v_i \) that it proposes to the others, and all good processes have to decide on a single value that has to be one of the proposed values. More precisely, the Consensus problem is defined by the following three properties (we actually consider the Uniform version [3] of the Consensus problem):

- **Termination**: Every good process eventually decides some value.
- **Uniform Validity**: If a process decides \( v \), then \( v \) was proposed by some process.
- **Uniform Agreement**: no two processes (good or bad) decide differently.

**Enriching the Model to Solve Consensus**  As noted previously, the Consensus problem has no deterministic solution in the simple crash (no-recovery) model. This model has to be enriched with a failure detector that, albeit unreliable, satisfies some minimal conditions in order that the Consensus be solvable. The crash-recovery model has also to be augmented with a failure detector so that the Consensus can be solved. With this aim, different types of failure detectors have been proposed. The protocol proposed in [9] use failure detectors that outputs list of “suspects”; so, their outputs are bounded. [1] uses failure detectors whose outputs are unbounded (in addition to lists of suspects, the outputs include counters). The advantage of the later is that they do not require the failure detector to predict the future behavior of bad processes. A positive feature of our protocol is that it does not require the explicit use of failure detectors (although these are required to solve the Consensus problem). Thus, it is not bound to any particular failure detector mechanism.

4  **The Basic Protocol**

4.1 **Basic Principles**

The proposed protocol borrows some of its principles from the total order protocol designed for the crash (no-recovery) model that is described in [3].

As illustrated in Figure 1, the protocol interfaces the upper layer through two variables: the Unordered set and the Agreed queue. Messages requested to be atomically broadcast are added to the Unordered set. Ordered messages are inserted in the Agreed queue, according to their relative order. The Agreed is a representation of the delivery sequence. Operations on the Unordered and Agreed variables must be idempotent, i.e., if the same message is added twice the result is the same as if it is added just once (since message have unique identifiers, duplicates can be detected and eliminated). We postpone the usage of A-checkpoint until Section 5.

The protocol requires the use of a Consensus protocol and of an unreliable (but fair) transport protocol offering the send, multisend, and receive primitives described in Section 3. The transport protocol is used to gossip information among processes. The interface with the Consensus protocol is provided by the propose and decided primitives. The propose primitive accepts two parameters: an integer, identifying a given instance of the Consensus, and a proposed value (a set of messages) for that instance. When a Consensus execution terminates, the decided primitive returns the messages decided by that instance of the Consensus in the variable result. The Consensus primitives must also be idempotent: upon recovery, a process may (re)invoke these primitives for a Consensus instance that has already started or even terminated.

The Atomic Broadcast protocol works in consecutive rounds. In each round, messages from the Unordered set are proposed to Consensus and the resulting decided messages moved to the Agreed queue. Before proceeding to the next round, the process logs its state in stable storage such that, if it crashes and later recovers, it can re-start ordering messages from the last agreed messages.

Processes periodically gossip their round number and their Unordered set of messages to other processes. This mechanism provides the basis for the dissemination of unordered messages among good processes. When a GOSSIP message is received from a process with a lower round number, this means that the sender of the gossip message may have been down and missed the last broadcasts.

4.2 **Protocol Description**

We now provide a more detailed description of the protocol. The protocol is illustrated in Figure 2. The state of each process \( p \) is composed of: \( k_p \): the round counter (initialized to 0); Proposed\(_p\): an array of sets of messages proposed to Consensus. Proposed\(_p\)[\( k_p \)] is the set of messages proposed to the \( k_p^{th} \) Consensus. All entries of the array are initialized to \( \bot \) (\( \bot \) means “this entry of the array has not yet been used”); Unordered\(_p\): a set of unordered messages, requested for broadcast (initialized to \( \emptyset \)); Agreed\(_p\): a queue of already ordered messages (initialized to \( \bot \)); gossip-k\(_p\): a variable that keeps the value of the highest Consensus round known as already decided (this value is obtained via the gossiping mechanism).

