Open Consensus

Romain Boichat[1],*Svend Frølund[2], Rachid Guerraoui[1]

[1]*Swiss Federal Institute of Technology, Lausanne*
[2]*Hewlett-Packard Laboratories, Palo Alto*

SUMMARY

This paper presents the abstraction of *open consensus* and argues for its use as an effective component for building reliable agreement protocols in practical asynchronous systems where processes and links can crash and recover. The specification of open consensus has a *decoupled*, *on-demand* and *re-entrant* flavour that make its use very efficient, especially in terms of forced logs, which are known to be major sources of overhead in distributed systems. We illustrate the use of open consensus as a basic building block to develop a modular, yet efficient, total order broadcast protocol. Finally, we describe our Java implementation of our open consensus abstraction and we convey our efficiency claims through some practical performance measures.

KEY WORDS: *Modularity, distribution, reliability, consensus, total order broadcast, open implementation*

1. INTRODUCTION

1.1. Context

It is widely accepted that modularity is a good idea, especially when writing reliable distributed protocols that are inherently complex. In practice however, very few reliable distributed programs are really modular, and very few abstractions are actually effective. One of the underlying reasons is that modularity is sometimes expensive: abstractions that are supposed to make a program modular turn out to be major sources of overhead. To be really effective, an abstraction must not only factor out some complexity, its overhead must also be *negligible*. Namely, the overhead introduced by the use of that abstraction in a given solution, with respect to an ad-hoc solution that bypasses that abstraction, should be negligible.

*Correspondence to: Romain Boichat, Distributed Programming Group, Communication Systems Department (DSC), Federal Institute of Technology Lausanne (EPFL), CH-1015 Lausanne, Switzerland, E-mail: Romain.Boichat@epfl.ch

The notion of a *consensus* service has recently been promoted as a central abstraction for building reliable distributed systems, and in particular for building their underlying distributed agreement protocols, e.g., total order broadcast, atomic commit, group membership and virtual synchrony [17, 16, 20]. Roughly speaking, a consensus service exports an operation *propose*(): processes invoke that operation with an initial parameter (each process might propose a different parameter), and all processes that do not crash gets as an output parameter the same returned decision [14, 7]. The idea of using consensus as a basic component to build agreement protocols is seductive because agreement problems are typically made of a "pure" agreement part, plus some "interpretation" part that is problem specific. The "pure" agreement part is similar in all the problems: it basically consists in agreeing on some value. Factoring out that part inside a consensus box can drastically simplify the description and implementation of the agreement protocols. In short, by considering consensus as a basic component in building various agreement protocols, one could benefit from the well-known advantages of modular programming in a difficult area, namely reliable distributed systems, where these advantages are sorely needed. Nevertheless, and as we pointed out, whether consensus can be an effective abstraction in building agreement protocols depends on the overhead introduced by the consensus abstraction with respect to ad-hoc protocols that bypass that abstraction.[†]

1.2. Motivation

Several implementations of consensus-based agreement protocols were given in [16, 20], and it was shown that the performance of those protocols are similar to the performance of ad-hoc agreement protocols. However, to convey this interesting result, a *crash-stop* system model was considered: processes are either up, or are down and never recover. In practice, processes may indeed crash, but some (or all) of them may recover. This *crash-recovery* model is a realistic system model for most of the applications we know of, but it introduces some fundamental difficulties in layering abstractions.

- If a process p_i crashes after *entering* some abstraction A, p_i might need to *re-enter* that abstraction upon recovery, which may not be possible unless entering the abstraction actually means storing some value on stable storage, e.g., the parameters of the abstraction invocation. To get a more concrete idea of this issue, consider the example of a total order broadcast protocol based on an underlying consensus abstraction [7, 23]. A consensus-based total order broadcast protocol typically uses a sequence of consensus instances, each instance being used to agree on a batch of messages [7]. If any process

[†]Obviously, the use of any abstraction always has an inherent overhead with respect to a solution that bypasses that abstraction: the inherent overhead is simply the cost of a local object invocation. However, in a distributed system, that overhead is usually considered negligible in comparison to *forced logs* and *communication delays*. Furthermore, in a distributed system, one may typically devise a protocol that is optimal for a given execution scenario (e.g., when no process crashes) and very inefficient in another scenario (e.g., if two processes crash). In practice, efficiency is a main concern in *nice* runs, where no process crashes, or is even suspected to have crashed. These are the runs that are the most frequent in practice and for which distributed protocols are usually optimised.

p_i crashes and recovers, p_i might not remember whether or not it proposed a value for consensus instance k, and which value it actually proposed. The specification of consensus requires every correct process to propose a value and precludes the possibility for any process to provide several different proposals for the same consensus instance (e.g., a process p_i cannot propose an initial value, crash, recover, and then propose a different value). As a consequence, proposing a value is typically defined as writing the initial value proposed on stable storage (e.g., [1]), and this must be performed by every correct process. Upon recovery, the forced log helps the process figure out what it might have proposed prior to the crash. The very same problem occurs with the decision, which is also typically defined as writing the final value on stable storage. Indeed, open consensus aims exactly at removing these forced logs by reshaping consensus.

- To ensure agreement, the processes must perform some forced logs so that they can remember which value they might have decided prior to a crash. Besides this usage, forced logs are also used to ensure integrity of the upper layer agreement protocol. If we consider for instance the total order broadcast example, integrity implies not delivering any message more than once. If consensus is used as a "closed" black-box to implement agreement, the two usages (agreement and integrity) must be clearly separated, which implies several forced logs. That is, the upper layer agreement protocol must perform specific forced logs to ensure integrity, and these must be different from those performed within the consensus box to ensure agreement. Open consensus indeed decouples the two aspects and only ensures the agreement part, thus allowing to reduce the number of redundant forced log with the upper layer.

In short, building an agreement protocol on top of a traditional consensus layer in a crash-recovery model has an inherent cost in terms of forced logs. Forced logs are usually considered very expensive because each one involves a synchronous write to the disk. One might be tempted to give up the use of a consensus box and develop ad-hoc protocols that minimises the number of forced logs. Another, more challenging, approach consists in figuring out a different way to factor out the consensus part of agreement protocols, i.e., a different way to shape consensus. This is exactly the approach promoted in this paper.

1.3. Contribution

This paper suggests a *reshaping* of consensus that makes it better suited for a practical use in reliable distributed programming.

1. We introduce the specification of a new consensus-like abstraction, which we call *open consensus*. Of course, proposing a new specification is fraught with the danger of defining a new abstraction that is either stronger than the original one, or on the contrary trivial (and hence useless). In both cases, we lose the benefits of reusing well-known results on the solvability of consensus. Fortunately, we define precise conditions under which open consensus and consensus are *equivalent* problems: under these conditions, any algorithm that implements one of the abstractions can be transformed to implement the other.

These conditions depend on the way open consensus is used, which is actually not surprising. Given that our open consensus abstraction exposes in its interface part of its implementation [22], its semantics indeed depend on its usage. Precisely because of this characteristic, in contrary to consensus, open consensus has some interesting flavours that make its use practical.

- *Re-entrant* flavour: a process can invoke the *propose*() operation of open consensus several times with different parameters, i.e., it can propose different values at different times. In particular, a process may propose a given value, crash, recover, and then propose a different value (e.g., if it has not logged the previous value): the same consensus decision will however be returned in both cases.
- *Decoupled* flavour: the pre-commitment of a decision is decoupled from its commitment: the actual coupling is under the control of the upper layer using the open consensus box (which can thus merge forced logs and reduce the number of required forced logs to solve consensus). Therefore, open consensus exports two primitives: *propose*() and *commit*() instead of only *propose*() for consensus.a process. This is precisely what makes it possible to merge forced logs of the upper layer with those of the open consensus box.
- *On-demand* flavour: processes do not all need to propose values and receive decisions. If a process is interested in receiving a consensus decision, it must invoke open consensus with a given parameter: otherwise the processes just act as *witnesses*.

2. We describe an open consensus algorithm where safety is ensured even if (all) processes crash (or keep crashing and recovering)) and messages are lost, whereas liveness (progress) is achieved if eventually, a majority of the processes remain up (for sufficiently long) and failure detection is eventually reliable. Interestingly, and despite its *re-entrant*, *decoupled* and *on-demand* flavours, our open consensus algorithm is rather simple. In particular, our notion of eventual failure detector reliability is captured by the simple failure detector specification of Ω, given for the crash-stop model in [6]. In comparison, new, and rather sophisticated, failure detector definitions were introduced in [1] to cope with process crash and recovery. Moreover, in nice runs (i.e., in failure-free and suspicion-free runs, which are the most frequent in practice), a process can reach a decision with open consensus after $\lceil\frac{n+1}{2}\rceil$ (concurrent) forced logs. Compared to consensus ($\lceil\frac{n+1}{2}\rceil+2$ forced logs, 3 are sequential), we do not increase the number of messages or the number of communication steps, but we drastically diminish the number of forced logs. Open consensus requires less forced logs than consensus since the forced logs are used to preserve only agreement and not to store propositions or decisions.

3. We illustrate the usefulness of our open consensus abstraction through an example of a reliable agreement protocol built upon this abstraction: a total order broadcast protocol. The resulting protocol is simple, modular, and efficient. It has the same communication pattern as a consensus-based total order broadcast protocol designed for a crash-stop model [7]. As in [7], a sequence of consensus (open consensus in our case) instances are used, each instance agrees on a batch of messages. However, we point out the fact

that our protocol introduces significantly less forced logs than an adaptation of the [7] consensus-based protocol to the crash-recovery model, i.e., a protocol than relies on a "traditional" consensus module in a crash-recovery model [25]. In fact, our algorithm is as efficient as the most efficient algorithm we know of to solve the same problem: that is, the algorithm of [23], which is non-modular and known to be rather complicated.[‡]

Underlying our open consensus abstraction, we argue for a modular approach to distributed programming. The distributed system is viewed as the problem domain from which fundamental abstractions should be extracted. Open consensus is indeed a candidate abstraction to build distributed agreement protocols. We describe in the paper the implementation of our agreement protocol framework in Java and we convey our efficiency claims using some performance measures. Although, for space limitation, we illustrate the use of open consensus through one agreement protocol, it is easy how to build other kinds of open consensus based, yet efficient, agreement protocols along the lines of [16].

1.4. Roadmap

The paper is organised as follows. We first describe our system model in Section 2. Section 3 introduces the specification of the open consensus abstraction and compares it with the traditional notion of consensus. We give in Section 4 an efficient algorithm that implements that specification and we discuss its analytical performance. We describe in Section 5 a total order broadcast algorithm built on top of open consensus, and we also discuss its analytical performance. Section 6 describes our Java implementation of open consensus and gives some practical performance measures. Section 7 summarises the paper and discusses some related work. Appendix 1 discusses the equivalence between open consensus and consensus.

