Making Consensus Practical

Romain Boichat^{+} Svend Frølund* Rachid Guerraoui^{+}
$^{+}$ Swiss Federal Institute of Technology, CH 1015, Lausanne
* Hewlett-Packard Laboratories, 1501 Page Mill Rd, Palo Alto

Abstract

This paper presents the abstraction of *lazy consensus* and argues for its use as an effective component for building distributed agreement protocols in practical asynchronous systems where processes and links can crash and recover. *Lazy consensus* looks like consensus, is equivalent to consensus, but is *not* consensus. The specification of lazy consensus has an *on-demand* and a *re-entrant* flavors that makes 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 lazy consensus as a building block to develop efficient atomic broadcast and atomic commitment protocols: two central abstractions in our *DACE* middleware environment.

Keywords: Abstraction, consensus, atomic broadcast, atomic commit, failure detectors, modularity, distribution.

Contact author: Romain Boichat - Romain.Boichat@epfl.ch.

Submission category: Manuscript track (Distributed Computing).[1]

Approximate word count: 10.000 - with appendices.

Declaration: This material has been cleared through author affiliations.

1 Introduction

Motivation. The Motivation of our *DACE (Distributed Asynchronous Collection Environment)* project is the development of a middleware framework for reliable distributed computing. Unlike many middleware projects that aim at achieving distribution *transparency*, our objective is rather to provide *abstractions* that encapsulate the complexity of reliable distributed computing [WWWK94].[2] In other words, our objective is not to hide distribution, but rather provide the programmer with convenient abstractions to write distributed and highly-available programs. In particular, we aim at the development of basic abstractions that support both transactional protocols [BHG87], and group communication protocols [BvR96].

It is indeed widely accepted that abstraction is a good idea, especially when writing distributed and reliable 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 the complexity of a well defined problem, it

[1]Without appendices, the paper could also be considered as a candidate for the regular track.

[2]Whereas *transparency* does not allow you to deal with details, *abstraction* allows you no to deal with details.

must also be *overhead-free*. Namely, the use of that abstraction should introduce an overhead that is negligible with respect to ad-hoc solutions that bypass that abstraction.

The objective of this paper is to describe a *consensus*-like abstraction [FLP85, CT96], named *lazy consensus*, and argue for its use as an effective building block to develop both typical transactional protocols [BHG87], like non-blocking atomic commitment [Ske91] and typical group communication protocols [BvR96] like atomic broadcast [BSS91]. *Lazy consensus* is one of the most fundamental abstractions underlying our DACE middleware framework, currently developed in Java.

Context. Our DACE project builds upon our experience with our *Bast* middleware framework.[3] A central component of Bast was the *consensus* service: an abstract form of agreement where the processes decide on a *common* value among a set of *proposed* values. Although usually considered from a theoretical point of view [FLP85, CT96], consensus can also be viewed as a basic service to build more practical agreement protocols [GS96], e.g., atomic commitment and atomic broadcast. The idea, promoted by several authors [CT96, GS96, Gue95, HMRT99], is very seducing because problems like atomic broadcast and atomic commit 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, and factoring out that part inside a consensus box can drastically simplify the description of the agreement protocols. In short, by considering consensus as a basic abstraction to build 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 badly needed.

Problem. Nevertheless, and as we pointed out, to state if consensus can indeed be an effective abstraction in building agreement protocols, one should figure out whether the use of consensus is *overhead-free* with respect to ad-hoc protocols that bypass consensus.[4] In our Bast framework, we implemented several consensus-based agreement protocols and we pointed out the modularity and efficiency of these protocols [GG98]. However, we considered a *crash-stop* system model: 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 [ACT98] is a realistic system model for most of the applications we know of, but it introduces a fundamental difficulty in layering abstractions.

Basically, 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 an atomic broadcast protocol based on an underlying traditional consensus abstraction [CT96, Lam89]. A consensus-based atomic broadcast protocol typically uses a sequence of consensus, each instance being used to agree on a batch of messages [CT96]. If any process p_i crashes and recovers, p_i might not remember whether or not it proposed a value for consensus instance k. Since the traditional spec-

[3]Bast was initially implemented in Smalltalk [GG98] and later ported to C++ in the form of a CORBA service [FG00].

[4]It is important to precisely define here what we mean by *overhead-free*. Notice first that the use of any abstraction always has a minimal overhead with respect to a solution that bypasses that abstraction: the minimal overhead is simply the cost of a local procedure/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). We focus here on efficiency in *normal* cases, where no process crashes, or is even suspected to have crashed. These are the cases that are the most frequent in practice and for which the protocols are usually optimized.

ification of consensus does not allow any process to propose different values, invoking consensus is typically defined as writing the initial value on stable storage (this is indeed the assumption made in [ACT98]). Upon recovery, the log will help p_i figure out what happened prior to the crash. The very same problem occurs with the decision, which must be defined as writing the final value on stable storage [ACT98]. Forced logs are usually considered very expensive because each one typically involves a synchronous write to the disk. At first glance, one might be tempted to give up the use of a consensus box and develop ad-hoc protocols, or open the box and customize its implementation, i.e., make specific assumptions about the consensus protocol. Neither case brings modularity.

Contribution. This paper suggests a *reshaping* of consensus that makes it better suited for a practical usage. More precisely, we introduce the specification of a new consensus-like problem, which we call *lazy consensus*. Of course, proposing a new specification is fraught with the danger of ending up with a problem that is either stronger than the original problem, or on the contrary trivial. In both cases, we do not have the benefits of reusing well-known results on the solvability of consensus. Fortunately, lazy consensus and consensus are equivalent problems (both in a crash-stop and in a crash-recovery model). However, the specification of lazy consensus has some interesting flavors that makes its use practical. First, lazy consensus has an *on-demand* flavor which basically means that processes do not all need to propose values and receive decisions. If a process is interested in receiving a consensus decision, it must propose a value: otherwise the process just act as a *witness*.[5] Second, lazy consensus has an *re-entrant* flavor: a process may propose different values for the same consensus instance. Typically, the process may propose a value, crash, recover, and then propose a completely different value (for the same consensus instance). The *on-demand* and *re-entrant* flavors are precisely what make lazy consensus practical. We describe a lazy consensus algorithm where, like in [ACT98, HMR98], processes can crash and recover, messages can be lost, and failure detection can be unreliable. A nice characteristic of our lazy consensus algorithm is that, in normal runs, processes need only *one* forced log (per process) before reaching a decision without or additional messages and communication steps. The log is used to preserve agreement and not to store propositions or decisions. As pointed out in [ACT98], *one* forced log is anyway needed to preserve agreement and would anyway be necessary in ad-hoc agreement protocol. To illustrate the usefulness of our lazy consensus abstraction, we describe two agreement protocols that build upon this abstraction: an atomic broadcast and an atomic commit protocol. Both algorithms are simple, modular, and efficient. They have the same communication pattern as protocols designed for a crash-stop model, and none of them require any forced log, beside what is required in the implementation of consensus, i.e., beside what is necessary to preserve agreement.

Roadmap. The paper is organized as follows. We first describe the system model in Section 2. Section 3 introduces the specification of lazy consensus and presents an efficient algorithm that implements that specification. We then describe in Section 4 an atomic broadcast and an atomic commit protocols built on top of lazy consensus. Section 5 discusses the architecture and presents some part of our DACE interface along with some performance measurements. Section 6 concludes the paper with some final remarks.

[5] In this sense, lazy consensus is similar to the consensus object [Her91].

2 Model

We consider a set of processes $\Pi = \{p_1, p_2, ..., p_n\}$. 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). While being up, a process can fail by crashing; it then stops working and becomes *down*. A process that is down can later recover; it then becomes up again and restarts by executing a recovery procedure. The occurence of a *crash* (resp. *recovery*) makes this 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 equiped 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 not affected by crashes. Processes communicate and synchronize by sending and receiving messages through channels. We assume a bidirectionnal channel between each pair of processes. Channels can lose or drop messages but ensure 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, then p_i has sent m to p_j.
- *Finite duplication:* if p_i sends a message m only finite number of times, then p_j receives m from p_i only a finite number of times.
- *Fair loss:* if p_i sends messages to a correct process p_j an infinite number of times, then p_j receives messages from p_i an infinite number of times.