The first four variables can be structured as two pairs of variables. The \((k_p, Proposed\(_p\)) pair is related to the current (and previous) Consensus in which \( p \) is (was) involved. The \((Agreed\(_p\), Unordered\(_p\)) pair is related to the upper layer interface. Statements associated with message receptions are
executed atomically. The sequencer task and the gossip task constitute the core of the protocol. Both tasks access atomically the variables $k_p$ and Unordered$_p$.

A-broadcast$(m)$ issued by process $p$ consists in adding $m$ to its set Unordered$_p$. Then the protocol constructs the common delivery order. A-deliver issued by $p$ takes the next message from the Agreed$_p$ queue and A-delivers it to the upper layer application. The activation of the protocol is similar in the initial case and in the recovery case: the gossip and sequencer tasks are started (line a).

The gossip task This task is responsible for disseminating periodically a relevant part of the current state of processes. The gossip message sent by a process $p$, namely GOSSIP($k_p$, Unordered$_p$), contains its round number and its set of unordered messages. The goal of the gossip task is twofold. In first place, it ensures the dissemination of data messages, such that they are eventually proposed to Consensus by all good processes. In second place, it allows a process that has been down to know which is the most up-to-date round.

Upon reception of a GOSSIP message, an active process $p$ updates its Unordered$_p$ set and checks if the sender $q$ has a higher round number ($k_p > k_q$). In this case, $p$ records that it has lagged behind by updating the gossip-k$_p$ variable. This variable is used by the sequencer task to get the result of the Consensus p has missed.

The sequencer task This task is the heart of the ordering protocol [3]. The protocol proceeds in rounds. In the round $k$, a process $p$ proposes its Unordered$_p$ set to the $k^{th}$ instance of Consensus. Before starting the Consensus, the proposed value is saved in stable storage. Note that the propose primitive must be idempotent: in case of crash and recovery, it may be called for the same round more than once. The result of the Consensus is the set of messages to be assigned sequence number $k$. These messages are moved (according to a deterministic rule) from the Unordered$_p$ set to the Agreed$_p$ queue. Then, the round number $k_p$ is incremented and the messages that remain in the Unordered$_p$ set are proposed by process $p$ during the next Consensus. The sequencer task has to execute some statements atomically with respect to the processing of GOSSIP messages. This is indicated by bracketing with “[“ and “]” the corresponding statements in the sequencer task.

To avoid running unnecessary instances of Consensus, a process does not starts a new round unless it has some messages to propose or it knows it has lagged behind other processes. In the later case, it can propose an empty set as the initial value for those Consensus it has missed (this is because for those Consensus a decision has already been taken without taking $p$’s proposal into account).

Logging into stable storage Logging is used to create checkpoints from which a recovering process can continue its execution and consequently make the protocol live. So, at a critical point, the values of relevant variables are logged into stable storage. In this paper we are interested in discussing a protocol that makes a minimal number of checkpoints (independently of those required by the underlying Consensus protocols). Thus, we only log the initial value proposed for each Consensus round. This guarantees that if process $p$ crashes before the Consensus decides, $p$ will propose the same value again after recovering. We will later argue that this logging step cannot be avoided.

Note that we do not log the Unordered$_p$ set or the Agreed$_p$ queue. The Agreed$_p$ queue is re-constructed upon recovery from the results of past Consensus rounds by the replay procedure. To ensure that messages proposed to Atomic Broadcast are not lost, the A-broadcast$(m)$ primitive does not returns until the message $m$ is in the agree...
queue. If the process fails before that, there is no guaranty that the message has been logged, so the message may have or may have not been A-broadcasted. The latter case, it is the same as if the process has failed immediately before calling \texttt{A-broadcast}(m). Note that these design options that aim at minimizing the number of logging operations, do not necessarily provide the more efficient implementation. Alternative designs are discussed below.


declaration of queue. Since the protocol only logs the initial values proposed for each instance of Consensus, the current round \( k_p \) and the \emph{Agreed} queue have to be re-constructed upon recovery. The current round is simply the round for which no initial value has been proposed yet. The agreed queue can be reconstructed by reading the results of the Consensus instances that have terminated. Thus, before forking the \emph{sequencer} and \emph{gossip} tasks, the process parses the log of proposed and agreed values (kept internally by Consensus).