2. SYSTEM MODEL

2.1. Processes

We consider a distributed system as a set of processes $\Pi = \{p_1, p_2, ..., p_n\}$. Each process represents a logical node in the system. At any 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). Note that we do not make here any assumption on the relative speed of processes. While being up, a process can fail by crashing; it then stops executing its program and becomes *down*. A process that is down can later recover; it then becomes up again and restarts by executing a recovery procedure. The occurrence of a *crash*

[‡][23] uses a consensus abstraction to explain the main idea of the total order broadcast algorithm, but the actual algorithm is efficient precisely because it bypasses that abstraction. In a sense, our paper suggests the best of both worlds: an efficient total order broadcast based on a consensus-like abstraction.

(resp. *recovery*) event makes a process transit from up to down (resp. from down to up). A process p_i is *unstable* if it crashes and recovers infinitely many times. We define an *always-up* process as a process that never crashes. We say that a process p_i is *correct* if there is a time after which the process is permanently up.[§] A process is *faulty* if it is not correct, i.e., either *eventually always-down* or *unstable*.

A process is equipped with two local memories: a volatile memory and a stable storage. The primitives **store** and **retrieve** allow a process that is up to access its stable storage. When it crashes, a process loses the content of its volatile memory; the content of its stable storage is however not affected by a crash and can be retrieved by the process upon recovery.

Finally, we assume the presence of a discrete global clock whose range ticks $\mathcal{T}$ is the set of natural numbers. This clock is used to simplify presentation and not to introduce time synchrony, since processes cannot access the global clock. We will indeed introduce some partial synchrony assumptions (otherwise, consensus and total order broadcast are impossible [14]), but as we will discuss, these assumptions will be encapsulated inside the specification of a failure detector and used only to ensure progress (liveness).

2.2. Link properties

Processes exchange information and synchronise by *sending* and *receiving* messages through channels. We assume the existence of a bidirectional channel between every pair of processes. We assume that every message m includes the following fields: the identity of its sender, denoted *sender(m)*, and a local identification number, denoted *id(m)*. These fields make every message unique. Channels can lose or drop messages and there is no upper bound on message transmission delays. We assume the same channel definition (given in [1]), which ensures the following properties between every pair of processes p_i and p_j:

No creation: If p_j receives a message m from p_i at time t, then p_i sent m to p_j before time t.

Finite duplication: If p_i sends a message m to p_j only a finite number of times, then p_j receives m only a finite number of times.

Fair loss: If p_i sends a message m to p_j an infinite number of times and p_j is correct, then p_j receives m from p_i an infinite number of times.

These properties characterise the links between processes and are independent of the process failure pattern occurring in the execution. The last two properties are sometimes called, respectively, *finite duplication* and *weak loss*, e.g., in [24]. They reflect the usefulness of the communication channel. Without these properties, any interesting distributed problem would be trivially impossible to solve. By introducing the notion of correct process into the *fair loss* property, we define the conditions under which a message is delivered to its recipient process.

[§] In practice, a process is required to stay up long enough for the computation to terminate. In asynchronous systems however, characterising the notion of "long enough" is impossible.

Indeed, the delivery of a message requires the recipient process to be running at the time the channel attempts to deliver it, and therefore depends on the failure pattern occurring in the execution. The *fair loss* property indicates that a message can be lost, either because the channel may not attempt to deliver the message or because the recipient process may be down when the channel attempts to deliver the message to it. In both cases, the channel is said to commit an *omission failure*.

2.3. Retransmission module

To simplify the presentation of our distributed algorithms in the next sections (open consensus and total order broadcast), we consider a *retransmission* channel, associated with two primitives: *s-send* and *s-receive*. These preserve the no creation and finite duplication properties of the underlying channels, and ensures the following validity property:

Validity: Let p_i be any process that s-sends a message m to a process p_j, and then p_i does not crash. If p_j is correct, then p_j eventually s-receives m.

We give in Figure 1 the algorithm of the retransmission module that relies on our more basic *send* and *receive* primitives. All messages that need to be retransmitted are put in the variable *xmitmsg* with their destination in the set *dst* (line 5). Messages in *xmitmsg* are erased once all recipients have acknowledged m, otherwise they are always retransmitted (lines 18-21).

Proposition 1. *Validity: Let p_i be any process that s-sends a message m to a process p_j, and then p_i does not crash. If p_j is correct, then p_j eventually s-receives m.*

Proof. Suppose that p_i s-sends a message m to a process p_j and then p_i does not crash. Assume by contradiction that p_j is correct, yet p_j does not s-receive m. There are two cases to consider: (a) p_j does not crash, or (b) p_j crashes, eventually recovers and remains always-up. For case (a), by the fair loss properties of the channels, p_j receives and then s-receives m: a contradiction. For case (b), since process p_i keeps on sending m to p_j, there is a time after which p_i remains up and sends m to p_j. As for case (a), by the fair loss property of the channels, p_j eventually receives m, then s-receives m: a contradiction. □

3. OPEN CONSENSUS: SPECIFICATION

We give here the semantics of our open consensus abstraction. We first recall the traditional specification of consensus in order to contrast it with open consensus. Second, we give the general idea of open consensus, and then a more precise specification of it.

```
for each process p_i:
procedure initialisation:
   xmitmsg[], dst[] ← ⊥; start task{retransmit}
procedure s-send(m)                                   {to s-send m to p_j}
   if m ∉ xmitmsg then xmitmsg ← xmitmsg ∪ m
   if p_j ∉ dst[m] then dst[m] ← dst[m] ∪ p_j
   for all p_j ∈ dst[m] do
      if p_j ≠ p_i then
         send m to p_j
      else
         simulate receive m from p_i
upon receive(m) from p_j do
   if m = ACK then
      dst[m] ← dst[m] \p_j
      if dst[m] = ⊥ then xmitmsg ← xmitmsg \m
   else
      s-receive(m); send ACK(m) to p_j
task retransmit                                       {retransmit all messages}
   while true do
      for all m ∈ xmitmsg do
         s-send(m)
```

Figure 1. Retransmission module

3.1. Traditional consensus: reminder

In the consensus problem [14], the processes are supposed to *propose* an initial value and eventually *decide* on the same final value, among one of the proposed values. Processes propose a value by invoking an operation *propose*() with their initial value as a parameter, and decide the value returned from that invocation. Of course, processes that crash are exempted from deciding. The problem was initially introduced in the crash-stop model [14] and a definition was given in [1] for the crash-recovery model. For consensus, as introduced before, a process is said to *propose* (resp. *decide*) a value when it writes that value into a specific stable storage location. The processes must satisfy the following properties.

Validity: If a process decides v, then v is the value proposed by some process.
Agreement: If no process proposes more than one value, then no two processes decide differently.
Termination: If every correct process proposes a value, then every correct process eventually decides some value.

Notice that the agreement and termination properties are not written here exactly as in traditional consensus specifications [14]. Indeed, it is usually implicitly assumed that no process

Prepared using cpeauth.cls

proposes more than one value. Similarly, it is usually implicitly assumed that every correct process proposes a value. We have explicited those assumptions here to clearly point out the difference between the agreement and termination properties of traditional consensus and open consensus.

3.2. Open consensus: overview

Like traditional consensus, open consensus enables the processes of a distributed system to *decide* on a common value *proposed* by one of the processes. However, unlike with traditional consensus, a process *using* open consensus can:

- Propose different values. A process can invoke the *propose*() operation of open consensus several times, with different parameters (*re-entrance* flavour). In particular, a process might propose a given value, crash, recover, and then propose a different value (e.g., if it has not logged the previous value).
- Control the actual commitment of a decision (*decoupled* flavour). That is, open consensus decouples the *pre-commitment* from the *commitment* of a decision and exposes that decoupling to the user of the consensus box. This is precisely what makes it possible to merge forced logs of the upper layer with those of the open consensus box.
- Not propose any value. In fact, the processes that do not propose any value participate in the open consensus implementation as "*witnesses*", but do not need to receive any decision. To receive a decision, they need to propose some value (*on-demand* flavour).

3.3. Open consensus: properties

To describe open consensus, we found convenient to represent it as a shared object that exports two operations: *propose*() and *commit*(). Operation *propose*() takes as a parameter a value in a set V (the set of consensus values) and returns a value in that very same set V. Operation *commit*() takes as a parameter a value in V and returns the value *ok*. We say that a process p_i *pre-commits* a value v if p_i gets v as an outcome of the invocation of *propose*(). We say that a process p_i *decides* a value v if p_i returns from the invocation of *commit*(v). Finally, we say that a process is a *proposee* if the process proposes some value. Open consensus has the following properties:

Validity: If a process pre-commits v, then v is the value proposed by some process.
Agreement: No two processes decide two different values.
Termination: If a process invokes *propose*() (resp. *commit*()) and then does not crash, it eventually returns from that invocation.¶

¶This property conveys a *wait-free* [18] characteristic of open consensus.

Not surprisingly, since our specification is somehow "open", the correctness of its implementations relies on the good behaviour of its user. Roughly speaking, we say that a process is *well-behaved* if p_i only invokes the operations in the order $propose(v); commit(v')$, where v' is the value returned from the *propose*() invocation. More precisely, we say that a process p_i is *well-behaved* if (1) whenever p_i returns from the invocation of $propose(v)$ with v' as an outcome parameter, p_i either crashes or invokes $commit(v')$, and (2) p_i only invokes $commit(v')$ if v' is the last value returned from p_i's invocation of $propose(v)$ since p_i's last crash and recovery.

We depict in Figure 2 four typical runs of open consensus. Figure 2(a) depicts a regular case where process p_1 proposes v_1, pre-commits v_1 and decides v_1. When process p_2 proposes v_2, p_2 pre-commits v_1, and then decides v_1. Figure 2(b) presents a case where a process crashes and recovers. Process p_1 proposes and pre-commits v_1, p_1 then crashes. When p_1 recovers, p_1 cannot invoke *commit*() since it is well-behaved; p_1 then proposes v_1', pre-commits and decides v_1'. In Figure 2(c), a process decides a value different from the one that it proposed. Process p_1 (resp. p_3) proposes and pre-commits v_1 (resp. v_3); but p_1 crashes and p_3 is slow and commits only later. When p_2 proposes v_2, p_2 pre-commits v_3 and then decides v_3 even though this value was not decided by p_3. Note that p_3 could not have pre-committed v_3 if p_1 did not crash. Figure 2(d) depicts a scenario where a process decides a proposition of a crashed process. Process p_1 proposes v_1 and crashes. Process p_2 proposes v_2 but pre-commits v_1. This is possible since some processes might have stored v_1 before p_1 crashed. Process p_2 then decides v_1, a value proposed by a crashed process.

