Unconscious Eventual Consistency with Gossips

Roberto Baldoni[1], Rachid Guerraoui[2,3], Ron R. Levy[2],
Vivien Quéma[1] and Sara Tucci Piergiovanni[1]

[1]DIS, Università di Roma "La Sapienza", 00198 Roma, Italy
[2]LPD, EPFL, CH 1015 Lausanne, Switzerland
[3]CSAIL, MIT, Cambridge, MA 02139, USA

Abstract. This paper presents a replication protocol that ensures eventual consistency in large-scale distributed systems subject to network partitions and asynchrony. A simulation study shows that the resulting protocol is scalable and achieves high throughput under load.
Our protocol does not rely on any form of consensus, which would lead to block the replicas in case of partitions and asynchrony. Our protocol instead ensures that (1) updates are continuously applied to the replicas and (2) no two updates are ever performed in a different order. Gaps might occur during periods of unreliable communication. They are filled whenever connectivity is provided, and consistency is then eventually ensured, but without any conscious commitment. That is, there is no point in the computation when replicas know that consistency is achieved. This unconsciousness is the key to tolerating perpetual asynchrony with no consensus support.

1 Introduction

A new class of so-called *interactive* distributed applications is emerging: distributed virtual environments, interactively steered scientific applications, collaborative design systems, etc [3]. These applications may need to run in a wide area asynchronous environment with widely distributed users and resources and no central authority. In such settings it is important for each user to have access to a local copy (replica) of every object of interest. This is key to allowing local progress without constantly relying on the network. The main technical challenge is then to maintain some form of consistency among all replicas of the same object [17].

Traditionally, many systems running on local area networks provide so-called single copy semantics that gives the user the illusion of accessing a single, highly available object. Typical solutions require users to access a quorum of replicas, to acquire exclusive locks on data they wish to update or to agree on a total order of updates to be applied at each replica. Maintaining single-copy semantics in a worldwide deployed system is practically very expensive and theoretically impossible [10]. It is thus necessary to use (weaker) consistency criteria. This is precisely what *eventual consistency* [19] provides. It guarantees that whatever the current state of the replica, if no new updates are issued and replicas can

communicate freely for a long enough period, the contents of all replicas eventually become identical. From an implementation point of view, the issues to solve in order to guarantee eventual consistency are [17]: (1) *update dissemination*: each update must eventually reach all replicas, and (2) *update ordering*: all updates must be *eventually* applied in the same order at each replica.

Some solutions (Bayou [19], OceanStore [15]) disseminate updates using epidemic (gossip) protocols. Update ordering [15, 19, 14, 20, 18] is achieved by having replicas deliver updates locally in any order (tentative order) and using *rollbacks* to eventually reach a total order. Total ordering is typically computed *a posteriori* using some form of consensus. This requires a "synchrony island" where agreement can be achieved to ensure that all replicas eventually agree on the exact update order. When that happens, each replica is *conscious* of the fact that total order has been reached.

This paper presents a replication protocol that achieves eventual consistency in large-scale distributed environments subject to network partitions and asynchrony. Update dissemination is performed using a classical gossip-based strategy [8]. Our replication protocol differs from others by the fact that it does not use any form of consensus, even only eventual. It defines an *a priori* total order that is never explicitly agreed upon among replicas. Updates are disseminated using gossips and subsequently delivered. In the case that some old update arrives after already having delivered subsequent messages, the replica has to roll back to the old state, apply the old update and re-deliver all subsequent messages. This means that, in theory, each replica should keep all delivered updates forever. However, in practice, it is possible to reach consistency with high probability without keeping all delivered updates.

A fundamental aspect of our protocol is that replicas are *unconscious* of when total order is reached, i.e. when they are in a consistent state. This unconsciousness is the key to reaching eventual consistency even if the network is permanently asynchronous. Our protocol has the following characteristics:

- *Non-blocking :* the protocol enables update delivery even during periods when the network is asynchronous or partitioned.
- *Self-stabilizing :* the protocol exploits periods of (even partial synchrony) and merging of partitions to reduce the number of rollbacks. (Note that the periods of synchrony are not relied on in order to reach consistency).
- *Scalable :* the protocol encompasses a self-sizing mechanism that guarantees high throughput when the number of broadcasters and/or the rate at which they broadcast updates increase.