\subsection*{4.3 On the Minimal Logging}

Our solution only requires the logging of the initial proposed value for each round of Consensus. We argue that this logging operation is required for every atomic protocol that uses Consensus as a black box. In fact, all Consensus protocols for the crash-recovery model we are aware of assume that a process \( p \) proposes a value by writing it on stable storage. For instance, upon recovery the protocol of [1] checks the stable storage to see if a initial value has been proposed.

\section{An Alternative Protocol}

We now present a number of modifications to our basic protocol that, although increasing slightly the complexity and the number of log operations, may provide some benefits in practical systems. The protocol proposes a state transfer mechanism and additional log operations to reduce the recovery overhead and increase throughput. Additionally, the protocol shows how to prevent the number of entries in the logs from growing indefinitely, by taking application-level checkpoints. The version of the protocol that takes into account the previous considerations is illustrated in Figure 3.

\subsection*{Avoiding the Replay Phase}

In the previous protocol, we have avoided any logging operation that is not strictly required to ensure protocol correctness. In particular, we have avoided to log the current round \( (k_p) \) and agreed queue \( (\text{Agreed}^p) \), since they can be recomputed from the entries of the array \( \text{proposed}^p \) that have been logged. This forces the recovering process to replay the actions taken for each Consensus result \( (i.e., \text{insert the messages in the agreed queue according to the predetermined deterministic rule}) \).

\begin{figure}[h]
\centering
\begin{lstlisting}
Initial values:
\[ k_p = 0; \forall k : \text{Proposed}^k[0] = \perp; \]
\[ \text{Unordered}^p = \emptyset; \text{Agreed}^p = \{ \text{A-checkpoint}\; \}; \]
\[ \text{gossip} = 0; \]

procedure \emph{reply} ()

\begin{itemize}
  \item same as before without its first line
  \item \text{proposed}(\text{Proposed}^p);
  \item \text{wait until decided}(k_p, \text{result});
  \item \text{agreed}(k_p + 1; \text{Agreed}^p); \text{result}
\end{itemize}

upon \text{initialization or recovery}:

\begin{itemize}
  \item \text{same as before with the addition of the first line}
  \item \text{retrieve}(k_p, \text{Agreed}^p);
  \item \text{retrieve} \{ \text{Unordered}^p \};
  \item \text{log} \{ \text{Unordered}^p \};
\end{itemize}

upon \text{receive} \text{GOSSIP}(k_q, U_q) \text{from} q:

\begin{itemize}
  \item \text{first two lines: same as before}
  \item \text{if} (k_p > k_q) \text{then} \text{gossip}^p_q = \text{max}(\text{gossip}^p_q, k_q) \text{if} q \text{is ahead}
  \item \text{else} \text{if} (k_p > k_q + \delta) \text{then} \delta \text{is a configuration parameter}
  \item \text{send} \text{STATE}(k_p - 1, \text{Agree}^p_q) \text{to} q
\end{itemize}

\text{Task gossip}:

\begin{itemize}
  \item \text{same as before}
  \item \text{repeat forever} \text{multisend} \\text{GOSSIP}(k_p, \text{Unordered}^p)
\end{itemize}

\text{Task checkpoint}:

\begin{itemize}
  \item \text{new task}
  \item \text{repeat forever} \text{implementation dependent frequency}
  \item \text{log} (k_p, \text{Agreed}^p)
  \item \text{if} \text{Proposed}^p, t < k_p \text{can be discarded from the log}
\end{itemize}