We assume in the rest of the paper that processes are well-behaved. Under this assumption, we show in the appendix that open consensus is equivalent to consensus in terms of solvability. That is, possibility and impossibility results that were proved in the literature about consensus indeed apply to open consensus. However, and as we show in the next section, open consensus has a more efficient implementation than consensus.

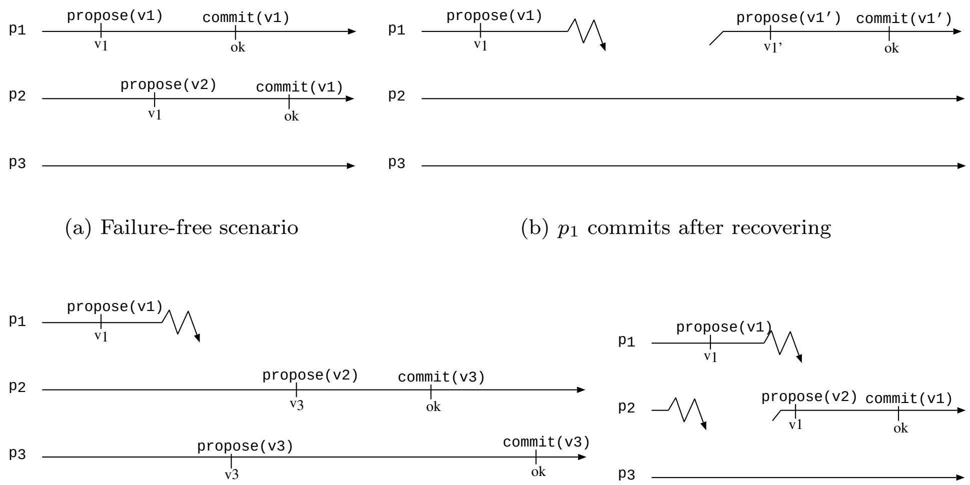
4. OPEN CONSENSUS: ALGORITHM

We describe here an open consensus algorithm and prove its correctness; we then discuss its analytical performance. More practical performance numbers are given in Section 6.

4.1. Description

4.1.1. Intuitive idea

The algorithm is based on a leader-follower scheme. Roughly speaking, leader processes try to concurrently reach a decision by storing it within a majority of the processes. The algorithm terminates when a single process is leader. When a process p_i invokes the *propose*() function with a value v, p_i sends it to the current leader. If p_i is actually the leader, p_i tries to gather the agreement on the value from half of the processes (other than itself). In the *commit*() function,



(a) Failure-free scenario

(b) p_1 commits after recovering

(c) p_1 crashes and p_3 decides p_2's proposition

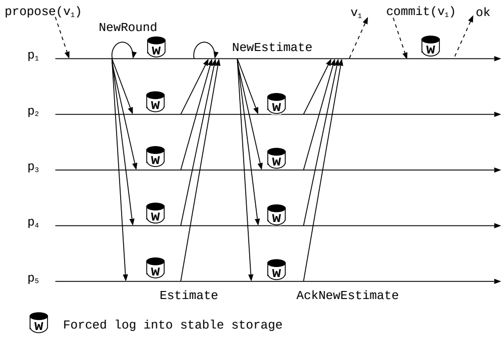
(d) p_2 decides p_1's proposition

Figure 2. Open consensus execution schemes

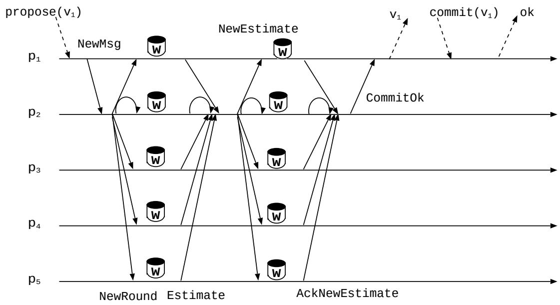
p_i decides v by logging it: a majority of the processes have then logged the decision. If p_i is not leader, the leader gathers the agreement directly from a majority of processes (instead of half if p_i is leader). Not surprisingly, the algorithm is optimised for runs where the proposee is leader.

More generally, the processes proceed in consecutive asynchronous rounds.‖ Each process has a local variable r defining the round it is currently involved in. Each round is made of two phases during which the processes exchange messages. Figure 3(a) depicts the messages and communication steps of open consensus if p_1 is leader and proposee, while Figure 3(b) presents the same steps but p_2 is leader. More precisely, when a process p_i proposes a value, p_i s-sends this value (into a NEWMSG message) to the leader if p_i is not leader. The leader then gathers estimates from a majority of processes to s-receive the latest estimate (NEWROUND and ESTIMATE messages). Second, if the leader is a proposee (resp. is not a proposee), then it waits for half (resp. majority) of the processes to agree on the estimate (NEWESTIMATE and ACKNEWESTIMATE messages). When a process s-receives either a NEWROUND (resp. NEWESTIMATE) message, it answers with an ESTIMATE (resp. ACKNEWESTIMATE) message with ack set to *true* or *false*. Ack is set to *true* if the following *acceptance* rule is satisfied: *The receiving process did not s-receive any* NEWROUND *or* NEWESTIMATE *message with a higher*

‖Although there are rounds, the protocol is not based on the rotating coordinator paradigm of [7, 1].



(a) p_1 is leader and proposee



(b) p_2 is leader

Figure 3. Open consensus: communication steps

round than the sending process. In any other case, ack is set to *false*. When p_i decides a value, p_i sends that value to all processes that have proposed (COMMITOK message).

4.1.2. Assumptions

Our algorithm relies on the assumptions that (1) all processes are well-behaved, (2) a majority of the processes are correct and (3) we have a failure detector with a specification similar to that of Ω in [6] (but adapted to a crash-recovery model): *There is a time after which some correct process is trusted by every process.* Failure detector Ω outputs a *trustlist*, i.e., a list of processes that are deemed to be currently up. We say that a process p_i is leader if p_i is the element of Ω.trustlist with the lowest process identity.**

4.1.3. Detailed description

Our algorithm is given in Figure 4. Each process p_i maintains a variable *decided* that contains the value that was decided. When p_i proposes, it sets the variable *proposed* to *true*, otherwise *proposed* is set to *false*. The variable *lastnewround* (resp. *lastnewest*) keeps track of the latest round at which p_i accepted a NEWROUND (resp. NEWESTIMATE) message. The actual round number is kept in the variable r, while the actual estimate is kept in the variable *est*.

There are four main parts in the protocol: (a) primitive *propose* s-sends the proposition to the leader if the process p_i is not leader, otherwise p_i launches task *coordinator*; (b) primitive

**One can implement Ω in a crash-recovery model with partial synchrony assumptions along the lines of [1].

commit decides the last pre-committed value (since the last recovery); (c) task *coordinator* gathers half of the processes to agree on a value (if p_i is not a proposee, the task gathers a majority of processes instead of half); and (d) primitives *receive* and *s-receive* handle all received messages, and stop task *coordinator* once p_i receives a decided value.