A failure detector is a distributed oracle that provides the processes with hints about crashes [CT96]. We consider the same failure detector $\diamond S_u$ as [ACT98]. This failure detector outputs a *trustlist* (set of processes) and an *epoch* number (vector of numbers indexed by the elements of *trustlist*) that represents a rough estimate about how many times the processes have crashed and recovered. $\diamond S_u$ has the following properties [ACT98]:[6]

- *Monotonicity:* At every correct process, eventually the epoch numbers are nondecreasing.
- *Completeness:* For every faulty process b, at every correct process there is a time after which either b is never trusted or the epoch number of b keeps increasing.
- *Strong Accuracy:* Some correct process is eventually trusted by all correct *and unstable* processes, and its epoch number stops changing.

3 Lazy Consensus

This section recalls the specification of consensus, introduces *lazy consensus*, and describes an algorithm that implements it. The correctness proofs of the our lazy consensus algorithm are given in Appendix A.

[6]As shown in [ACT98], one can implement the failure detector $\diamond S_u$ in a partially synchronous system model, i.e., $\diamond S_u$ encapsulates the synchrony of the system.

3.1 Specifications

In the consensus problem, every process is supposed to *propose* a value [FLP85], and the processes need to satisfy the following properties [ACT98]:

Termination: Every correct process eventually decides.

Agreement: No two processes decide differently.

Validity: If a process decides v then some process has previously proposed v.

As we pointed out in the introduction, the very specification of consensus introduces an important overhead in its use in a crash-recovery model. Basically, every proposal and every decision should be logged on stable storage [ACT98]. In [ACT98, HMR98], the authors describe consensus protocols for a crash-recovery model, and indeed assume that every invocation and every decision of consensus coincides with a forced log. Hence, beside at least a forced log that is anyway needed to preserve agreement, additional forced logs are needed for the interaction with the consensus box: these introduce a *pure* overhead to the consensus abstraction. In [RR99], the authors describe an atomic broadcast protocol that relies on a consensus box in a crash-recovery model. The protocol is efficient in terms of messages and communication steps[7], but to cope with recovery, every process needs to perform "at least" two forced logs before delivering a decision, even in a normal execution. As pointed out by the authors, the inefficiency of the scheme is inherent to the use of consensus as a black-box.

We introduce here a new consensus problem, which we call *lazy consensus*. This problem requires the processes to ensure the *agreement* and *validity* properties above, plus the following *termination* property:

Termination: If a process proposes a value, then unless it crashes, it eventually decides some value.

Only the *termination* property differs from the original properties of consensus [ACT98]. In lazy consensus, a process might need to propose several times before deciding, i.e., if it crashes and recovers. However, the process might propose different values and yet get a "valid" decision - lazy consensus is in this sense *idempotent*. Furthermore, lazy consensus does not require all processes to propose a value. The processes that do not propose can participate in the consensus implementation as "*witnesses*", but do not need to receive any decision - lazy consensus has an *on-demand* flavor.

Our termination property sounds indeed weaker that the original termination property of consensus. However, a closer look at the properties is sufficient to see that any algorithm that solves consensus can be transformed to solve lazy consensus, and vice-versa (we prove the equivalence in Appendix B). Interestingly, and as we show in the following, lazy consensus has an efficient protocol and, as we show in Section 4, it constitutes a useful building block to develop efficient agreement protocols.

3.2 Algorithm

Figure 1 describes a lazy consensus protocol. As in [ACT98], we use the failure detector $\diamond S_u$, and we assume a majority of correct processes. However, our algorithm differs from [ACT98] in the way the consensus starts, how it ends, and the number of logs it involves. Not all, but

[7]The protocol has the same communication pattern than the protocol of [CT96], which was designed for a crash-stop model.

```
1:  for each process $p_i$:
2:  **procedure** initialization:
3:     **for all** $p_j \in \Pi \setminus p_i$ **do**
4:        xmitmsg[$p_j$] $\leftarrow \bot$
5:     *decided*,*interested* $\leftarrow \bot$; *proposed* $\leftarrow$ **false**; $(r_{p_i}, estimate_{p_i}, ts_{p_i}) \leftarrow (1,\bot,0)$;
6:  **procedure** s-send$_{p_i}$(m)                                          {*to s-send m to $p_j$*}
7:     **if** $p_j \neq p_i$ **then**
8:        xmitmsg[$p_j$] $\leftarrow m$; **send** $m$ to $p_j$
9:     **else**
10:       simulate **receive** $m$ from $p_i$
11: **task** retransmit
12:    **while** true **do**
13:       **for all** $p_j \in \Pi \setminus p_i$ **do**
14:          **if** xmitmsg[$p_j$] $\neq \bot$ **then**
15:             **send** xmitmsg[$p_j$] to $p_j$
16: **upon** propose($v_{p_i}$) **do**
17:    *proposed* $\leftarrow$ **true**
18:    **wait until task** *participant* **and** *coordinator* are **not active**
19:    **if** *decided* $= \bot$ **then**                                   {*otherwise has decided meanwhile*}
20:       $estimate_{p_i} \leftarrow v_{p_i}$; **fork task**{4phases,retransmit}     {*to avoid starting the same task more than once*}
21: **task** 4phases
22:    $c_{p_i} \leftarrow (r_{p_i} \bmod n)+1$; **fork task**{skip_round,participant}
23:    **if** $p = c_{p_i}$ **then**
24:       **fork task**{coordinator}
24: **task** coordinator
25:    *interested* $\leftarrow \bot$
26:    Phase NEWROUND
27:       **if** $ts_{p_i} \neq r_{p_i}$ **then**
28:          S-SEND($r_{p_i}$,NEWROUND) to all
29:          **wait until** [received($r_{p_i}, estimate_{p_j}, ts_{p_j}$,ESTIMATE) from $\lceil (n+1)/2 \rceil$ processes]
30:          $t \leftarrow$ largest $ts_{p_j}$ such that $p_i$ received ($r_{p_i}, estimate_{p_j}, ts_{p_j}$,ESTIMATE)
31:          $estimate_{p_i} \leftarrow$ select one $estimate_{p_j}$ such that $p_i$ received ($r_{p_i}, estimate_{p_j}, t$,ESTIMATE)
32:          $ts_{p_i} \leftarrow r_{p_i}$; **store**{$r_{p_i}, estimate_{p_i}, ts_{p_i}$}
33:    Phase NEWESTIMATE
34:       S-SEND($r_{p_i}, estimate_{p_i}$,NEWESTIMATE) to all
35:       **wait until** [received($r_{p_i}$,ACK,*proposed*) from $\lceil (n+1)/2 \rceil$ processes]
36:             **if** *proposed* is **true** in received($r_{p_i}$,ACK,*proposed*) **then**
37:                *interested* $\leftarrow$ *interested* $\cup\ p_j$
38:       S-SEND($estimate_{p_i}$,DECIDE) to *interested*
39:    **terminate task**{4phases,participant,coordinator}                     {*coordinator finishes here*}
40: **task** participant
41:    Phase ESTIMATE
42:       **if** $ts_{p_i} \neq r_{p_i}$ **then**
43:          S-SEND($r_{p_i}, estimate_{p_i}, ts_{p_i}$,ESTIMATE) to $c_{p_i}$
44:          **wait until** [received($r_{p_i}, estimate_{c_p}$,NEWESTIMATE) from $c_{p_i}$ ]
45:          **if** $p \neq c_{p_i}$ **then**
46:             $(estimate_{p_i}, ts_{p_i}) \leftarrow (estimate_{c_p}, r_{p_i})$; **store**{$r_{p_i}, estimate_{p_i}, ts_{p_i}$}     {*stored only by participants*}
47:    Phase ACK
48:       S-SEND($r_{p_i}$,ACK,*proposed*) to $c_{p_i}$
49:    **if** $p \neq c_{p_i}$ **then**
50:       *proposed* $\leftarrow$ **false**; **terminate task**{4phases,participant}          {*witnesses finish here*}
51: **task** skip_round
52:    $d \leftarrow D_{p_i}$                                                      {*query $\diamond S_u$*}
53:    **if** $c_{p_i} \in d.trustlist$ **then**
54:       **repeat** $d' \leftarrow D_{p_i}$                                         {*query $\diamond S_u$*}
55:       **until** [$c_{p_i} \notin d'.trustlist$ **or** $d.epoch[c_{p_i}] < d'.epoch[c_{p_i}]$ **or** received some message ($r$,...) such that $r > r_{p_i}$]
56:    **terminate task** {4phases,participant,coordinator}
57:    $d \leftarrow D_{p_i}$ **until** $d.trustlist \neq \emptyset$
58:    $r_{p_i} \leftarrow$ the smallest $r > r_{p_i}$ such that $[(r \bmod n)+1] \in d.trustlist$ and $r \geq max\{r' | p$ received$(r', ...)\}$
59:    **fork task**{4phases}
60: **upon receive** $m$ from $p_j$ **do**
61:    **if** $m = (estimate$,DECIDE) **and** *decided* $= \bot$ **then**
62:       *decided* $=$ *estimate*; **stop task**{skip_round}
63:    **if task** participant is **dormant then**
64:       **fork task**{4phases,retransmit}                                      {*to wake up processes*}
65: **upon recovery do**
66:    INITIALIZATION()
67:    **retrieve**{$r_{p_i}, estimate_{p_i}, ts_{p_i}$}
```

Figure 1: Lazy consensus using $\diamond S_u$

only one process is required to start the consensus by proposing some value v, while allowing any process to propose.