Our simulations convey the fact that our protocol achieves reasonable latency during synchronous periods (due to a small number of rollbacks) and achieves high throughput under high load.

This paper is organized as follows. Section 2 presents the ramifications underlying unconscious eventual consistency. Section 3 describes our protocol. A performance evaluation is presented in Section 4. Finally, related work is presented in Section 5, before concluding the paper in Section 6.

2 Ensuring Unconscious Eventual Consistency

Roughly speaking, eventual consistency stipulates that all replicas eventually converge to the same state, i.e. deliver the same set of updates in the same total order. Eventual consistency can be achieved by having replicas (called processes in the rest of the paper) deliver updates in their order of arrival and then eventually re-order already delivered updates using a rollback mechanism. This section starts by discussing few points that must be taken into account while implementing eventual consistency. We then describe a naive implementation. Finally, we discuss the drawbacks of this naive implementation to introduce the improvements that are brought by the protocol presented in this paper.

2.1 Few Comments on Eventual Consistency Implementations

Update Ordering. As explained before, achieving eventual consistency requires every process to eventually deliver updates in the same order. Since updates can continuously be applied (i.e. processes can *re*-deliver updates until the total order is reached), it is only needed that each two updates be univocally associated to unique sequence numbers. On the other hand, it is not necessary that assigned sequence numbers be *consecutive* (i.e. *gaps* in the sequence are allowed). Nevertheless, for avoiding rollbacks, it is better that they be consecutive as this allows processes to know whether it is worth waiting for updates.

Update Dissemination. Eventual consistency requires that all updates eventually reach all processes. Reliable communication is therefore necessary. However, in a large scale environment, ensuring strong reliable communication can be very expensive. Consequently, most solutions [15, 19] rely on epidemic dissemination [13, 4, 7], even if they do not provide strong reliability. Therefore, just like [15, 19], our protocol only provides eventual consistency with high probability.

Unconscious Consistency. The total order used to achieve eventual consistency can be defined *a priori* (by associating to each update a pair composed of ID of the process that issued the update and a local sequence number). This allows achieving eventual consistency without relying on consensus. On the other hand, not relying on consensus implies that processes never know when a consistent state has been reached. As a consequence, we say that eventual consistency is implemented in an *unconscious* manner.

2.2 A Naive Protocol

Eventual consistency can be naively implemented as follows. Consider a finite and ordered set of processes $\{p_1, \ldots, p_n\}$. Each process acts as a *sequencer*; it keeps a local sequence number that is increased before broadcasting a new message (update). Along with the sequence number, each process tags the message m with its id. The resulting message (m, id, seq) is then disseminated to all processes. A total order is defined on these messages using the sequence number

and id. More precisely: for any pair of messages m and m', m precedes m' iff (i) $seq < seq'$ or (ii) $seq = seq'$ and $id < id'$.

Upon reception of a message, a process cannot possibly know if it will ever receive another message preceding it in the total order. Indeed, there may exist *gaps* in the sequence of broadcast messages. It therefore doesn't make sense for a process to wait for other messages. Consequently, processes deliver messages upon reception. If a message m_1 is received after a message m_2 preceding it in the total order, a rollback is performed on m_2. Subsequently, m_1 and m_2 are delivered in the correct (total) order.

2.3 Towards a Better Protocol

The drawback of the naive implementation is that there is no mechanism to reduce the number of rollbacks. In particular, with a large number of sequencers, the number of rollbacks in the system drastically increases. Consider that there are N sequencers in the system identified by $s_1 < \ldots < s_N$. Each sequencer sequences k messages. Moreover, consider that messages are broadcast using a reliable FIFO broadcast primitive. If $N = 1$, all messages are received in the correct order by all processes. Thus, no rollbacks are necessary. However, with a larger number of sequencers, the number of possible rollbacks increases. Consider the case $N = 2$ with s_1 and s_2 starting to broadcast at the *same time* and *same rate*. Moreover, consider that messages sent by s_2 are systematically received before messages sent by s_1. Messages arrive at each process in the following order: $(m_2, s_2, 1)$, $(m_1, s_1, 1)$, $(m_4, s_2, 2)$, $(m_3, s_1, 2)$, etc. Consequently, each process needs to rollback k messages (those sent by s_2). Extending the previous example to a system with $N = m$ sequencers, it is trivial to demonstrate that each process performs $(m - 1) \times k$ rollbacks.