- In the primitive *propose*, invoked by a process p_i, p_i either s-sends the proposition in a NEWMSG message to the leader (if p_i is not the leader) or starts gathering estimates by invoking the coordinator task with *true* since p_i is a proposee (lines 10-13). Process p_i enters then a loop and waits for the value to be pre-committed. While waiting for the pre-commitment, upon a leader change, p_i s-sends the proposition (NEWMSG) to the new leader (lines 15-19). Once the value has been pre-committed, p_i returns from *propose*().
- In the primitive *commit*, when p_i decides the pre-committed value, p_i simply sets *decided* to the decided value (line 23) and sends a COMMITOK message to all processes that proposed (lines 24-26). It is possible that p_i has already decided when p_i invokes *commit*(); this case arises when p_i is not a proposee and is part of the majority set. In all cases, p_i returns *ok*.
- In task *coordinator*, the variable *local* is set to *true* if p_i is leader and proposee. When a process leader p_l s-receives a NEWMSG message, p_l starts (if it is not already doing it) to gather estimates by s-sending a NEWROUND message to all (line 30). When a process p_j s-receives such messages from p_l, p_j returns in an ESTIMATE message its actual estimate with ack set to *true* if p_j satisfies the acceptance rule. Otherwise, p_j s-sends an ESTIMATE message with ack set to *false*. If p_l s-receives a majority of ESTIMATE message with all ack set to *true*, then p_l selects the latest estimate (line 33) and s-sends it into a NEWESTIMATE message to all except p_l. When p_j s-receives such message, p_j s-sends an ACKNEWESTIMATE message with ack set to *true* if p_j satisfies the acceptance rule. Otherwise p_j s-sends an ACKNEWESTIMATE message with ack set to *false*. Finally, if p_l s-receives from half of the processes an ACKNEWESTIMATE message with all ack set to *true*, p_l returns the pre-committed estimate and buffers all the messages it receives or s-receives (lines 37-38). If p_l is not a proposee (local is set to *false*), p_l executes the same first steps but s-sends NEWESTIMATE to all (instead of all except p_l), waits for a majority of ACKNEWESTIMATE messages, sends a COMMITOK message to all processes that proposed and returns the pre-committed estimate which is in fact already decided (lines 40-45). Note that for this case, the leader does not buffer any message but empty its retransmission module.
- In the primitives *receive* and *s-receive*, when p_i receives (resp. s-receives) a message from p_j, p_i first verifies if it has already decided a value. In this case, p_i sends *decided* to p_j. When p_i s-receives a NEWMSG and p_i is leader, p_i starts task *coordinator* (if it is not already running) with *false* since p_i is not a proposee. When p_i s-receives a NEWROUND (resp. NEWESTIMATE) message, p_i s-sends an ESTIMATE (resp. ACKNEWESTIMATE) message with ack set to *true* or *false* following the acceptance rule. When p_i receives the decision value of consensus, p_i first stops task *coordinator* if it is active, sets *decided* and *pre-committed* to the decided value and empty its retransmission module.

```
for each process p_i:
procedure initialisation:
  pre-committed ← ⊥; decided ← ⊥; proposed ← false
  (r_{p_i}, lastnewround_{p_i}, est_{p_i}, lastnewest_{p_i}) ← (p_i,0,⊥,0)
upon propose(v_{p_i}) do
  proposed ← true
  wait until task coordinator is not active        {avoid starting the task more than once}
  if decided = ⊥ then                              {otherwise has decided meanwhile}
    if est_{p_i} = ⊥ then est_{p_i} ← v_{p_i}
    if p_i ∈ Ω.trustlist then
      pre-committed ← start task coordinator(true)
    else
      s-send (NEWMSG,v_{p_i}) to first(Ω.trustlist)
    while pre-committed = ⊥ do
      upon change in Ω do
        if p_i ∈ Ω.trustlist then
          pre-committed ← start task coordinator(true)
        else
          s-send (NEWMSG,v_{p_i}) to first(Ω.trustlist)
  return(pre-committed)
upon commit(v_{p_i}) do
  if decided = ⊥ then                              {if p_i ∉ Ω.trustlist, then a majority has stored v}
    lastnewest_{p_i} ← r_{p_i}; est_{p_i} ← v_{p_i}; decided ← v_{p_i};store{lastneweste_{p_i},est_{p_i}, decided}
    for all p_k such that s-received(ACKNEWESTIMATE,r_{p_i},proposed,ack) do
      if proposed is true then send(COMMITOK,est_{p_i}) to p_k
    empty retransmission buffer; treat all buffered messages
  return(ok)
task coordinator(local)
  while p_i ∈ Ω.trustlist do
    s-send(NEWROUND,r_{p_i}) to all
    wait until [s-received(ESTIMATE,r_{p_i}, est_{p_j}, lastnewest_{p_j},ack) from ⌈(n+1)/2⌉ processes]
    if received only ESTIMATE with ack = true then
      temp_{p_i} ← est_{p_j} | lastnewest_{p_j} | p_i s-received (ESTIMATE,r_{p_i}, est_{p_j}, lastnewest_{p_j},ack)
      if local then
        s-send(NEWESTIMATE,r_{p_i}, temp_{p_i}) to all \p_i
        wait until [s-received(ACKNEWESTIMATE,r_{p_i},proposed,ack) from ⌊n/2⌋ processes]
        if received only ACKNEWESTIMATE with ack = true then
          buffer all messages that p_i s-receive; return(temp_{p_i})
      else
        s-send(NEWESTIMATE,r_{p_i}, temp_{p_i}) to all
        wait until [s-received(ACKNEWESTIMATE,r_{p_i},proposed,ack) from ⌈(n+1)/2⌉ processes]
        if received only ACKNEWESTIMATE with ack = true then
          for all p_k such that s-received(ACKNEWESTIMATE,r_{p_i},proposed,ack) do
            if proposed is true then send(COMMITOK,est_{p_i}) to p_k
          empty retransmission buffer; pre-committed ← est_{p_i}; decided ← est_{p_i}
    r_{p_i} ← r_{p_i} + n
upon s-receive m or receive m from p_j do
  if decided ≠ ⊥ then
    send (COMMITOK,decided) to p_j
  else if m = (NEWMSG,v_{p_j}) then
    if p_i ∈ Ω.trustlist and task coordinator is not active then
      if est_{p_i} = ⊥ then est_{p_i} ← v_{p_j}; start task coordinator(false)
```

```
53: else if m = (NEWROUND,r_{p_j}) then                                    {Continued from s-receive}
54:   if lastnewround_{p_i} > r_{p_j} or lastnewest_{p_i} > r_{p_j} then
55:     s-send (ESTIMATE,r_{p_i}, est_{p_i},false) to p_j
56:   else
57:     lastnewround_{p_i} ← r_{p_j}; store{lastnewround_{p_i}}
58:     s-send(ESTIMATE,r_{p_i}, est_{p_i}, lastnewest_{p_i},true) to p_j
59: else if m = (NEWESTIMATE,r_{p_i}, temp_{p_j}) then
60:   if lastnewround_{p_i} > r_{p_j} or lastnewest_{p_i} > r_{p_j} then
61:     s-send(ACKNEWESTIMATE,r_{p_i},proposed,false) to p_j
62:   else
63:     lastnewest_{p_i} ← r_{p_j}; est_{p_i} ← temp_{p_j}; store{lastnewest_{p_i},est_{p_i}}
64:     s-send(ACKNEWESTIMATE,r_{p_i},proposed,true) to p_j
65: else if m = (COMMITOK,est_{p_j}) then
66:   if task coordinator is active then stop task coordinator
67:   decided ← est_{p_j}; pre-committed ← est_{p_j}; empty retransmission buffer
68: upon recovery do
69:   initialisation; retrieve{lastnewround_{p_i}, est_{p_i}, lastnewest_{p_i},decided}
```

Figure 4. Open consensus

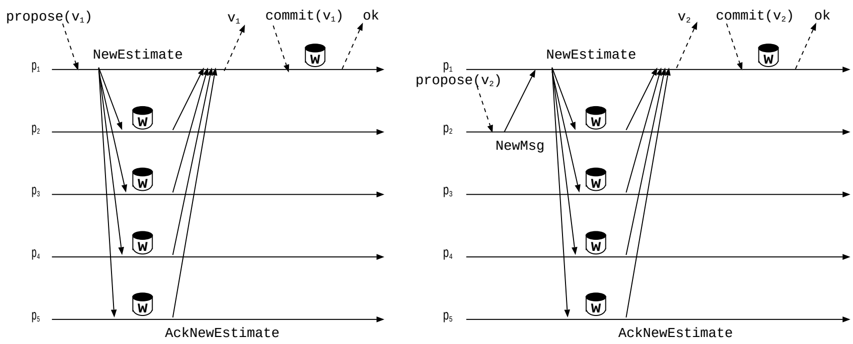
4.1.4. Remarks

Note also that in round 0, the leader p_1 can simply set its estimate to its *own* proposed value and skip the phase used to select the estimate (NEWROUND-ESTIMATE). It is also easy to see that the coordinator does not have to store its round number into stable storage in this case. We omitted these obvious optimisations from the code. Figure 5 depicts the communication steps for such scenario: in Figure 5(a), the proposee is leader, and in Figure 5(b), the proposee is p_2 and the leader is p_1. Therefore, in a nice run where p_1 is leader, the algorithm requires only $\lceil \frac{n+1}{2} \rceil$ forced logs and one round-trip communication step for p_1 to decide (the same number of forced logs but three communication steps if the proposee is not leader).[††]

4.2. Correctness

Lemma 2. *Validity: If a process pre-commits v, then v is the value proposed by some process.*
Proof (sketch). The decided value is chosen at line 33 (est_{p_j}) and est_{p_j} is modified in lines 8 and 23 (the other modifications are meaningless since they are induced by the first two). Line 23 does not impact the pre-committed value since it is executed in the *commit*() function. Therefore, line 8 is the only modification that affects the pre-committed value. Line 8 sets est_{p_i} to the value proposed; indeed, by the algorithm of Figure 4 and by the properties of the links, it is impossible for a process to pre-commit a value that was not proposed (out of thin air). □

[††]Note that if all processes propose, our algorithm is also quiescent [2].



(a) p_1 is leader and proposee (b) p_1 is leader and p_2 proposee

Figure 5. Open consensus communication step for round 0

Lemma 3. *If a process p_i is leader, pre-commits v and then p_i does not crash, then a majority of processes have stored v in stable storage.*
Proof (sketch). By the algorithm of Figure 4, when p_i pre-commits w_i for round r, $\lfloor \frac{n}{2} \rfloor$ other process than p_i have stored w_i and $lastnewround = r$. Since every process is well-behaved, then p_i invokes $commit(w_i)$ and stores w_i, therefore there is a majority of processes that have stored w_i. However, there can be more than one process that invokes $propose()$ and $commit()$. By line 38, once p_i returns from $propose()$, p_i will not modify any variable since p_i buffers all the messages that it receives or s-receives. Therefore, if p_i does not crash (i.e., decides) and another process p_j invokes $propose(v_j)$, then p_j pre-commits w_i since there cannot be two different majorities in the system (line 32-33). By line 32 and the fact that p_i does not answer to any message, p_j must receive an ESTIMATE message with w_i. By lines 59-64, this message must be tagged with the higher *lastnewest* otherwise p_i could not have decided w_i. □

Lemma 4. *If a process decides v, then a majority of processes have stored v in stable storage.*
Proof (sketch). Remember that we assume that every process is well-behaved, therefore a process invokes $propose()$ and then $commit()$ with the last value pre-committed by itself since its last recovery. There are two cases to consider: (i) the proposee is not leader, or (ii) the proposee is a leader. For case (i), by the algorithm of Figure 4, when a process p_i returns from $propose(v_i)$, the value returned w_i is already stored at a majority of processes, i.e., w_i can be in fact already decided for p_i if p_i is part of the majority set that acknowledged w_i. Therefore, when p_i has already decided and invokes $commit(w_i)$, p_i does nothing (line 27). Lemma 3 and the notion of well-behaved solve case (ii). □

Lemma 5. *Agreement: No two processes decide two different values.*
Proof (sketch). Suppose that a process p_i (resp. p_j) decides v (resp. v'). Assume by contradiction that $v \neq v'$ and without loss of generality that p_i decides before p_j. By lemma 4 and by the algorithm of Figure 4, when p_i decides in round r, then a majority of processes have stored v and a value of *lastnewest* $\geq r$. All *lastnewest* could not be equal to r because some process could have invoked *propose*() with a higher round and the *lastnewest* value would be changed (but not the *est*). By lemma 4, there is also a majority of processes that have stored v'. There must be then a process that has stored both v and v'. This is impossible since once p_i has decided v, when p_j proposes, p_j must have received an ESTIMATE message with v. This message is tagged with the highest *lastnewest*, otherwise p_i would have decided a value different from v. □

Lemma 6. *If a process invokes propose() (resp. commit()) and then does not crash, it eventually returns from that invocation.*
Proof (sketch). The proof is trivial for *commit*(). For *propose*(), by (i) the fact that there is a majority of correct processes in the system, and (ii) by the property of Ω, there is time after which there is only one eventual perpetual leader p_l in the system. If a correct process proposes, then p_l eventually s-receives a NEWMSG message and can then pre-commit. □

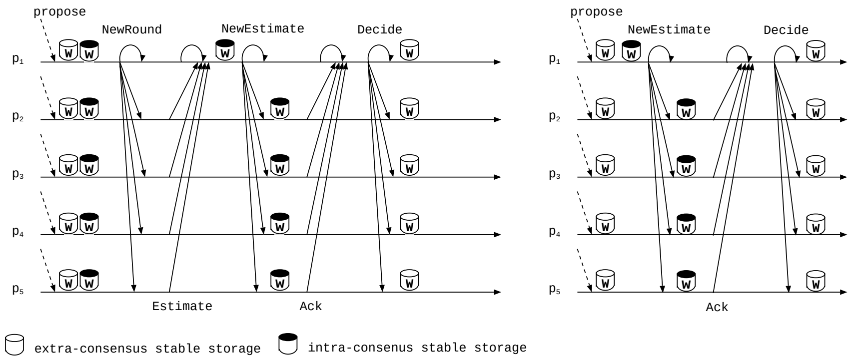
Proposition 7. *The algorithm of Figure 4 satisfies the validity, agreement and termination properties of open consensus.*
Proof (sketch). Follows directly from lemmata 2, 5 and 6. □

4.3. Analytical evaluation

In [1], the authors described a consensus protocol for a crash-recovery model, and indeed assumed that every invocation and every decision of consensus coincides with a forced log. Hence, besides the required forced log to preserve agreement, additional forced logs are needed for the interaction with the consensus box: these introduce a *pure* overhead to the consensus abstraction.[‡‡]

As depicted in Figures 6(b) and 5 (resp. Figures 6(a) and 3), in a nice run, the number of communication steps needed to reach a decision is the same for both algorithms. However, a process can reach a decision after one local forced log in our algorithm (one forced log for the agreement), whereas three local sequential forced logs are required in [1] (one forced log for the proposition, one for the agreement and one for the decision). Globally, for a process to decide in [1], $\lceil \frac{n+1}{2} \rceil$+2 forced logs must have been performed. In our case, a process can decide after $\lceil \frac{n+1}{2} \rceil$ forced logs. Moreover, our open consensus algorithm introduces fewer messages than [1] since not every process is required to propose, and only those that propose receive a decision message. In the case where all processes propose a value, then the number of messages is the same in both algorithms.