Since we do not force a process to always propose the same value, a client of the lazy consensus abstraction does not need to log the propositions. Similarly, the client does not need to log the decision: it *reruns* consensus if needed. Proposing different values for a consensus does not affect the validity of *decision* since $\lceil (n+1)/2 \rceil$ processes log the *estimate* of the decision on stable storage. Also, we do not send the decisions to processes that did not propose any value. A process is either in an *active* or *passive*; when active it executes some code, otherwise it is *dormant* and waits to be waken up by some consensus message.[8] Our algorithm is particularly efficient (logs and messages) in normal runs - where no process crashes or is suspected to have crashed.

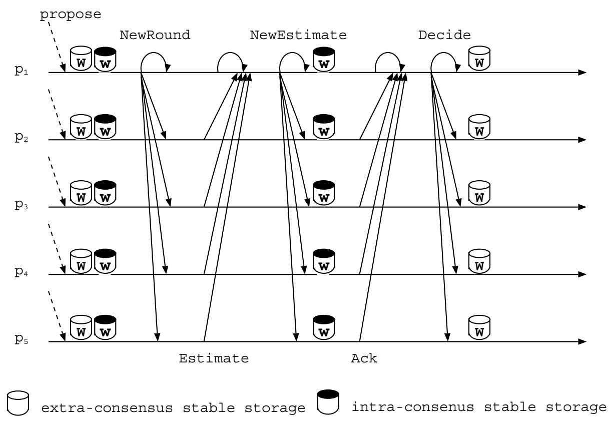
The basic structure of the algorithm of Figure 1 consists of rounds of four phases (the algorithm uses the idea of the rotating coordinator). First, the coordinator c broadcasts a NEWROUND message to initiate a new round (line 28). Then c waits for the ESTIMATE of $\lceil (n+1)/2 \rceil$ processes and chooses the most recent ESTIMATE (using ts_{p_i} in line 31). The chosen estimate is stored in stable storage (forced log) and sent by c to all processes in the third phase with a NEWESTIMATE message, as depicted in line 34. The witnesses wait to receive the NEWESTIMATE message and store the received *estimate* in stable storage (line 46). The witnesses have then finished their duties, and stop all tasks besides *retransmit* (line 50).[9] They enter a *passive* or *dormant* state where processes will be woken up by either a message from a consensus round or a *propose* message. A process is *passive* or *dormant* when it has finished executing tasks *coordinator* and/or *participant*. Finally, the coordinator waits for $\lceil (n+1)/2 \rceil$ processes to acknowledge the chosen *estimate*, and sends the *decision* to the processes that have proposed (these processes are gathered in the variable *interested* as depicted in line 37). The coordinator then finally stops all tasks, except *retransmit*, and enters a passive state.

A round r can be interrupted by a task called *skip_round* starting at line 51 (which runs in parallel with tasks *coordinator* and *participant*): a process p_i aborts its execution of round r if (1) it suspects the coordinator c of round r, or (2) it trusts c but detects an increase in the epoch number of c, or (3) it receives a message from a round $r' > r$. When p_i aborts round r, p_i jumps to the lowest round $r' > r$ such that p_i trusts the coordinator of round r' and p_i has not (yet) received any message with a round number higher than r'. We stop task *skip_round* (line 62) only when the decision has been received (instead of line 50 with the other tasks) to avoid process to hang on decisions. Suppose for instance that the first coordinator sends out a NEWESTIMATE message, all participants send their acknowledgments in line 47, and then would kill skip_round in line 50. If the coordinator crashes before sending a DECIDE message, all processes interested will hang without a decision since task *skip_round* will not suspect the coordinator.

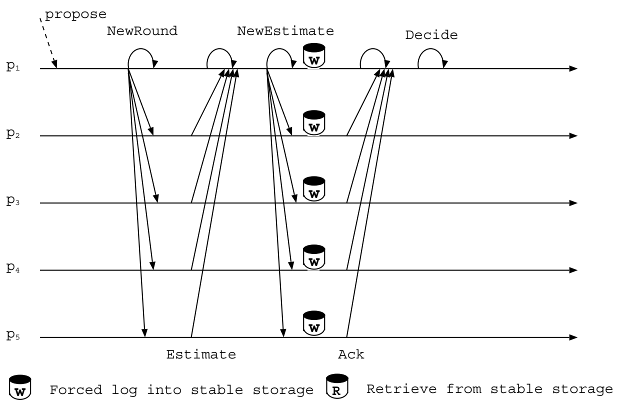
In each round, a process p_i accesses the stable storage only once to store the estimate of the current round. Upon recovery, the process retrieves the latest estimate (line 67) and waits either to be woken by (1) a *propose* or (2) a regular message from other processes involved in a consensus. A process that crashes and recovers has no idea if it was a witness or a coordinator. Neither does it remember if it has *proposed* some value nor if it still needs to *decide*. When receving a message (except a DECIDE message), a process restarts task *4phases*. Note that we assure atomic access to all shared variables since we have concurrent tasks.

[8]We say that a process is dormant even when the task *retransmit* is executed (since this task never stops

[9]Our lazy consensus algorithm is not *quiescent* [ACT97] since we do not stop the task *retransmit* in order to be able to deliver the DECIDE messages. This is due to the unreliable channel properties; over reliable channel properties, we could stop task *retransmit*: but this would mean that the lower layer does the retransmission.



(a) [ACT98] with no crash



(b) lazy consensus with no crash

Figure 2: Comparison in a normal run

3.3 Evaluation

Here we compare our lazy consensus algorithm with the two other consensus algorithms we know of in the crash-recovery model, namely, the algorithms of [ACT98] and [HMR98].

Consider the algorithm of [ACT98]. As depicted in Figure 2(a) and Figure 2(b), in a normal run, the number of communication steps needed to reach a decision is the same in both algorithms. However, the number of forced logs in our algorithm is by far smaller than [ACT98]. We require at least one forced log per process, whereas [ACT98] requires at least four. Beside eliminating the logging of the proposed and the decided value, we also eliminate the logging of the round number, i.e., r_{p_i}. Basically, when recovering after a crash, ts_{p_i} will always be equal to r_{p_i}, except if p_i crashed before logging its first *estimate*. Even if the coordinator sends a NEWESTIMATE message directly instead of waiting for new estimates, a process will receive a new message from a higher round and catch up with other processes. Thus, the round number will be automically updated. Moreover, our lazy consensus algorithm introduces fewer messages than [ACT98] in all configurations, except when all processes propose a value. In this case, the number of messages is the same.

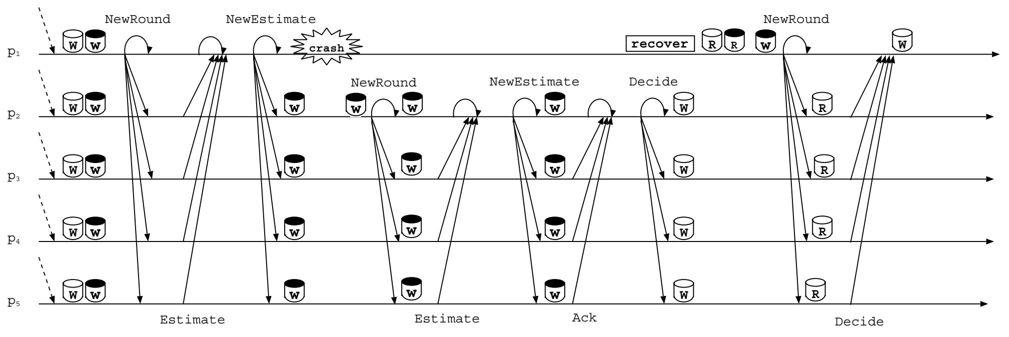


Figure 3: [ACT98] with a coordinator crash

Even if *lazy consensus* is optimized for normal runs, it behaves quite well if a process crashes. For example, if the coordinator crashes, as shown in Figure 3 and Figure 4, lazy consensus is more efficient than [ACT98] both message-wise and log-wise.[10]

Consider now the algorithm of [HMR98]. First, we assume a failure detector with unbounded outputs (namely $\Diamond S_u$), whereas [HMR98] assume a failure detector with bounded outputs. As pointed out in [ACT98], failure detectors with bounded outputs, as assumed in [HMR98], do not really encapsulate the synchrony of the system and have an inherent flaw.[11] Second, the algorithm of [HMR98] is inherently decentralized and optimized for the case where all processes need to receive a decision, whereas our algorithm is optimized for the case where few processes need to decide in normal runs. Finally, the algorithm of [HMR98] solves traditional consensus and hence needs two additional forced logs to store the proposal and the decision. It would be interesting to see how one could obtain a decentralized algorithm that solves lazy consensus relying on the failure detector $\Diamond S_u$.