The protocol described in the next section exploits periods of synchrony to reduce the possible number of sequencers (and hence reduce the number of rollbacks) and to assure that each sequencer (actually implemented by a set of processes) gives consecutive sequence numbers.

3 Protocol

This section starts by an overview of the protocol. We then describe its basic behaviour. Follow the presentation of a self-stabilization mechanism and the description of a self-sizing mechanism that improves the protocol's scalability.

3.1 Overview

For scalability and fault-tolerance reasons, the protocol we propose implements each sequencer as a pool of processes organized in a *coalition*. Each process wishing to disseminate an update has access to a primitive called ecBroadcast. This primitive first requests a sequence number from *the coalition the process relies on* and then uses gossiping to disseminate the update together with its sequence number.

Creating coalitions. If a process p_i that does not rely on a coalition wants to ecBroadcast a message, it first tries to discover an already existing coalition. If it does not find one, it creates a new coalition including itself and some other processes (to get the desired size of the coalition) in a new coalition.

Sequencing using coalitions. A coalition c_k is a set of processes (called *members*) acting as a common sequencer. Within a coalition, processes are sorted using their identifiers. We note $c_k[x]$ the xth process in c_k (x is called *rank* of process $c_k[x]$) and we note $card(c_k)$ the cardinality of coalition c_k. Processes belonging to a coalition issue sequence numbers as follows: let c_k be a coalition and let p_j be a process belonging to c_k, $p_j = c_k[x]$. Process p_j assigns monotonically increasing sequence numbers belonging to the sequence $SN^{c_k[x]} = (sn_n)_{n \in \mathbb{N}}$ with $sn_n = n \times card(c_k) + x$. Along with this sequence number, messages are tagged with the id of the process that issued the sequence number.

Note that the above-described mechanism ensures that a coalition issues *distinct*, totally-ordered sequence numbers. Moreover, the protocol is such that each process requests sequence numbers to coalition members in a round-robin way. This allows (1) balancing the load over all coalition members and (2) increasing the probability that successively issued sequence numbers be *consecutive*.

Dissemination. We rely on a gossip-based protocol for message dissemination [8]. It has been shown that these protocols are able to ensure high delivery ratios. Moreover, for improving reliability during periods when the network is highly asynchronous or partitioned, the protocol uses a *pull* mechanism similar to the one presented in [19].

Delivering messages. Processes try to deliver messages in sequence. This is done by waiting until the preceding messages have been delivered before delivering the current one. However, a process cannot possibly know about all preceding messages for three reasons: (1) there might be other coalitions issuing sequence numbers, (2) the sequence numbers issued by the coalition the process relies on are not necessarily consecutive, and (3) the gossiping mechanism used for dissemination is not reliable. Therefore, a process only waits for a given period of time before delivering received messages. Consequently, a message can be received after consecutive messages have already been delivered. In this case a rollback mechanism is used to undeliver messages and re-deliver them in the correct order. Our experiments show that in the case when only one coalition is present in the system, the number of rollbacks is close to zero.

Self-stabilization. As explained above, it is desirable to have a single coalition in the entire system. The protocol encompasses a self-stabilization mechanism that aims at leading to a system with only one coalition. Members of different coalitions get to know each other when they receive messages sequenced by a different coalition. If a member p_i of a coalition A receives a message coming

from another coalition B, then it builds a new coalition C including all members of A and B. As explained below, the size of the resulting coalition is readjusted after the merger. Note that this sizing mechanism tries to select the most *stable* processes, i.e. the processes that have been in the system for the longest time.

Each time a coalition member switches to another coalition, it starts issuing new sequence numbers as explained above. Therefore a process could reissue the same sequence number twice. This problem is solved by adding an epoch number to each sequenced message. When a process joins a coalition, it associates an epoch number to this new coalition. This epoch number must be greater than the epoch number of the last coalition the process was a member of. Epoch numbers do not change the way processes deliver messages. We just need to change the way the total order on messages is defined such that the epoch number takes precedence over the sequence number and finally the process id.