‡‡The same conclusion can be drawn for the consensus algorithm of [19].



(a) p_2 is leader (b) p_1 is leader (nice run)

Figure 6. Consensus

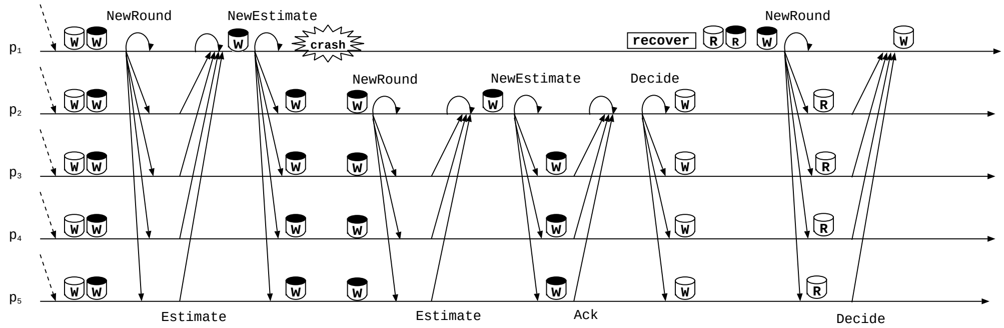


Figure 7. Consensus with a crashed coordinator

We now compare open consensus with consensus in case of a recovery scenario. Even if *open consensus* is optimised for nice runs, it behaves quite well in the case of a process crash. As shown in Figure 7, 8 and 9, open consensus is more efficient than consensus, both in terms of the number of communication steps and forced logs. As depicted in Figure 7, with consensus, if the coordinator crashes, another process takes up, becomes coordinator and solves consensus. When the process that has crashed recovers, it re-proposes by reading its location into stable storage and decides. We need to compare with two scenari for our algorithm: (i) if a proposee crashes as depicted in Figure 8, and (ii) if a leader process crashes as shown in Figure 9. For case (i), p_1, which is proposee and leader, crashes. Since no other process has proposed, no process tries to solve consensus. When p_1 recovers, p_1 retries to solve consensus, pre-commits and then decides the value. Note that p_1 proposed another value that it proposed in its first

Prepared using cpeauth.cls

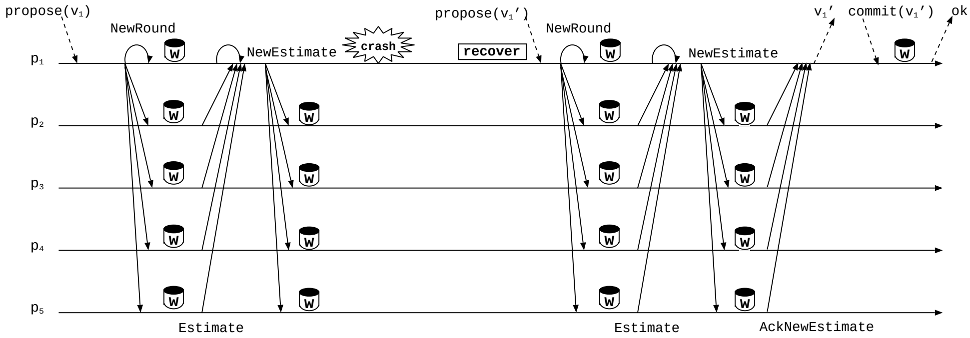


Figure 8. Open consensus with a crashed proposee

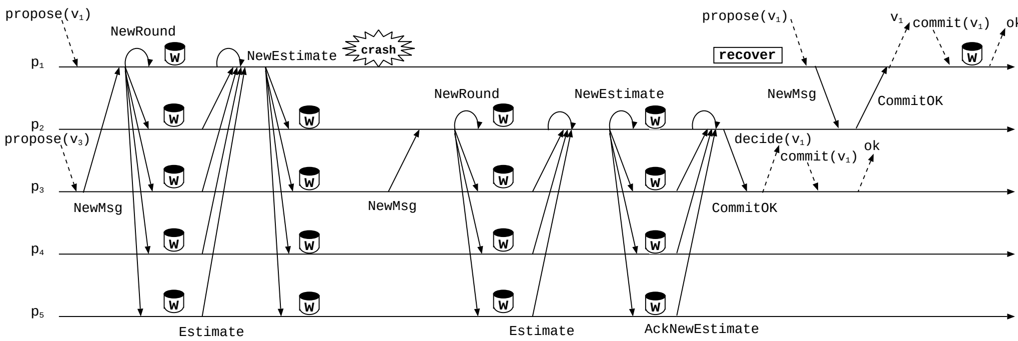


Figure 9. Open consensus with a crashed leader

trial. For case (ii), the coordinator crashes. Therefore, p_3 which is proposee suspects p_1 and then sends its proposition to the new leader. The new leader returns its pre-committed value to p_3, which then decides. When p_1 recovers, p_1 reinvokes *propose*(), pre-commits and then decides.

5. PUTTING OPEN CONSENSUS TO WORK: TOTAL ORDER BROADCAST

This section illustrates the effective use of open consensus to build modular yet efficient agreement algorithms. We describe a total order broadcast algorithm using open consensus and then prove its correctness. We compare then the performance of our algorithm with an algorithm based on a traditional consensus abstraction.

Prepared using cpeauth.cls

5.1. Specification

Total order broadcast is a communication abstraction that allows processes to broadcast and deliver messages in such a way that they agree on both the set of messages they deliver and the order in which these messages are delivered. We specify the underlying abstraction, in a crash-recovery model, with two primitives *TO-Broadcast* and *TO-Deliver*. These primitives satisfy the following properties (As in [21], we assume here that each message codes the process which initiated that message, denoted by $sender(m)$.):

Validity: For any message m, every process TO-Delivers m at most once and only if m was previously TO-Broadcast by $sender(m)$.

Agreement: If any process TO-Delivers a message m, then all correct processes eventually TO-Deliver m.

Termination: If a process TO-Broadcasts a message m and then does not crash, it eventually TO-Delivers m.

Total Order: Let p_i and p_j be any two processes that TO-Deliver some message m. If p_i TO-Delivers some message m' before m, then p_j also TO-Delivers m' before m.

It was shown in [7] that total order broadcast and consensus are equivalent problems in the crash-stop model. In particular, an algorithm was given to transform consensus into total order broadcast. [25] shows that this algorithm can be adapted to the crash-recovery model. Nevertheless, the use of traditional consensus as a building block introduces superfluous forced logs (as we shall discuss below). We present here an open consensus based, yet efficient, total order broadcast for the crash-recovery model. Thanks to the *on-demand*, *decoupled* and *re-entrant* flavours of open consensus, our transformation does not add any forced log to open consensus (beside what is needed inside open consensus).

5.2. Algorithm

Our algorithm is given in Figure 10. The algorithm uses a series of consecutive open consensus (or simply consensus) instances: each consensus instance being used to agree on a batch of messages. Each process differentiates consecutive instances by maintaining a local counter (k): each value of the counter corresponds to a specific consensus instance. We describe first the main data structure of the algorithm. A local set *Received* keeps all messages that needs to be decided, and another set *TO_Delivered* keeps track of all TO-Delivered messages. Intuitively, the algorithm works as follows. When there are still messages to be TO-Delivered, i.e., *Received*-*TO_Delivered* is not empty, process p_i launches a consensus instance and waits for the pre-commitment of the value. Note that we assume here that new messages keep on being broadcast, and that accesses and modifications of the variables are atomic.

An important aspect of our algorithm is the handling of the decoupling between the pre-commitment and the commitment of an open consensus decision. Once a value has been pre-committed, if a process p_i is a proposee and a leader, p_i knows that half of the processes (other than itself) have agreed on this value. Therefore, p_i can perform some execution steps before deciding the value. Indeed, p_i orders the messages following a deterministic order and

```
Every process p_i executes the following:
procedure initialisation:
  Received[] ← ⊥; AwaitingToBeDelivered[] ← ⊥; k ← 0; TO_Delivered[] ← ⊥
upon TO-Broadcast(m) do
  Received ← Received ∪ m
TO-Deliver(k) occurs as follows:
  while Received - TO_Delivered ≠ ⊥ do
    k ← k + 1; propose(k, Received-TO_Delivered)
    wait until[receive(pre-commit(k, msgSet^k))]
    msgSet^k ← msgSet^k in some deterministic order; commit(k, msgSet^k)          {TO-Deliver}
    TO_Delivered ← TO_Delivered ∪ msgSet^k; send(k, msgSet^k) to all \p_i
upon receive or s-receive(batch,msgSet) from p_j do
  if batch < k then
    for all k ≥ l > batch do
      send(l, msgSet^l) to p_j
  else if batch = k + 1 then
    k ← k + 1; commit(k, msgSet^k)                                                 {TO-Deliver}
    TO_Delivered ← TO_Delivered ∪ msgSet^k; empty retransmission buffer for batch k
    while AwaitingToBeDelivered[k + 1] ≠ ⊥ do
      k ← k + 1; commit(k, AwaitingToBeDelivered[k])                               {TO-Deliver}
      TO_Delivered ← TO_Delivered ∪ msgSet^k; empty retransmission buffer for batch k
  else
    AwaitingToBeDelivered[batch] ← msgSet; s-send(k,msgSet^k) to p_j
upon recovery do
  initialisation
  for all decided msgSet^k do
    retrieve(msgSet^k,k); TO_Delivered ← TO_Delivered ∪ msgSet^k
  Received ← TO_Delivered
```

Figure 10. Total order broadcast with open consensus

then decides this new set of messages. The same deterministic ordering function is used among all processes. Note that in the meantime (between returning from *propose*() and invoking *commit*()), the process does not answer to any messages.