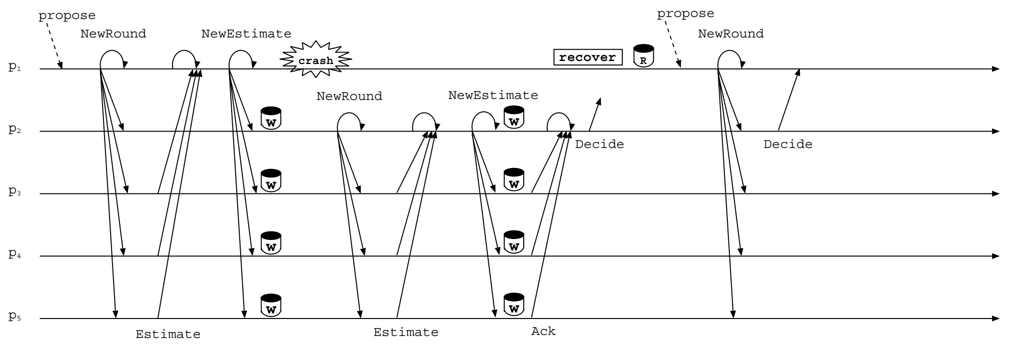


Figure 4: lazy consensus with a coordinator crash

4 Putting lazy consensus to work

This section illustrates the effective use of lazy consensus through two examples: an atomic broadcast and a (non-blocking) atomic commit protocols. The correctness proofs of the protocols are given in Appendix A.

4.1 Atomic Broadcast

Atomic Broadcast is a communication primitive that allows processes to broadcast and deliver messsages in such a way that they agree on both the set of messages they deliver and the order of message deliveries. We specify the underlying problem, in a crash-recovery model, with the following properties:

Termination: If a process *A-Broadcast*s a message m, then unless it crashes, it eventually *A-Delivers* m.

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

[10]To speed up our lazy consensus protocol further in the cases of crashes and recoveries, a process could even send a DECIDE message when it holds the decision and would thus avoid to rerun a consensus. For presentation simplicity, we do not discuss the optimization here because our objective is to optimize normal runs.

[11]In a crash-recovery model, the completeness and the accuracy properties of the failure detectors are contradictory. Note however that one can devise an algorithm that has the same decentralized flavor as [HMR98] and yet relies on $\Diamond S_u$.

Validity: For any message m, every process *A-Delivers* m only if m was previously *A-Broadcast* by *sender(m).*

Total order: If two processes p_i and p_j *A-Deliver* m and m', then p_i *A-Delivers* m before m' if and only if p_j *A-Delivers* m before m'.

[CT96] have shown that atomic broadcast and consensus are equivalent problems in asynchronous systems prone to process crash (no-recovery) failures. In particular, an algorithm was given to transform consensus into atomic broadcast. It has been shown in [RR99] that this algorithm can be adapted to crash-recovery model. Nevertheless, the use of traditional consensus as a building block results is inefficient since it introduces useless forced logs. Our algorithm, described in Figure 5, is an efficient adaptation of the transformation of [CT96] to a crash-recovery model using lazy consensus. Thanks to the on-demand and re-entrant flavours of lazy consensus, our transformation does not require any forced log to [CT96] (beside what is needed inside lazy consensus). Our atomic broadcast algorithm is exactly the same as the one of [CT96], plus the two lines 17 and 18 and the semantics of *R-Broadcast* (reliable broadcast). For modularity purposes and to keep the protocol close to the original one, we assume the existence of a reliable broadcast primitive that, informally, ensures the following properties: a) If a process *R-Broadcast* a message m, then unless it crashes, it *R-delivers* m ; and b) If a process *R-delivers* m, then unless it crashes, all correct processes *R-deliver* m. This *R-Broadcast* can be implemented as the gossip function in [RR99], for instance.

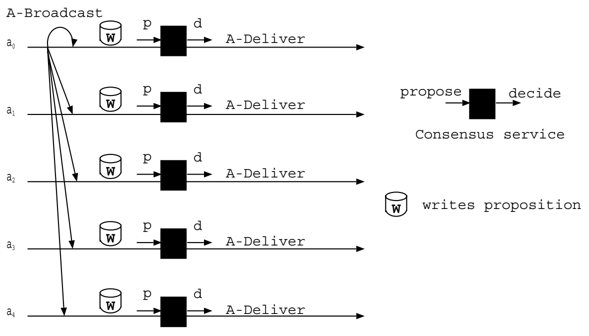
```
1: Every process $p_i$ executes the following:

2: Initialization:
3:    R_Delivered ← ∅; A_Delivered ← ∅; k ← 0; A_Undelivered ← ∅; A_Deliver ← ∅;
4: to execute A-Broadcast($m$)                                          {Task 1}
5:    R-Broadcast($m$)
6: upon R-Deliver($m$) do                                               {Task 2}
7:    R_Delivered ← R_Delivered ∪ $m$
8: upon A-Deliver($k$) do                                               {Task 3}
9:    while R_Delivered - A_Delivered ≠ ∅ do
10:       $k \leftarrow k + 1$
11:       A_Undelivered ← R_Delivered − A_Delivered
12:       L-PROPOSE($k$, A_undelivered)
13:       wait until[received(decide($k$, $msgSet^k$))]
14:       $A\_Deliver^k$ ← $msgSet^k$ - A_Delivered
15:       atomically deliver all messages in $A\_Deliver^k$ in some deterministic order
16:       A_Delivered ← A_Delivered ∪ $A\_Deliver^k$
17: upon recovery do                                                    {added}
18:    Initialization                                                   {added}
```

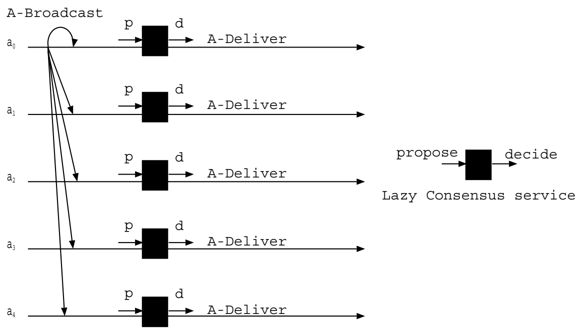
Figure 5: Atomic broadcast with lazy consensus

When a process *A-Broadcasts* a message, it *R-Broadcasts* the messsage m to all processes (Task 1). Once a process receives a message, it appends it to *R_delivered* (Task 2). When p_i *A-Delivers* a message m, p_i adds m to the set *A_delivered* (line 16). Thus *R_delivered* - *A_delivered*, denoted *A_undelivered* (line 11), is the set of messages that p_i *R-Delivered* but not yet *A-Delivered*. Intuitively, these messages are the ones that were submitted for Atomic Broadcast and are not yet *A-Delivered* (Task 3). In Task 3, process p_i periodically checks whether *A_undelivered* contains messages. If so, p_i enters its next execution of consensus by proposing *A_undelivered* as the next batch of messages to be *A_delivered*. Process p_i then waits for the decision of consensus, denoted by $msgSet^k$. Finally, p_i *A-Delivers* all messages in $msgSet^k$ except those it already *A-Delivered*.

The only assumption we make, beside the existence of a lazy consensus box and a reliable broadcast primitive, is the fact that processes keep broadcasting new messages. This is necessary if a process crashes and recovers, in order to start proposing again. If the process does not receive new messages, *R_delivered* will be empty and so will *R_delivered* - *A_delivered*; line 10 in Figure 5 will never be executed, thus contradicting the agreement property of atomic broadcast.