\text{Task sequencer}:

\begin{itemize}
  \item \text{same as before}
  \item \text{repeat forever}
  \item \text{if} \text{Proposed}^p[0] = \perp \text{then}
    \text{Wait until} \text{Unordered}^p \text{or} (\text{gossip} > k_p);
    \text{Proposed}^p[0] = \text{Unordered}^p;
    \text{Ensure that despite crashes} \text{p always proposes}
    \text{the same input to the k}^p \text{Consensus}
    \text{log} \text{Proposed}^p[0];
  \end{itemize}

\begin{itemize}
  \item \text{else} \text{fork} \text{task} \{ \text{sequencer and gossip and checkpoint} \}
  \item \text{upon} \text{A-broadcast}(m):
    \item \text{issue} \text{upper layer}
    \item \text{same as before}
\end{itemize}

\begin{itemize}
  \item \text{fork} \text{task} \{ \text{sequencer} \}
\end{itemize}

\begin{itemize}
  \item \text{upon} \text{receive} \text{GOSSIP}(k_q, U_q) \text{from} q:
    \item \text{new message}
    \item \text{if} k_p < k_q - \delta \text{then} \text{p is late}
    \item \text{terminate task} \{ \text{sequencer} \}
      \text{Skip Consensus whose number k is such that}
      \text{if so, during the processing of the STATE message,}
      \text{the sequencer task is aborted}
      \text{agreed}(k_p + 1, \text{Agreed}^p_q)
    \item \text{else gossip}^p_q = \text{max}(\text{gossip}^p_q, k_q)
      \text{send} \text{STATE}(k_p - 1, \text{Agree}^p_q) \text{to} q
\end{itemize}

\end{lstlisting}
\caption{Alternative protocol}
\end{figure}
Faster recovery can be obtained at the expense of periodically checkpointing both variables. The frequency of this checkpointing has no impact on correctness and is an implementation choice (that must weight the cost of checkpointing against the cost of replaying). Note that old proposed values that do are not going to be replayed can be discarded from the log (line c).

**Size of logs and application-level checkpoint** A problem with the current algorithm is that the size of the logs grows indefinitely. A way to circumvent this behavior is to rely on an application-level checkpointing. In some applications, the state of the application will be determined by the (totally ordered) messages delivered. Thus, instead of logging all the messages, it might be more efficient to log the application state which logically “contains” the Agreed queue. For instance, when the Atomic Broadcast is used to update replicated data, the most recent version of the data can be logged instead of all the past updates. Thus, a checkpoint of the application state can substitute the associated prefix of the delivered message log.

In order to exploit this property, one needs to augment the interface with the application layer with an upcall to obtain the application state (see Figure 1). The upcall, state=A-checkpoint(µp), accepts as an input parameter a sequence of delivered messages and returns the application state that “contains” those updates. A-checkpoint(⊥) returns the initial state of the application. In order to know which messages are associated with a given checkpoint, a checkpoint vector clock VC(µp) is associated to each checkpoint. The vector clock stores the sequence number of the last message delivered from each process “contained” in the checkpoint. An application-level checkpoint is defined by the pair (A-checkpoint(µp), VC(µp)). The sequence of messages delivered to a process is redefined to include an application checkpoint plus the sequence of messages delivered after the checkpoint. A message m belongs to the delivery sequence if it appears explicitly in the sequence or if it is logically included in the application checkpoint that initiates the sequence (this information is preserved by the checkpoint vector clock).

In our protocol, the application state is periodically checkpointed and the delivered messages in the Agreed queue are replaced by the associated application-level checkpoint. This not only offers a shorter replay phase but also prevents the number of entries in the logs from growing indefinitely.