When p_i invokes *commit*(), in fact, p_i sets the *decided* variable to the new ordered set. Once p_i has decided the set, p_i updates *TO_Delivered* and then sends the decision to every process. When a process p_j receives the decision, there are three cases to consider: (i) p_j is lagging, e.g., $k_{p_j} < k_{p_i}$, (ii) p_j is ahead, e.g., $k_{p_j} > k_{p_i}$, and (iii), p_j is in synch with p_i, e.g., $k_{p_j} = k_{p_i}$. For case (i), p_j puts the received decision in a buffer where it keeps all future decisions (*AwaitingToBeDelivered*) and s-sends its current state in order to receive all missing decisions between k_{p_j} and k_{p_i}. For case (ii), p_j simply sends all missing decisions to p_i, e.g., all decisions between k_{p_j} and k_{p_i}. Finally, for the last case, p_j TO-Delivers the decided set, removes the messages from the retransmission module (if there are any) for batch k and tries to TO-Deliver the following batches (k_{p_j} + 1,...). When p_i crashes and recovers, p_i retrieves all the decided values and appends them to reconstruct the set *TO_Delivered*, in order not to violate the integrity property of total order broadcast. Once a process recovers, p_i sets *Received* to *TO_Delivered* otherwise line 7 would never be false, thus keeping on proposing useless batches. Note that our algorithm is quiescent [2] if there are no unstable processes in the system. Indeed, when a batch k has been TO-Delivered by every correct process, no more messages for this batch are sent. It is quiescent since once a batch has been TO-Delivered by p_i, p_i stops its retransmission module for this batch

5.3. Correctness

Lemma 8. *Validity: For any message m, every process TO-Delivers m at most once and only if m was previously TO-Broadcast by* sender*(m).*

Proof (sketch). Consider the first part. A process can only TO-Deliver at most once a message m since *TO_delivered* and k are kept up to date. When a process recovers, it rebuilds the *TO_Delivered* set, therefore a process cannot TO-Deliver m more than once. Consider now the second part. For a message m to be TO-Delivered, m has first to be proposed. To be proposed, m has to belong to the *Received* set, and to be in this set, m has to be TO-Broadcast (no message come out of thin air). □

Lemma 9. *Agreement: If any process TO-Delivers a message m, then all correct processes eventually TO-Deliver m.*

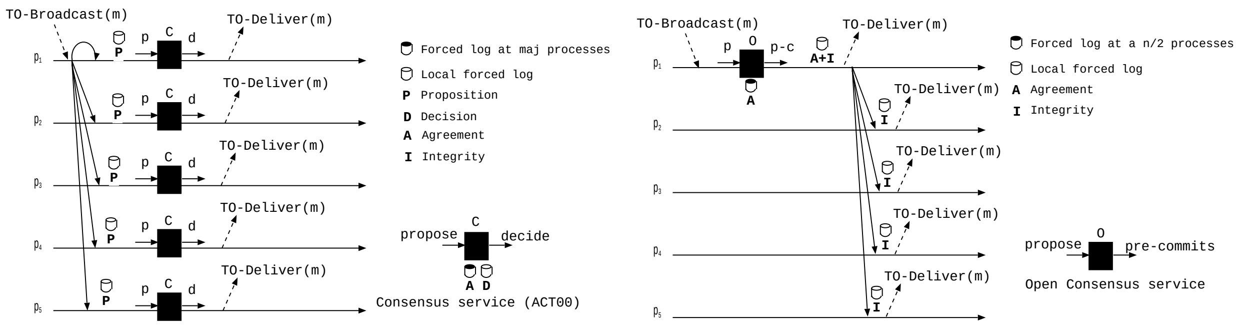
Proof (sketch). Remember that we suppose that new messages keep being broadcast, such that *Received* is never empty. Therefore, a correct process p_i has always messages to propose. Indeed, p_i keeps on sending decisions to every other process. There is a time after which all correct processes stop crashing and remain up. By the fair loss properties of the links, these correct processes eventually receive a decision. If they are lagging compared to p_i, by lines 15 and 23, every correct process receives all missing decision and TO-Delivers m. □

Lemma 10. *Termination: If a process TO-Broadcasts a message m and then does not crash, it eventually TO-Delivers m.*

Proof (sketch). If a process p_i TO-Broadcasts m and then does not crash, *Received* contains m. Since *Received* - *TO_delivered* is not empty, p_i proposes m in line 8. By the termination property of open consensus, p_i returns and pre-commits *msgSet*. There are two cases to consider: (a) $m \in$ *msgSet*, and (b) $m \notin$ *msgSet*. Case (a) is trivial since p_i then decides *msgSet* and TO-Delivers m. For case (b), m stays in *Received-TO_delivered* but p_i keeps on proposing m. Since p_i does not crash, p_i never loses the content of *Received* and eventually pre-commits a *msgSet* which contains m, thus TO-Delivering m. □

Lemma 11. *Total Order: Let p_i and p_j be any two processes that TO-Deliver some message m. If p_i TO-Delivers some message m' before m, then p_j also TO-Delivers m' before m.*

Proof (sketch). Trivial from lemma 10. Since every process TO-Delivers the same batch of messages. By the algorithm of Figure 10, the total order property of total order broadcast is satisfied. □



(a) Total order broadcast with consensus [25]

(b) Total order broadcast with open consensus

Figure 11. Comparison in a nice run

Proposition 12. *The algorithm of Figure 10 satisfies the validity, agreement, termination and total order properties of total order broadcast.*

Proof (sketch). Follows directly from lemmata 8, 9, 10, and 11. □

5.4. Analytical Evaluation

We compare our algorithm with the solution given by [25]: to our knowledge, that is the only consensus-based total order broadcast that was devised in a crash-recovery model. As we pointed out in the introduction, the algorithm of [23] indeed implements a total order broadcast primitive in a crash-recovery model, but bypasses the consensus abstraction.

The algorithm of [25] is efficient in terms of messages and communication steps, but to cope with recovery, a process can only TO-Deliver a message after $3\lceil\frac{n+1}{2}\rceil$ forced logs. Moreover the algorithm does not store the TO-Delivered messages but leaves that up to the upper layer. As pointed out by the authors of [25], the inefficiency of the scheme is inherent to the use of consensus as a black-box. In our algorithm, the process that is leader and proposee can TO-Deliver a message after $\lceil\frac{n+1}{2}\rceil$ forced logs. If the proposee is not leader, a message is then TO-Delivered after $\lceil\frac{n+1}{2}\rceil+1$ forced logs. Figure 11 compares, in a nice run, our total order broadcast algorithm with the algorithm of [25], i.e., the figure actually compares the impact of using open consensus with that of using traditional consensus in a crash-recovery model ([25]). Note that our algorithm is also simpler since it does not require every process to invoke consensus, and is quiescent. The algorithm of [25] uses an inherently non-quiescent gossip function (to achieve reliable broadcast semantics).

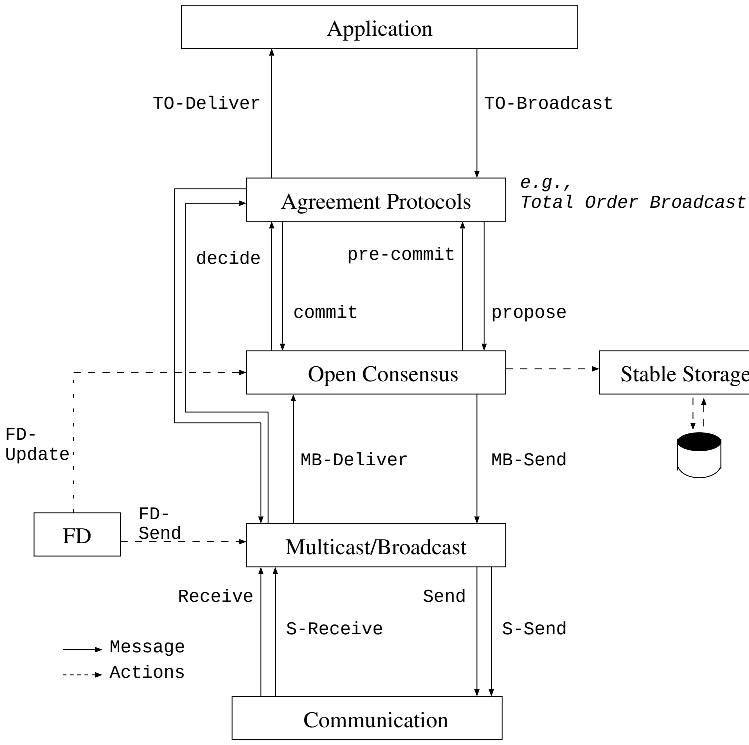


Figure 12. Architecture

6. FRAMEWORK ARCHITECTURE

We sketch in Figure 12 the overall architecture of our abstraction library. The architecture is divided in five layers *Communication*, *Multicast/Broadcast*, *Open Consensus*, *Total Order Broadcast* and *Application*. These are described below. A specific module implements a failure detection scheme and a stable storage module abstracts a hard disk. These components were implemented with SUN's JDK Java 1.2.1 and have been tested on Solaris 2.7. The different layers communicate through method invocation and listeners for upcalls. All messages are buffered in each layer to avoid network bottleneck. For example, if a message cannot be sent because buffers are full, the *Communication* layer notifies the *Multicast/Broadcast* layer which itself notifies its upper layer, and so on.