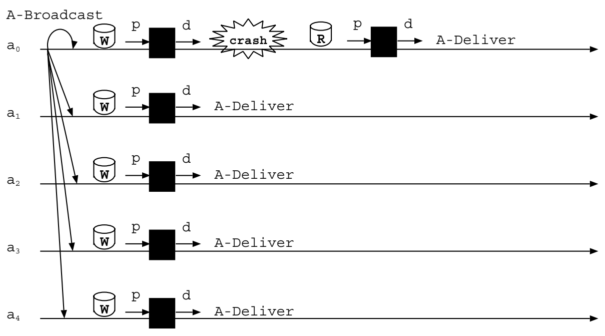
(a) Atomic broadcast with traditional consensus ([RR99])



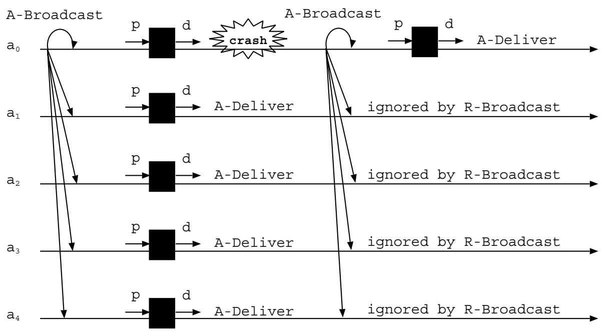
(b) Atomic broadcast with lazy consensus

Figure 6: Comparison in a normal run

We compare below our algorithm with the only atomic broadcast algorithm we know of in a crash-recovery model, namely, the algorithm of [RR99]. Both algorithms have the same communication pattern in normal runs. As shown in Figure 6(a), our protocol does not introduce any forced log, beside what is required within the implementation of lazy consensus. In [RR99], and because of the use of a traditional consensus box, each process needs to log the proposed value (Figure 6(b)), even in normal runs. The behavior of both protocols is different in the case of recovery. As depicted in Figure 7(a) and Figure 7(b), we do not need to access stable storage, but we introduce more messages to achieve recovery. In a recovery scenario, our approach is minimal in terms of number of logs but not message-wise.



(a) Atomic broadcast with traditional consensus ([RR99])



(b) Atomic broadcast with lazy consensus

Figure 7: Comparison in a run with crash-recovery

4.2 Atomic Commit

In this section, we show how a modular non-blocking atomic commit protocol can be easily built using our *lazy consensus* service. The commit problem is typically solved among a transaction manager and a set of data managers. The transaction manager, denoted TM, initiates a transaction and issues read and write operations to *data manager* processes [BHG87]. At the end of the transaction, the TM together with the data managers, must decide on the *commit* or *abort* outcome of the transaction. We consider here the *non-blocking* atomic commit problem where correct processes must eventually decide despite failures [Ske91]. The outcome of the transaction depends on votes from the data managers. A data manager votes *yes* to indicate that it is able to make the temporary writes permanent, and votes *no* otherwise. If the outcome of the atomic commit protocol is *commit*, then all the temporary writes are made permanent; if the outcome is *abort*, then all temporary writes are ignored. The problem is specified, in a crash-recovery model, using the following properties.

Termination: If a process *votes*, then unless it crashes, it eventually decides.

Uniform agreement: No two processes decide differently.

Validity: A process decides *commit* only if all processes vote *yes*.

Non-Triviality: If all processes vote *yes*, and no process is suspected to have crashed, then all correct processes eventually decide *commit*.[12]

The algorithm we propose is very simple, thanks to the use of the lazy consensus box, and a reliable broadcast primitive (as introduced in the previous section). TM waits for all *non-suspected* processes to vote. If all processes vote yes, the TM proposes *commit* to lazy consensus. Otherwise it proposes *abort*. Once the decision has been reached, TM *R-Broadcasts* the decision to all processes. A process p_i *AC-Decides* when it receives a decision at line 30. All tasks can then be stopped. For any other process p_i, p_i S-SENDs its vote and waits for the decision until it suspects TM. If p_i suspects TM, it proposes *abort* to lazy consensus, waits for the decision, and stops all tasks. Since the decision is "stored" in the *lazy consensus* service, the decisions will always be identical. In a normal run, the performance of our protocol is similar to the communication scheme of the well known Three Phase Commit protocol (3PC) [Ske91]. Our protocol is simply more modular.

5 Architecture

We sketch in this section the overall architecture of our DACE agreement library and we give some excerpt of our class interfaces and some performance measurements. The architecture is divided in three layers *Communication*, *Multicast/Broadcast* and *Consensus*. These are described below. A failure detector module implements $\diamond S_u$ and 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 listeners and messages are buffered in each layer to avoid network bottleneck. For example, if a message cannot be sent because buffers are full, the MultiCast/Broadcast and Consensus layers are notified.

Communication. This layer handles point-to-point as well as multipoint communication schemes. The *Communication layer* is based on the model described in Section 2. It uses sockets

[12]This property is that the of the weak-non-blocking atomic commit protocol as introduced in [Gue95]. We believe this specification captures the practical view of atomic commitment: when a TM time outs a database server, it *aborts* the transaction.

```
for each process $p_i$:
procedure initialization:
  for all $p_j \in \Pi \setminus p_i$ do
    xmitmsg[$p_j$] $\leftarrow \perp$
  *proposition* $\leftarrow$ true
procedure s-send$_{p_i}$(m)                                        {*to s-send m to $p_j$*}
  if $p_j \neq p_i$ then
    xmitmsg[$p_j$] $\leftarrow m$; send $m$ to $p_j$
  else
    simulate receive $m$ from $p_i$
task retransmit
  while true do
    for all $p_j \in \Pi \setminus p_i$ do
      if xmitmsg[$p_j$] $\neq \perp$ then
        send xmitmsg[$p_j$] to $p_j$
task A-Commit
  *proposition*=true; *counter* $\leftarrow 0$
  if $p_i$ is $TM$ then
    S-SEND(vote,VOTE) to $TM$
    wait until [received(vote,VOTE)] from all processes $p_i$ except those that are suspected      {*query D*}
        when received a vote: *proposition* $\leftarrow$ *proposition* and vote; *counter* $\leftarrow$ *counter* +1
    if proposition and *counter* = n then                          {*if all processes vote yes*}
      L-PROPOSE(*commit*)
    else
      L-PROPOSE(*abort*)
    wait until [received(decide)]
    *R-Broadcast*(decide) to all $p_i \in \Omega$
  else
    S-SEND(vote,VOTE) to $TM$
    wait until [received(decide)] from $TM$                         {*AC-Decides*}
    terminate task{*retransmit,suspect*}
task suspect
  while $TM \notin$ *d.untrusted* do                                 {*query D*}
    terminate task *A-Commit*
    L-PROPOSE(*abort*)
    wait until [received(decide)]
    terminate task *retransmit*
upon recovery do
  INITIALIZATION
```

Figure 8: (Non-blocking) Atomic commit with lazy consensus

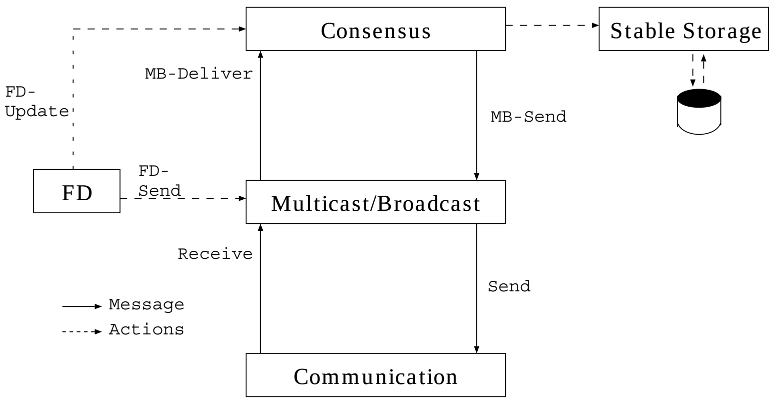


Figure 9: Architecture

and each process has a unique id. Process ids are taken from an ordered set and TCP/IP is used for communications. To decide which process listen to the connection and which one connects, we use the simple scheme that process with a lower id (acts as a client) and connects to process with a greater id (acts as server). This way, we avoid double connections and each process knows what to do in case of reconnection. The Communication layer has no other functionnality beside handling send and receive events.

Multicast/Broadcast. This layer multicasts and broadcasts messages with different semantics to a process group. The various semantics are *reliable or simple* sends and receives. The primitive is *reliable* in the sense of [CT96], while simple send makes only one trial to send the message. For consensus, we use only the *simple send* and receive primitives which try only to send a message once. This layer sends and delivers messages using the *Communication layer* with the primitives *Send* and *Receive*.