Self-sizing coalitions. Scalability of the sequencing service is obtained by dynamically adjusting coalition size according to the load on coalition members. This load depends on the number of broadcasters and the rate at which they broadcast. These two parameters are often impossible to determine a priori in the target environments. The self-sizing mechanism described in Section 3.4 dynamically modifies the size of coalitions, based on the average number of sequence number requests that coalition members receive during a period of time.

3.2 Main Protocol

Data structures. Each process p executing the algorithm contains the following set of data structures. *coalition* represents the coalition process p relies on. It is a list of processes. $optimalSize$ is the size that the *coalition* must have. *epoch* represents the epoch process p is in. $nextSN$ is the next sequence number from the coalition that p relies on and expects to deliver next. *pending* is the list of messages that process p received but did not yet deliver. Each entry in the *pending* list contains $[m, sn, ts]$, where m is the message to be delivered, sn is its sequence number (integrating the process id, epoch number and sequence number attributed by the sequencing service), and ts the time at which message m was received. The $deliveryTimeout$ parameter indicates the time process p should wait before delivering the first message in *pending*. All messages that have been delivered so far are stored in the *delivered* list. Finally, $nbOfRetries$ refers to the number of attempts to retrieve a coalition process p must do before creating its own coalition.

Note that for the sake of clarity, some functions (resp. messages) that are described below take a parameter, named $info$, that is a data structure carrying various data on the process that called the function (resp. sent the message). For instance $info.coalition$ contains the coalition the process relies on; $info.epoch$ carries its epoch; etc.

The isNext() function. To ease the reading of the algorithm, we have isolated the isNext() function (Figure 1), whose role is to indicate if a message must

be delivered (returns true) or if it must stay in the *pending* list. This function enforces the following policy: the protocol can only wait for messages that are sequenced by the coalition the process relies on and at the same epoch as the one the process is currently in. All other messages are delivered as soon as they are received.

```
1: function isNext(sn, ts)
2:    if (sn.pid ∈ coalition) ∧ (sn.epoch = epoch) then
3:       if (sn.number = nextSN) ∨ (ts + deliveryTimeout < getTime()) then
4:          nextSN := sn.number + 1
5:          return true
6:       else
7:          return false
8:    return true
```

Fig. 1. The isNext() function.

Algorithm executed by any process. Figure 2 depicts the algorithm executed by any process p_i. The coalitionUpdate() function aims at updating the knowledge p_i has about existing coalitions. It is called each time a new message is received. It simply changes p_i's coalition if p_i's epoch is lower than the epoch of the coalition given in parameter.

```
For each process p_i

 1: procedure ecBroadcast(m)
 2:    ⟨sn⟩ := getSN()
 3:    gossip ⟨m, sn, info⟩
 4:    pending.add([m, sn, getTime()])

 5: function getSN()
 6:    repeat nbOfRetries times
 7:       ⟨info⟩ := getCoalition()
 8:       if info ≠ ∅ then
 9:          coalitionUpdate(info)
10:          return snRequest()
11:    info.coalition = {p_i}
12:    info.epoch = epoch + 1
13:    coalitionUpdate(info)
14:    return snRequest()

15: upon gossip⟨m, sn, info⟩ from p_j do
16:    coalitionUpdate(info)
17:    pending.add([m, sn, getTime()])
```

```
For each process p_i

18: upon pending.first = [m, sn, ts]
          with isNext(sn, ts) do
19:    rolledback = ∅
20:    while m ≺ delivered.last do
21:       rollback(delivered.last)
22:       rolledback.add(delivered.removeLast())
23:    ecDeliver (m)
24:    delivered.add(m)
25:    while rolledback ≠ ∅ do
26:       ecDeliver (rolledback.removeFirst())
27:    pending.remove([m, sn, ts])

28: procedure coalitionUpdate(info)
29:    if info.epoch > epoch then
30:       coalition := info.coalition
31:       epoch := info.epoch
32:       nextSN := 0
```

Fig. 2. Algorithm executed by any process p_i.