6.1. Communication

This layer handles point-to-point as well as multi-point communication. The *Communication* layer is based on the model described in Section 2. It uses sockets and affects to each process a unique id. Process ids are taken from an ordered set and both TCP/IP and UDP/IP can be used for communications. For TCP/IP, to decide which process listens to the connection and which one connects, we use a simple scheme where a process with a lower id (acts as a client) connects to a process with a greater id (acts as server). We hence avoid double connections and ensures that each process knows what to do in case of reconnection, in particular in case of recovery. The *Communication* layer has no other functionality besides handling *send* and *receive* events. We give below an excerpt of the corresponding class for TCP/IP.

```
public class Communication extends UnicastRemoteObject {
  protected interface Listener{
              public void receiveMsg(Message m);
              ...
        }
  protected class SocketSender extends Sender {...}
  protected class SocketReceiver extends Receiver {...}
  ....
  public static void closeServer() throws IOException {...}
  public void closeClientChannel() throws IOException {...}
  ../
}
```

6.2. Multicast/Broadcast

This layer handles multicasts and broadcasts messages with different semantics to a process group. The various semantics are: (a) those of the retransmission module defined in Section 2 (`s-send` and `s-receive`), and (b) *simple* sends and receives also defined in Section 2 (`send` and `receive`). The simple `send` makes only one trial to send the message. We have implemented the retransmission module as a thread in the *Multicast/Broadcast* layer. This layer sends and receives messages using the primitives *send* and *receive* of the *Communication layer*. We give below some excerpt of this class.

```
public class MulticastBroadcast implements Communication.Listener {
  protected interface Listener{public void notifyOverwriteException(String error);}
  protected class MulticastBroadcastSender extends Sender {...}
  protected class NetworkReceiver extends Receiver {...}
  ...
  public void notifyOverwriteException(String error) {...}
  public void send(Message m, int[] dst) {...}
  public void s-send(Message m, int[] dst) {...}
  public Message receive(Message m, int dst) {...}
  public void s-receive(Message m, int dst) {...}
  ...
}
```

6.3. Open consensus

This layer implements the open consensus algorithm. The main operations exported by this class are the operation `propose` and `commit`. Several inner classes are used for the implementation of this operation, i.e., for the actual open consensus algorithm. The layer invokes the `Multicast/Broadcast` class to send messages and the `StableStorage` class (*stableStore* function) to store critical fields in a file and retrieve them upon recovery. The `MulticastBroadcast.Listener` interface extends the interface `EventListener`, while the `Sender` and `Receiver` classes extends the class `Thread`.

Each of the inner classes within `OpenConsensus` corresponds to a specific thread involved in the implementation of the operations `propose` and `commit`: (1) a thread `Coordinator` that corresponds to the task *coordinator* described in the open consensus algorithm (lines 28-46 of Figure 4), (2) a thread `Commit` that handles all the commit invocations (lines 21-27), and (3) a thread `Propose` that handles the propose invocations (lines 5-20). The last three classes are not static because they are bound to a single instance of consensus. Finally, class `OpenConsensusSender` (resp. `OpenConsensusReceiver`) treat the messages that need to be sent (resp. received). The class `OpenConsensusReceiver` corresponds to the receive and s-receive primitives, i.e., lines 47-67 of Figure 4. We give below an excerpt of the class `OpenConsensus`.

```
public class OpenConsensus implements MulticastBroadcast.Listener {
  protected static class OpenConsensusSender extends Sender {...}
  protected static class OpenConsensusReceiver extends Receiver {...}
  protected class Coordinator extends Thread {...}
  protected class Commit extends Thread {...}
  protected class Propose extends Thread {...}
  ....
  protected synchronized void stableStore(int[] fields) {...}
  ....
  public int propose(int value) {...}
  public boolean commit(int value) {...}
  ...
}
```

6.4. Total order broadcast

The *TotalOrderBroadcast* layer atomically broadcasts and delivers messages. It invokes the `OpenConsensus` class to solve consensus. As for our `OpenConsensus` implementation, layers communicate via method invocations and listeners for upcalls. Class `TotalOrderBroadcastSender` (resp. `TotalOrderBroadcastReceiver`) handles the messages that need to be sent (resp. received). The class `TotalOrderBroadcastReceiver` corresponds to the lines 12-23 of Figure 10. The thread `Propose` invokes the `OpenConsensus` layer (lines 7-9), while the `to-deliver` primitive implements the TO-Delivery of messages (lines 10-11). The `to-broadcast` primitive is invoked when a programmer desires to TO-Broadcast a message (lines 4-5). We give below an excerpt of the `TotalOrderBroadcast` class.

```
public class TotalOrderBroadcast implements OpenConsensus.Listener {
  protected static class TotalOrderBroadcastSender extends Sender {...}
  protected static class TotalOrderBroadcastReceiver extends Receiver {...}
  protected class Propose extends Thread {...}
  ....
  public void to-broadcast(Messageset msgSet) {...}
  public void to-deliver(int k, Messageset msgSet) {...}
  ...
}
```

6.5. Stable storage

The stable storage module abstracts a hard disk. It is accessed every-time: (a) open consensus needs to store some variable into stable storage, and (b) a process recovers and retrieves its persistent state. We give below an excerpt of this class.

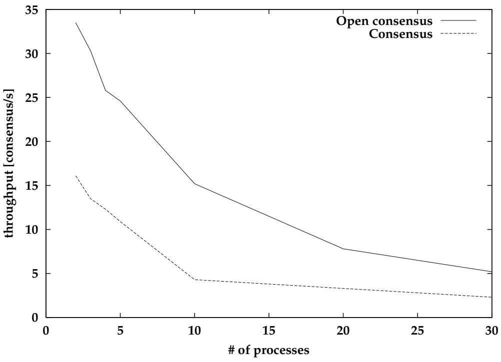
```
public class StableStorage {
  protected String storageFileName;
  ...
  public synchronized void stableStore( int[] fields ) {...}
  public synchronized void stableRetrieve( int[] a ) {...}
  ...
}
```

6.6. Failure detector

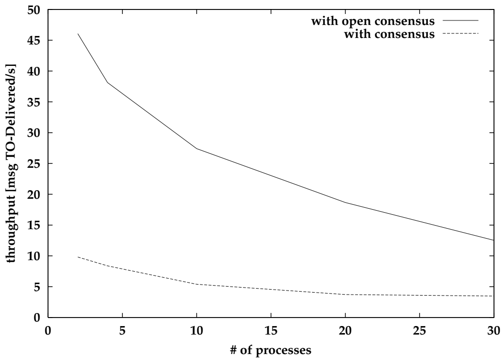
A failure detector abstracts a distributed oracle that provides the processes with hints about crashes [7]. The failure detector Ω is implemented along the lines of $\diamond S_u$ from [1]. The failure detector outputs a *trustlist* at every process. The *trustlist* is a set of processes that are deemed to be currently up. We give below an excerpt from our `FDetector` module. The class `elementTL` contains the processes that are trusted (*trustlist*) by the failure detector. The interface `FDListener` updates the upper layers of changes in the *trustlist*, while the thread `FDSenderThread` keeps on retransmit I_AM_ALIVE messages to every process. When a process suspects a new process or stops suspecting a process, it updates the consensus layer with *FD-Update.*

```
public class FDetector implements MulticastBroadcast.Listener, MulticastBroadcast.FDListener {
  protected class elementTL {...}
  protected interface FDListener {...}
  protected class FDSenderThread extends Sender {...}
  ...
}
```


Prepared using cpeauth.cls



(a) Consensus



(b) Total order broadcast

Figure 13. Throughput comparison

6.7. Experimental Measures

Figure 13 gives the throughput in nice runs of open consensus vs consensus on the one hand, and open consensus based total order broadcast vs consensus based total order broadcast on the other hand. Our performance measures were performed on a LAN interconnected by Fast Ethernet (100MB/s) on a normal working day. The LAN consisted of 60 UltraSUN 10 (256Mb RAM, 9 Gb Harddisk) machines. All stations were running Solaris 2.7, and our implementation was running on Solaris JVM (JDK 1.2.1, native threads, JIT). The effective message size transmitted was of 1Kb. Figure 13(a) compares open consensus and the consensus of [1]. To have a fair comparison, we measure the case where all processes propose and decide. Not surprisingly our comparison depicts the fact that the more forced logs an implementation has, the worse the performance is. We have then implemented two total order broadcasts, one over open consensus and one over consensus: performance results are summarized in Figure 13(b). Again, since open consensus makes less forced logs, the performance of total order broadcast based on open consensus is by far better than the one based on the traditional consensus.

7. CONCLUDING REMARKS

On the one hand, theoreticians have stated and proved fundamental results about the solvability of the *consensus* problem under various system models and assumptions [14, 12, 7, 18]. On the other hand, developers of reliable distributed systems have been focusing on designing and implementing efficient solutions to "practical" agreement problems like *total order broadcast* and *atomic commit* [8, 15, 3, 11, 5, 26]. For a long time, the two research trends have been undertaken separately. Relatively recently, several authors suggested the use of consensus as a basic building block to devise modular solutions to "practical" agreement problems [7, 16, 17, 20]. In particular, it was shown that the use of consensus to solve various agreement problems does not introduce any significant overhead with respect to non-modular agreement algorithms that bypass consensus to solve the very same problems [16]. To convey that result, the authors of [16] considered however a system model where channels are reliable, a majority of the processes remain always up, and processes that crash do never recover.

Nevertheless, consensus, according to its original specification, cannot be effective in a practical crash-recovery system model where processes and channels may crash and recover. This is because the use of consensus introduces inherent additional forced logs (which are known to be major sources of overhead) in comparison with non-modular algorithms that bypass consensus. This issue is conveyed for instance in [25], where the authors describe a total order broadcast for the crash-recovery model, based on a traditional consensus box. The protocol is modular, but rather inefficient in terms of forced logs. This inefficiency is not due to the protocol per se, but to the use of an underlying traditional consensus box. In [23], Lamport presents a total order broadcast in the crash recovery model based on a consensus box, and discusses how to make that protocol efficient, by however breaking the encapsulation of consensus.

The motivation of this work was to propose a reshaping of consensus that makes it effective in such a practical crash-recovery system model. In other words, the aim was to figure out whether we can define a consensus-like box that would preserve modularity and yet enables efficiency. Doing so is however not trivial, precisely because to keep the theoretical benefits of reusing consensus (and all related results), its reshaping should not diminish the inherent algorithmic complexity encapsulated by consensus. We propose in this paper the abstraction of *open* consensus, and we define the precise conditions under which the two problems are equivalent. The use of open consensus is however more efficient. Roughly speaking, our new specification provides consensus with pragmatic *decoupled*, *re-entrant* and *on-demand* flavours. The significant optimisations we obtain in our modular agreement algorithms (in terms of forced logs) are not achieved at the expense of stronger assumptions or additional messages and communication steps, with respect to alternative algorithms that are based on the traditional notion of consensus or simply ad-hoc algorithms [1, 19, 23, 25, 26]. Typically, our *open consensus* abstraction could be used as an underlying building block to devise fault-tolerant middleware service. For example, the central abstraction of the CORBA Object Group Service of [13] is a consensus one. Replacing that abstraction with our new open consensus can help build efficient fault-tolerant services in a practical crash-recovery model.