Consensus. In our context, we view consensus as an application layer. This layer makes calls to the Multicast/Broadcast layer through primitives *MB-Send* and *MB-Deliver*. We have implemented the task retransmit as a thread in the Consensus layer. This layer is made up of four more threads {*4phases, coordinator, participant, skip_round*} (presented in Figure 1). The Consensus layer also accesses the *Stable Storage* module that executes writes and reads on a hard disk. It is accessed everytime (a) consensus needs to log some variable into stable storage and, (b) a process recovers and retrieves its persistent state.

Failure Detector. The failure detector approximates $\diamond S_u$ based on $\diamond S_e$ following the approach of [ACT98]. The failure detector module at every given process sends I_AM_ALIVE messages to all messages through the Multicast/Broadcast layer with *FD-Send*. When a process suspects a new process or stops suspecting a process, it updates the consensus layer with *FD-Update*.

5.1 Class Interfaces

We present here the three interfaces (excerpt) of each layer, along with the failure detector and the stable storage interfaces.

```
public class Consensus implements MulticastBroadcast.Listener {
  protected class ConsensusSender extends Sender {...}
  protected class ConsensusReceiver extends Receiver {...}
  ....
  public void notifyOverwriteException(String error) {...}
  public void updateSuspectedPrss(int nbPrss) {...}
  public synchronized void stableStore( int[] fields ) {...}
  ...
}

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 stubbornSend(Message m, int[] dst) {...}
  public void reliableSend(Message m, int[] dst) {...}
  public Message receive() {...}
  ...
}
```

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

public class FDetector implements MessagePassingLayer.Listener, MessagePassingLayer.FDListener {
  protected class elementTL {...}
  protected interface FDListener {public void  updateLastReceived(Message m);}
  protected class FDSenderThread extends Sender {...}
  ...
  public void notifyOverwriteException(String error) {...}
  ...
}

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

5.2 Performance

We give here the performance measurements of our lazy consensus implementation in DACE. These were made 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 performing on Solaris JVM (JDK 1.2.1., native threads, JIT). The effective message size transmitted was of 1Kb. These tests consider normal runs: no process crashes or is suspected to have crashed. Figure 10 depicts the number of consensus per second executed for each type of consensus (lazy, ACT98, ACT98 improved). Not surprisingly our comparison convey the fact that the more logs a consensus has, the worst the performance is. We have implemented two versions of [ACT98], one that sticks to the original pseudo-code given in [ACT98] and one improved version that only log the round r_{p_i} together with the $estimate_{p_i}$ thus saving one log (this is a straightforward optimization of ACT98a).

6 Conclusion

Agreement problems are at the heart of reliable distributed computing, and a lot of effort has been devoted to those problems, from both a theoretical and a practical point of view. Theoreticians have stated and proved fundamental results about the solvability of the *consensus* problem under various system models and assumptions [FLP85, CF95, CT96, Her91].

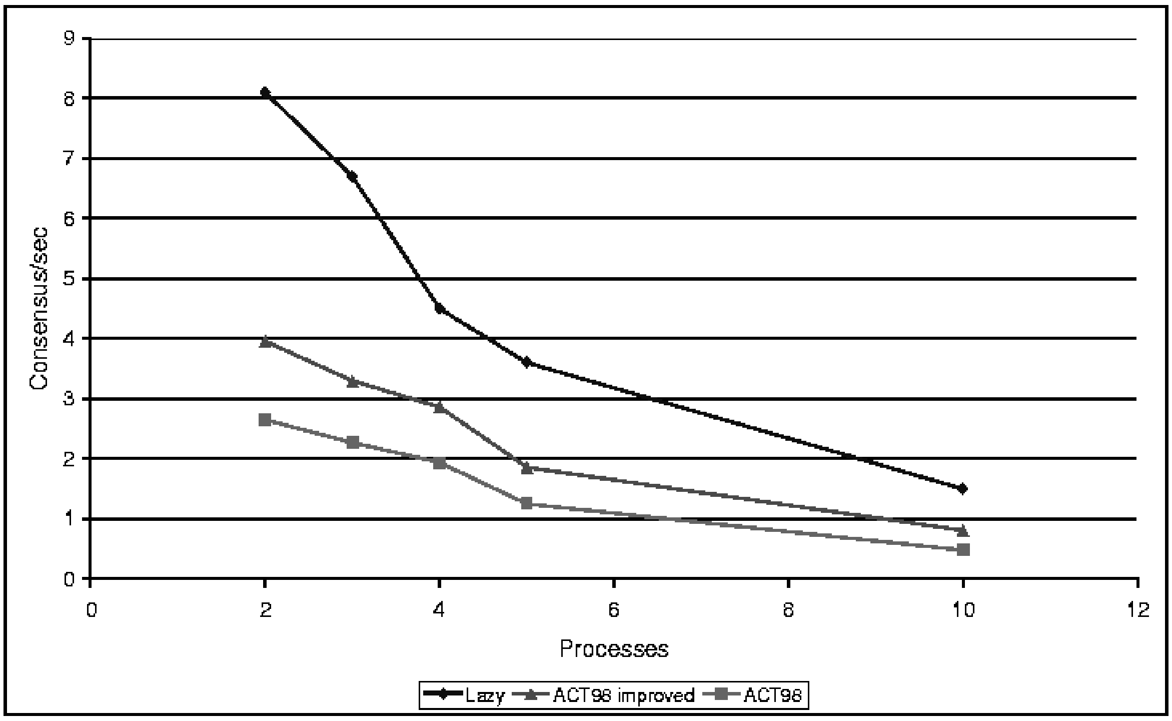


Figure 10: Consensus comparison

Developers of reliable distributed systems have been focusing on designing and implementing efficient solutions to "practical" agreement problems like *atomic broadcast* and *atomic commit* [Anc97, BSS91, DKM93, GR93, BHG87, EMS95, BvR96, Ske91, NMMS97, OP98]. In atomic broadcast, the processes need to *agree* on the sequence of messages they deliver, whereas in atomic commit the processes need to *agree* on outcomes of distributed transactions (*commit* or *abort*). 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 [CT96, GS96, Gue95, HMRT99].

We have recently been exploring the feasibility of this idea in the context of our Bast and DACE middleware frameworks. In this paper, we claim that to be effective, consensus should be *lazy*. Our specification of what we call *lazy consensus* can be viewed as a reshaping of consensus for practical usage. Roughly speaking, this new specification provides consensus with pragmatic *re-entrant* and *on-demand* aspects. Basically, by permitting different proposed values from the same process (*re-entrance*), we free the process from the obligation to log its proposed value. Furthermore, a recovered process might simply need to propose a value if it wishes to receive a decision (*on-demand*). By doing so, we also free the process from the obligation to log its decided value. As we pointed out, for all our protocols, the important optimizations we obtain in terms of forced logs are not achieved at the expense of stronger assumptions, or additional messages and communication steps, with respect to alternative protocols we are aware of [ACT98, HMR98, RR99, Ske91]. For specific configurations of the system, one could even use our consensus abstraction to optimize the communication pattern of both atomic broadcast and atomic commit. Indeed, consider the case where consensus is implemented by a subset of n processes that is different from the subset that needs to perform atomic broadcast (resp. atomic commit). In normal runs, only one member of the consensus service needs to propose a value and decide. By relaxing the requirement that all correct processes need to propose and decide, our specification enables us to save $2n - 2$ messages ($n - 1$ messages sent to all members of the consensus service and $n-1$ messages sent from the coordinator of the consensus service to all members of that service). Furthermore, typical optimizations to agreement protocols can also be applied to our protocols. For instance, we can, just like in [GS96], trade efficiency with reliability and obtain more efficient protocols. It is also important to notice that both our atomic broadcast and atomic commit protocols are quite simple. They are actually similar to the algorithms described in [CT96] and [GS96], respectively, which were designed with a crash-stop model in mind. The complexity raised by the crash-recovery model is factorized within the implementation of lazy consensus. Exposing that complexity leads to ad-hoc protocols that are far more complex, yet not more efficient (see for example the description of the non-blocking atomic commit protocol of [BHG87]).

It would be interesting to explore the generalization of our *on-demand* and *re-entrance* flavors other kinds of distributed abstractions. In particular, we have recently been exploring these ideas on various forms of weak agreement, along the lines of [MDB99].

References

[ACT97] M. K. Aguilera, W. Chen, and S. Toueg. Quiescent reliable communication and quiescent consensus in partitionable networks. TR 97-1632, Department of Computer Science, Cornell University, June 1997.

[ACT98] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC-12)*, Andros, Greece, September 1998.

[Anc97] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, May 1997.

[BHG87] P. A Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. In *ACM Transactions on Computer Systems*, number 9(3) in ACM Press Books, pages 273–314, August 1991.