**State Transfer** In the basic protocol, a process that has been down becomes aware that it has missed some Consensus rounds when it detects that some other process is already in an higher round of Consensus (through the gossip messages). When this happens, it activates the Consensus instances that it has missed in order to obtain the correspondent agreed messages. A process that has been down for a long period may have missed many Consensus and may require a long time to “catch-up”.

An alternative design consists in having the most up-to-date process send a STATE message containing its current round number kp and its Agreedp queue. When a process p that is late receives a STATE message from a process q with a higher round number (kp < kq) it stops its sequencer task (line e), updates its state such that it catches up with that process, and re-starts its sequencer task from the updated state (line f), effectively skipping the Consensus instances it has missed.

Both approaches coexist in the final protocol. A late process can recover by activating the Consensus instances that it has missed or by receiving a STATE message. The amount of de-synchronization that triggers a state transfer can be tuned through the variable δ (line d).

Note that, for clarity, we have made the STATE message to carry the complete Agreed queue. Simple optimizations can minimize the amount of state to be transferred. For instance, since the associated GOSSIP messages carries the current round number of the late process, the STATE message can be made to carry only those messages that are not known by the recipient.

**Sending Message Batches** For better throughput, it may be interesting to let the application propose batches of messages to the Atomic Broadcast protocol, which are then proposed in batch to a single instance of Consensus. Unfortunately, the definition of Atomic Broadcast implies that every message that has been proposed by a good process be eventually delivered. When there are crashes, a way to ensure this property is not to return from A-broadcast(m) before m is logged. In the basic protocol we wait until the message is ordered (and internally logged by the Consensus). In order to return earlier, the A-broadcast interface needs to log the Unorderedp set.

**Incremental logging** As described, the protocol emphasizes the control locations where values have to be logged. The actual size of these values can be easily reduced. When logging a queue or a set (such as the Unordered set) only its new part (with respect to the previous logging) has to be logged. This means that a log operation can be saved each time the current value of a variable that has to be logged does not differ from its previously logged value.

6 Related Problems

**Consensus vs Atomic Broadcast** In this paper we have shown how to transform a Consensus protocol for the asynchronous crash-recovery model into an atomic broadcast
Atomic Broadcast and Transactional Systems  It has been shown that a deferred update replication model for fully replication databases can exhibit a better throughput if implemented with an Atomic Broadcast-based termination protocol than if implemented with Atomic Commitment [11]. The same report also proposes designs for Atomic Broadcast protocols in the crash-recovery model but these solutions are not Consensus-based.

Atomic Broadcast and Quorum-Based Systems  In a recent report [14] we show how to extend the Atomic Broadcast primitive to support the implementation of Quorum-based replica management in crash-recovery systems. The proposed technique makes a bridge between established results on Weighted Voting and recent results on the Consensus problem.

Total Order Multicast to Distinct Groups  In this paper we have focused on the Atomic Broadcast problem for a single group of processes. The problem of efficiently implementing atomic multicast across different groups in crash (no-recovery) asynchronous systems has been solved in several papers [5, 13]. Since these solutions are based on a Consensus primitive, it is possible to extend them to crash-recovery systems using an approach similar to the one that has been followed here.

7 Conclusion

This paper has proposed an Atomic Broadcast primitive for asynchronous crash-recovery distributed systems. Its concept has been based on a building block implementing Consensus. This building block is used as a black box, so our solution is not bound to any particular implementation of Consensus. The protocol is non-blocking in the following sense: as long as the underlying Consensus is live, the Atomic Broadcast protocol does not block good processes despite the behavior of bad processes. Moreover, our solution does not require the explicit use of failure detectors (even though those are required to solve the underlying Consensus). Thus, it is not bound to a particular failure detection mechanism. Also, we have shown that Atomic Broadcast can be solved without requiring any additional log operation in excess of those required by the Consensus. Finally, we have discussed how additional log operations can improve the protocol.

Acknowledgments  We are grateful to Rachid Guerraoui for his constructive comments on an earlier version of the paper.

References