The flavours of open consensus make it a good candidate to build, not only a modular and efficient total order broadcast algorithm, but also other kinds of agreement algorithms in a modular, yet efficient manner. One can follow the approach of [16] to build a modular yet efficient atomic commit, group membership and view synchronous algorithms. Moreover, it is easy to see how one could easily and efficiently implement the primary-backup scheme of [9] in a crash-recovery model using our open consensus abstraction. In [10], the authors proposed a consensus-based form of primary-backup replication [4]. To make the replication scheme efficient, the authors had however to violate consensus encapsulation by assuming a specific consensus algorithm (the algorithm of [7]), and optimised their replication scheme with that consensus algorithm in mind. More recently (in [9]), the authors replaced the consensus box with a different building block, named *lazy consensus*. The specification (1) assumes that the processes invoke consensus with a function passed as a parameter, and (2) precludes the possibility for two processes to invoke consensus with two different values, unless one of them is suspected to have crashed. The resulting specification is designed for the specific replication technique considered by the authors. Our open consensus specification is more general, yet simpler. It is more general in the following senses. First, in our case, a process does not receive a decision unless it invokes consensus (i.e., our *on-demand* flavour is more general). Second, we introduce additional notions of *re-entrance* and *decoupling*: these notions would help optimise the replication scheme of [9] in terms of forced logs while preserving modularity. Our specification is simpler because we only replace the properties of consensus with slightly different properties of the same nature (termination, agreement and validity), without introducing properties of different natures, e.g., precluding two processes from proposing two values unless one of them is crashed or suspected to have crashed. (Such a property actually restricts the applicability of the specification to the specific asynchronous system model augmented with a failure detector. Even more importantly, it is not clear whether the resulting abstraction is actually equivalent to the actual original consensus abstraction.)

REFERENCES

1. M.K. Aguilera and W. Chen and S. Toueg. Failure Detection and Consensus in the Crash-Recovery Model. Distributed Computing, 13(2):99-125, May, 2000.
2. M.K. Aguilera and W. Chen and S. Toueg. On Quiescent Reliable Communication. SIAM Journal on Computing, 29(6):2040-2073, April, 2000.
3. P.A Bernstein and V. Hadzilacos and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison Wesley, 1987.
4. N. Budhiraja and K. Marzullo and F.B. Schneider and S. Toueg. The Primary-Backup Approach. In S. Mullender, editor, Distributed Systems, ACM Press Books, chapter 8, pages 199-216, Addison-Wesley, second edition, 1993.
5. K. Birman and R. van Renesse. Software Reliability for Networks. Scientific American, 274(5), May, 1996.
6. T.D. Chandra and V. Hadzilacos and S. Toueg. The Weakest Failure Detector for Solving Consensus. Journal of the ACM, 43(4):685-722, July, 1996.
7. T.D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, 43(2):225-267, 1996.
8. D. Dolev and S. Kramer and D. Malkhi. Early Delivery Totally Ordered Broadcast in Asynchronous Environments. In Proceedings of the 23rd IEEE International Symposium on Fault-Tolerant Computing (FTCS-93), Toulouse, France, pages 296-406, June, 1993.
9. X. Défago and A.Schiper. Semi-Passive Replication and Lazy Consensus. Technical Report DSC 2000-27, Department of Communication Systems, Swiss Federal Institute of Technology, May, 2000.

10. X. Défago and A. Schiper and N. Sergent. Semi-Passive Replication. In Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS-98), West Lafayette, USA, pages 43-50, October 1998.
11. P. Ezhilchelvan and R Macedo and S. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. In Proceedings of the 15th IEEE International Conference on Distributed Computing Systems (ICDCS-95), Cambridge, USA, pages 296-306, May, 1995.
12. C. Fetzer and F. Cristian. On the Possibility of Consensus in Asynchronous Systems. In Proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems, Newport Beach, USA, pages 85-91, December, 1995.
13. P. Felber and R. Guerraoui. Programming with Object Groups in CORBA. IEEE Concurrency, 8(1), January-March, 2000.
14. M.J. Fischer and N.A. Lynch and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM, 32(2):374-382, April, 1985.
15. J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.
16. R. Guerraoui and A. Schiper. The Generic Consensus Service IEEE Transactions on Software Engineering, 27(1): 29-41, 2001.
17. R. Guerraoui. Revisiting the relationship between Non Blocking Atomic Commitment and Consensus problems. In Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG-95), Le Mont-St-Michel, France,number 791 in Lecture Notes in Computer Science, pages 87-100, Springer-Verlag, September, 1995.
18. M. Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages and Systems, 13(1):124-149, January, 1991.
19. M. Hurfin and A. Moustefaoui and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS-98), West Lafayette, USA, pages 228-234, October 1998.
20. M. Hurfin and R. Macedo and M. Raynal and F. Tronel. A General Framework to Solve Agreement Problems. In Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS-99), Lausanne, Switzerland, pages 56-65, October 1999.
21. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In S. Mullender, editor, Distributed Systems, ACM Press Books, chapter 5, pages 97-146, Addison-Wesley, second edition, 1993
22. G. Kiczales. Beyond the Black Box: Open Implementation. IEEE Software, January, 1996.
23. L. Lamport. The Part-Time Parliament. Technical Report 49, Systems Research Center, Digital Equipement Corp, Palo Alto, September, 1989, A revised version of the paper also appeared in ACM Transaction on Computer Systems vol.16 number 2.
24. N.A. Lynch. Distributed Algorithms. Morgan Kaufmann, 1996.
25. L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous systems where processes can crash and recover. In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS-00), Taipei, Taiwan, pages 288-295, April, 2000.
26. D. Skeen. NonBlocking Commit Protocols. In ACM SIGMOD International Conference on Management of Data, pages 133-142, 1981.

8. APPENDIX 1: Equivalence between consensus and open consensus

We show here that consensus and open consensus are equivalent in a crash-recovery model (under the assumptions that all processes are well-behaved). First, we describe an algorithm that transforms open consensus into consensus (Figure 14) and then we describe an algorithm that transforms consensus into open consensus (Figure 15). Note that the aim here is not to devise efficient algorithms but rather show that solvability results that are stated on consensus are valid for open consensus and vice-versa.

To distinguish the primitives that define these problems, we denote by *propose* the primitive for consensus, and by *o-propose* and *commit* those for open consensus. For both transformations, we assume that all processes are well-behaved and that a majority of processes are correct.

8.1. Transforming open-consensus to consensus (Figure 14).

This algorithm assumes the existence of an open consensus box. By the definition of consensus, proposing a value coincides with the forced log of its proposition. Process p_i then o-proposes the proposition and waits for the pre-committed value. When p_i pre-commits a value, p_i then decides the value by invoking *commit*(). Note that returning from the *propose*() primitive coincides with the forced log of the decision. Remember also that all correct processes invoke *propose*() since it is an assumption of consensus. When a process crashes and recovers, it checks if it already decided (by testing if the decision is stored), and if so decides. Otherwise, if the process already proposed (by testing if the proposition is stored), it invokes again $o{-}propose()$.

Proposition 13. *The algorithm of Figure 14 satisfies the validity, agreement and termination properties of consensus.*

Proof (sketch). The validity property of consensus is trivial since it is the same validity property of open consensus. Consider now the agreement property. Since every time a process o-proposes, it o-proposes only a value that was proposed earlier, or the value received if it is the first proposition. By the agreement property of open consensus and by the algorithm of Figure 14, the agreement property of consensus is satisfied. Consider now the termination property of consensus. By the definition of the notion of correct process, there is a time after which all correct processes stop crashing and remain always-up. Hence, by the algorithm of Figure 14, there is a time after which every correct process eventually o-proposes some value. By the termination property of open consensus, every correct process eventually returns from $o - propose()$, then finally decides. □

8.2. Transforming consensus to open-consensus (Figure 15).

This algorithm assumes the existence of a consensus box. Basically, every process that o-proposes, invokes *propose*() (this coincides with a forced log) and then sends the value to all processes, to make sure that every process proposes some value. When a process p_i receives

Prepared using cpeauth.cls

procedure propose(v_{p_i}) {*The procedure call coincides with the forced log of (propose(v_{p_i}))*}
  if propose($v_{proposed}$) has occurred **then**
    o-propose($v_{proposed}$)
  else
    o-propose(v_{p_i})
upon receive(*pre-commit*) **do**
  commit(*pre-commit*); **return**(*pre-commit*) {*Upcall coincides with the forced log of the decision*}
upon recovery **do**
  initialisation; **retrieve**(*decision*,propose($v_{proposed}$))
  if *decision* has occurred **then**
    return(*decision*)
  else if propose($v_{proposed}$) has occurred **then**
    o-propose($v_{proposed}$)

Figure 14. Transforming open-consensus to consensus

an initial value, p_i verifies that it did not already proposed and, if so, p_i does not propose the initial value it received but the one it proposed earlier (due to the agreement property of consensus). Otherwise, p_i proposes the received proposition (which coincides with a forced log). Once a process decides, it returns from $o-propose()$. Upon commit, p_i does nothing but returning the decision since it has been already decided. When p_i recovers, p_i retrieves the decision and the proposition if there are any.

Proposition 14. *The algorithm of Figure 15 satisfies the validity, agreement and termination properties of open consensus.*
Proof (sketch). The validity property follows from the validity property of consensus. The agreement property of consensus ensures that a process must not propose with different values. However, since a process stores the proposition and checks to always give the same initial proposition, the agreement property of consensus is never violated and thus satisfied. Consider now the termination property. If a process p_i invokes $o-propose()$, since p_i s-sends the proposition to every process. There is a time after which every correct process stops crashing and remain always-up. By the property of the retransmission module and the algorithm of Figure 15, every correct process proposes the same value and decides. Indeed, p_i returns from the invocation of $o-propose()$. Of course, if p_i crashes, p_i does not need to return. For primitive $commit()$, it is trivial since it only returns the decision. □

```
procedure o-propose(v_{p_i})
  if propose(v_{proposed}) has not occurred then
    propose(v_{p_i})            {The procedure call coincides with the forced log of (propose(v_{p_i}))}
  else
    propose(v_{proposed})
  s-send(propose(v_{p_i}) to all \p_i
upon receive(decision) do       {This upcall coincides with the forced log of the decision}
  return(decision)
upon commit(decision) do
  return(decision)
upon receive propose(v_{p_j}) from p_j do
  if propose(v_{proposed}) has not occurred then
    propose(v_{p_j})            {The procedure call coincides with the forced log of (propose(v_{p_j}))}
  else
    propose(v_{proposed})
upon recovery do
  initialisation; retrieve(decision,propose(v_{proposed}))
```

Figure 15. Transforming consensus to open-consensus