[BvR96] K. Birman and R. van Renesse. Software reliability for networks. In *Scientific American*, number 274 (5) in Scientific American, May 1996.

[CF95] F. Cristian and C. Fetzer. On the possibility of consensus in asynchronous systems. In *IEEE Pacific Rim International Symposium on Fault-Tolerant Systems*, December 1995.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[DKM93] D. Dolev, S. Kramer, and D. Malkhi. Early delivery totally ordered broadcast in asynchronous environments. In *IEEE Symposium on Fault-Tolerant Computing*, pages 296–306, June 1993.

[EMS95] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *IEEE Conference on Distributed Computing Systems*, pages 296–306, May 1995.

[FG00] P. Felber and R. Guerraoui. Programming with object groups in corba. *IEEE Concurrency*, 8(1), Jan-Mar 2000.

[FLP85] M. J. Fischer, 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.

[GG98] B. Garbinato and R. Guerraoui. Flexible protocol composition in bast. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18)*, pages 22–27, Amsterdam, The Netherlands, May 1998.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[GS96] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building agreement protocols. In *IEEE Symposium on Fault-Tolerant Computing*, pages 168–177, June 1996.

[Gue95] R. Guerraoui. Revisiting the relationship between non blocking atomic commitment and consensus problems. In *Distributed Algorithms*, number 791 in Lecture Notes in Computer Science, pages 87–100. Springer-Verlag, September 1995.

[Her91] M. Herlihy. Wait-free synchronization. In *ACM Transactions on Programming Languages and Systems*, number 13(1) in ACM Press Books, January 1991.

[HMR98] M. Hurfin, A. Moustefaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *IEEE International Symposium on Reliable Distributed Systems*, October 1998.

[HMRT99] M. Hurfin, R. Macedo, M. Raynal, and F. Tronel. A general framework to solve agreement problems. In *IEEE International Symposium on Reliable Distributed Systems*, October 1999.

[Lam89] 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 TOCS vol.16 number 2.

[MDB99] A. Montresor, R. Davoli, and Ö. Babaoğlu. Middleware support for the development of distributed applications in partitionable systems. UBLCS 99-05, Department of Computer Science, University of Bologna University, April 1999.

[NMMS97] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the internet inter-orb protocol to provide corba with fault tolerance. In *Usenix Conference on Object Oriented Technology and Systems*, pages 81–90, 1997.

[OP98] M. Ogg and F. Previato. Experience with replicated distributed objects: The nile project. *Theory and Practice of Object Systems*, 4(2), 1998.

[RR99] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous systems where processes can crash and recover. Technical Report TR-99-7, FCUL, April 1999. http://www.di.fc.ul.pt/biblioteca/tech-reports/.

[Ske91] D. Skeen. Nonblocking commit protocols. In *ACM SIGMOD International Conference on Management of Data*, pages 133–142, 1991.

[WWWK94] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical report, Sun Microsystems Laboratories, Inc, November 1994.

A Proofs

A.1 Consensus

Our proofs are quite similar to the proofs of [ACT98]. Our definition of starting a round r is slightly different since we do not log r. Except for line numbering, there are no differences for *validity* and *agreement. Termination* is different; the proof of *lemma 58* from [ACT98] (*lemma 58bis* for us) is different since there is only one message sent instead of two in [ACT98]. Our proof does not use the fact that *outside task retransmit*, p cannot send messages of the form (*,DECIDE) since the message is not sent to all processes. Finally, *Lemmatas 61* and *62* are not necessary in our case and are thus ignored. Since the termination property is different from [ACT98], so is *lemma 63.* However, our proof is similar to that of *lemma 63.* Finally, in the terminology used in [ACT98], *good* processes are in our terminology *correct* while *bad* processes are *faulty.*

Lemma 58bis. Let p and q be any two correct processes. If (1) p S-SENDs m to q after p *stabilizes*, (2) m is the last message p S-SENDs to q, and (3) no processes have decided, then q receives m from p infinitely often.

Proof (Sketch). By (1),(2) and (3), p sends m to q infinitely often in task *retransmit* (line 11). By the *fair loss* property of the channels, q receives messages from p infinitely often. Note that m is the only message that p sends to q infinitely often since, in task *retransmit*, p eventually sends no message different from m to q. Therefore, by the *no creation* and *finite duplication* properties of the channels, q receives from p only finitely many messages different from m. Since q receives messages from p infinitely often, it follows that q receives m from p infinitely often.□

Lemma 64bis (Termination). Suppose there is a majority of correct processes. If a correct process p proposes, then unless it crashes, it eventually decides.

Proof (Sketch). In order to obtain a contradiction, suppose that correct process p never decides. By *lemma 56*, each correct process p can start only finitely many rounds. Since p proposes, p blocks in some round r_p. Let $r = \max\{r_p | p \text{ is correct}\}$ and let p be a correct process that blocks in round r.

- *Case 1: p is the coordinator of round r.*

Process p never decides, so in round r, either p waits forever at line 29 or at line 35.

- *Case 1.1: p waits forever at line 29*

We claim that for every correct process q, p eventually receives $(r, estimate_q, ts_q,\text{ESTIMATE})$ from q after p *stabilizes.* Then by the assumption of a majority of correct processes, p does not wait forever at line 29: a contradiction.

To show the claim, note that since p waits forever at line 29 of round r, we have $ts_p \neq r$. Thus, p never updates ts_p to r, and so p never updates $estimate_p$ in round r. By *lemma 52-1*, p never starts phase NEWESTIMATE. So p never S-SENDs NEWESTIMATE messages in round r.

- *Case 1.1.1: $q = p$*

Since $ts_p \neq r$, in round r, after p stabilizes and forks task *participant*, p *s-sends* $(r, estimate_p, ts_p,\text{ESTIMATE})$ to itself (line 43). Thus p receives this message after it stabilizes.

- *Case 1.1.2: $q \neq p$*

Before p waits forever at line 29, it S-SENDS (r,NEWROUND) to q (line 28) after p stabilizes and this is the last message p S-SENDs to q. By *lemma 59*, q eventually receives (r,NEWROUND) after q stabilizes. By *lemma 9*, q eventually starts a new round $r' \geq r$. By the definition of r, we have: $r' \leq r$. Thus $r' = r$ and so q starts round r. In round r, we have: $ts_q \neq r$ (otherwise q sets ts_p to r in line 46. which implies that that q received a NEWESTIMATE message from p contradiciting that p never S-SENDS NEWESTIMATE messages). Then q S-SENDs message ($r, estimate_q, ts_q$,NEWESTIMATE) to p (line 43). Process q waits forever in line 44 since p never S-SENDs a NEWESTIMATE message to q. Therefore ($r, estimate_q, ts_q$,ESTIMATE) is the last message q S-SENDs to p. By *lemma 59*, p eventually receives ($r, estimate_q, ts_q$,ESTIMATE) from q after p stabilizes. This conclude the proof of the claim.

- *Case 1.2: p waits forever at line 35*

We claim that for every correct process q, p eventually receives (r,ACK) from q after p stabilizes. Then by the assumption of a majority of correct processes, p does not wait forever at line 35: a contradiction. We now show the claim

- *Case 1.2.1:* $q = p$

Before p waits forever at line 35, it S-SENDs a NEWESTIMATE message to itself (and it does so after p stabilizes). Thus p receives this message from itself. So in task *participant*, p finishes phase ESTIMATE and S-SENDs (r,ACK) to itself. Therefore p receives this message from itself after it stabilizes.

- *Case 1.2.2:* $q \neq p$

Before p waits forever at line 35, it S-SENDs a NEWESTIMATE message to q and this the last message p S-SENDs to q. By *lemma 59*, q eventually receives this message from p after q stabilizes. By *lemma 58*, q eventually starts a round $r' \geq r$. By the definition of r, we have $r' \leq r$ and so q blocks in round r. In round r, after q stabilizes, q finishes phase ESTIMATE (since q receives a NEWESTIMATE message from p) and S-SENDs message (r,ACK) in phase ACK. This is the last message q S-SENDs to p, since q blocks in round r. By *lemma 59*, p eventually receives (r,ACK) from q after p stabilizes.

- *Case 2: q is not the coordinator of round r*

Let $c \neq p$ be the coordinator of round r. By *lemma 60*, c is a correct process and c receives messages of round r from p infinitely often. By *lemma 58*, c eventually starts a round $r' \geq r$. By the definition of r, we have: $r' \leq r$. Thus $r' = r$ and so c blocks in round r. In case 1, we showed taht the coordinator of round r does not block in round r: a contradiction. □

A.2 Atomic Broadcast

Theorem A.1. The algorithm of Figure 5 fulfills the validity, agreement, total order and termination properties of atomic broadcast.

Lemma 2. For any two correct processes p and q, and all $k \geq 1$: (1) If p executes propose(k,-), then q eventually executes propose(k,-). (2) If p *A-delivers* messages in $A\_deliver_p^k$, then q eventually *A-delivers* messages in $A\_deliver_q^k$, and $A\_deliver_p^k = A\_deliver_q^k$.

Proof (Sketch). We suppose that new messages keep on coming, so that *R_delivered* is never empty. The proof is by simultaneous induction on (1) and (2). For $k = 1$, we first show that if p executes propose(1,-), then q eventually executes propose(1,-). When p executes propose(1,-), $R\_delivered_p$ must contain some message m. By our supposition, since $A\_delivered_q$ was initially empty, we have $R\_delivered_q$ - $A\_delivered_q \neq 0$. Thus, q eventually executes line 9 and propose(1,-).

We now prove that if p *A-delivers* messages in $A\text{-}deliver^1_p$, then q eventually *A-delivers* messages in $A\_deliver^1_q$, and $A\_deliver^1_p = A\_deliver^1_q$. From the algorithm, if p *A-delivers* messages in $A\_deliver^1_p$, it previously executed propose(1,-). From (1), all correct processes eventually execute propose(1,-). By the termination property of lazy consensus, every process that does not crash eventually decide(1,-). Since $A\_delivered_p$ and $A\_delivered_q$ are initially empty, and $msgSet^1_p = msgSet^1_q$, we have: $A\_deliver^1_p = A\_deliver^1_q$. The induction is then trivial. □

Lemma 3 (Agreement and Total Order). The algorithm in Figure 5 satisfies the agreement and total order properties of Atomic Broadcast.

Proof. Immediate from Lemma 2 and the fact that processes *A-deliver* messages in each batch in the same deterministic order. □

Lemma 4 (Termination). If a process *A-broadcast*s a message m, then unless it crashes, it eventually *A-delivers* m.

Proof (Sketch). If a process p does not crash and has *A-broadcast* m, $R\_delivered_p$ contains m. $R\_delivered_p$ - $A\_delivered_p$ being not empty, p will propose m in line 13. By the termination property of lazy consensus, it will eventually receive a decide message. There are two cases: 1) $m \in msgSet$ and 2) $m \notin msgSet$. 1) is trivial, with 2) m stays in $R\_delivered_p$ but p will keep on proposing m in *msgSet*. Since p never crashes, it will never loose the content of $R\_delivered_p$ and eventually a decide message will contain m inside, thus *A-delivering* m. □

Proof of Theorem A.1. Validity is trivial. Agreement, termination and total order follow from lemmata 1,3 and 4. □

A.3 Atomic Commit

Theorem A.2. The algorithm of Figure 8 fulfills the validity, non-triviality, agreement and termination properties of atomic commmit.

Lemma 5 (Validity). A process decides *commit* only if all processes vote yes.

Proof (Sketch). Suppose by contradiction that p decides *commit* whereas some process has voted *no*. To decide *commit*, a process must either 1) propose *commit* or 2) receive the decision *commit*. For 1) p can propose only at line 23 and will only propose *commit* if all processes vote yes due to the double condition in guard line (line 22). For 2) to happen, *commit* must have been proposed, since the only place where *commit* can be proposed is line 23, it results that for a process to receive a decide(*commit*) all processes must have voted yes: a contradiction. □

Lemma 6 (Agreement). No two processes decide differently.

Proof (Sketch). Follows from the agreement property of lazy consensus. □

Lemma 7 (Non-Triviality). If all processes vote yes and no process is suspected to have crashed, then all correct processes eventually decide *commit*.

Proof (Sketch). Since all votes are yes, TM will propose and decide *commit*. By the properties of *R-Broadcast*, all processes will decide *commit*. □

Lemma 8 (Termination). If a process *votes*, then unless it crashes, it eventually decides.

Proof (Sketch). If a process does not crash, there are two cases, 1) p is TM or 2) p is a regular process.

- Case 1: $p = TM$

If TM does not crash, it will eventually propose since it waits only for votes from *unsuspected* processes. Since TM proposes, it will receive a decision, thus proving that p decides either *commit* or *abort*.

- Case 2: $p \neq TM$

there are again two cases: 1) p suspects TM or 2) p does not suspect TM.

- Case 2.1: p suspects TM

if p supsects TM, task *suspect* will stop task *A-Commit*, then propose *abort* and wait for the decision, thus proving that p decides.

- Case 2.2: p does not suspect TM

If TM is not suspected, task *suspect* will never stop task *commit*. Process p will wait not forever in line 29. TM will not crash and will therefore propose either *abort* or *commit*. It will then receive the decision and *R-Broadcast* it to all processes. Thus proving that p will also decide.

Proof of Theorem 2. Follows from lemmata 5, 6, 7 and 8. □

B Equivalence between consensus and lazy consensus

We show here that consensus and lazy consensus are equivalent in a crash-recovery model. First we describe an algorithm that transforms lazy consensus into consensus (Figure 11) and then we describe an algorithm that transforms consensus into lazy consensus (Figure 12). 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 lazy consensus and vice-versa.

To distinguish the primitives that define these problems, we denote by *propose* and *decide* those that define consensus, and by *l-propose* and *l-decide* those that define lazy consensus.

In the (uniform) consensus problem, every process is supposed to *propose* a value, and the processes need to satisfy the following properties:

Termination: Every correct process eventually decides.

Agreement: No two processes decide differently.

Validity: If a process decides v then some process has previously proposed v.

In the (uniform) lazy consensus problem, the processes need to satisfy the following properties:

Termination: If a process l-proposes a value, then unless it crashes, it eventually l-decides some value.

Agreement: No two processes l-decide differently.

Validity: If a process l-decides v then some process has previously l-proposed v.

The algorithm that transforms lazy consensus into consensus is given in Figure 11. This algorithm assumes the existence of a lazy consensus box. Every process stores the proposed value on stable storage, then l-proposes it. If the process l-decides a value, it simply decides it, i.e., returns that value.

```
1: procedure consensus($v_{p_i}$)
2:     store(propose($v_{p_i}$))
3:     l-propose ($v_{p_i}$)
4:     wait until $l - decide(decision)$
5:     return($decision$)

6: upon recovery do
7:     if propose($v_{p_i}$) had occured then
8:         retrieve(propose($v_{p_i}$))
9:         l-propose($v_{p_i}$)
```

Figure 11: Transforming lazy-consensus to consensus

Theorem B.1. The algorithm of Figure 11 fulfills the validity, agreement and termination properties of consensus.

Proof (Sketch). The validity and agreement properties of consensus follow from the validity and agreement properties of lazy-consensus. 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 are always-up. Hence, by the algorithm of Figure 11, there is a time after which every correct process eventually proposes some value and never crashes afterwards. By the termination property of lazy consensus, every correct process eventually l-decides. By the algorithm of Figure 11, every correct process eventually decides. □

The algorithm that transforms consensus into lazy consensus is given in Figure 12. This algorithm assumes the existence of a consensus box. Basically, every process that l-proposes a value sends a message to all, to make sure that all processes that do not crash propose some value. A process that decides some value, l-decides that value if it has l-proposed some value.

```
procedure s-send$_{p_i}$(m)                                   {to s-send m to q}
  if $p_j \neq p_i$ then
    xmitmsg[$p_j$] $\leftarrow m$; send $m$ to $p_j$
  else
    simulate receive $m$ from $p_i$
task retransmit
  while true do
    for all $p_j \in \Pi \setminus p_i$ do
      if xmitmsg[$p_j$] $\neq \perp$ then
        send xmitmsg[$p_j$] to $p_j$
procedure L-Consensus($v_{p_i}$)
  S-SEND propose($v_{p_i}$) to all processes
  l-propose($v_{p_i}$)
  wait until received[DECIDE(decision)]
  return(decision)
```

Figure 12: Transforming consensus to lazy-consensus

Theorem B.2. The algorithm of Figure 12 fulfills the l-validity, l-agreement and l-termination properties of consensus.

Proof (Sketch). The validity and agreement properties of lazy consensus follow from the validity and agreement properties of consensus. Consider now the termination property of lazy consensus. Let p_i be any process that l-proposes some value and does not crash. By the *fair loss* property of the channels, every correct process eventually receives message *propose*(v_{p_i}) from p_i. By the algorithm of Figure 12, every correct process proposes some value. By the termination property of consensus, every correct process eventually decides. By the algorithm of Figure 12, process p_i eventually l-decides. □