Process p_i can use the ecBroadcast() function to initiate the broadcast of a message m. This function first gets a sequence number using the getSN() function; it then gossips the message together with its sequence number and

information about p_i (coalition and epoch); finally, it adds message m to the *pending* list. The getSN() function first tries to retrieve a coalition using the getCoalition()[1] function. Then, it uses the snRequest()[2]. function to get a sequence number from the coalition returned by the getCoalition() function. Note that each time the snRequest function is invoked, it sends the request to a different member in order to balance the load over all coalition members and to increase the probability to successively issue consecutive sequence numbers. After $nbOfRetries$ unsuccessful tries, the getSN() function creates a coalition.

When process p_i receives a gossip message m, it first updates its coalition if necessary; it then adds m to the *pending* list. Messages stored in the *pending* list are delivered as soon as they are first in the list and that the isNext() function returns true. Note that the delivery of a message may require rolling back and re-delivering previously delivered messages (Lines 19-22 and 25-26).

3.3 Self-stabilization

The mechanism described in this section aims at leading to a system with only one coalition. We start by describing a protocol executed by coalition members to merge coalitions. Then, we present an age-based mechanism that allows selecting stable processes, i.e. processes that remained in the system for the longest time. Finally, we show how faults impacting coalition members are handled.

Merging coalitions Each coalition member p_i executes an algorithm in charge of merging coalitions. This algorithm differs from the one executed by standard processes by the coalitionUpdate() function (Figure 3). Its behavior is the following: when the coalition given in parameter is the same as p_i's coalition, the function simply updates p_i's epoch if it is lower than the one passed as a parameter. When coalitions differ, the function merges the two coalitions and uses the size() function to try to reach the coalition's optimal size. This function either truncates the coalition using the truncate() function, or adds processes returned by the getProcess() function. Next paragraph explains how processes are selected by these two functions.

Aging mechanism. To improve the stability convergence time, the protocol encompasses an *aging* mechanism[3] that aims at selecting the most *stable* members. The aging mechanism shares similarities with the mechanism used to improve the reliability of epidemic broadcast algorithms [8]. The basic idea underlying this mechanism is that each process has an age that reflects the number of messages the process delivered (the age is incremented every N deliveries). Each

[1] For space reasons, the getCoalition() function is not described. This function either returns the coalition p_i relies on (if such a coalition exists), or broadcasts a "coalition request" message to discover a coalition.

[2] For space reasons, the snRequest() function is not described. This function simply requests a sequence number from one member of the coalition p_i relies on.

[3] For space reasons, we do not provide the pseudo-code of this mechanism.

For each coalition member p_i

```
1: procedure coalitionUpdate(info)
2:   if info.coalition = coalition then
3:     if info.epoch > epoch then
4:       epoch := info.epoch
5:       nextSN := 0
6:   else
7:     merge(coalition, info.coalition)
8:     size(coalition)
9:     epoch := max(epoch, info.epoch) + 1
10:    nextSN := 0
```

For each coalition member p_i

```
11: procedure merge(c1, c2)
12:   c1 := c1 ∪ c2

13: procedure size(c)
14:   if card(c) > optimalSize then
15:     truncate(c)
16:   else
17:     while (card(c) < optimalSize) ∧
          hasMoreProcesses()
18:         c := c ∪ getProcess()
```

Fig. 3. Algorithm executed by any coalition member p_i.

process stores the age of coalition members and propagates them with each message (in the *coalition* list). Then, the truncate() function selects the members with highest age. Eventually, stable processes will have a higher age than all other processes, which guarantees that all coalition members will be stable.

Note that there is no guarantee that two executions of the truncate() function by two different coalition members will produce the same result. Indeed, this depends on the knowledge that these two members have about the ages of all coalition members. Nevertheless, this is not an issue because the probability of having different knowledge can be decreased by increasing N.

Moreover, to further increase the speed at which stability is reached, the getProcess() function returns "old" processes. This is achieved by having each coalition member maintain a (short) list of the oldest processes it knows.

Handling faults in coalitions. As described, the protocol does not handle faulty coalition members. This does not affect the correctness of the protocol, but it alters its stability convergence time. Faulty members are handled using a *heartbeat* protocol among coalition members (Figure 4). Each member periodically (δ) sends a $PING$ message to other members in the coalition. Members maintain two data structures: *alive* is the list of processes from which a $PING$ message has been received. This list is reset periodically. *suspected* is the list of processes that the member suspects. This list is built by adding members of the coalition that are not in *alive* after $(2 * \delta)$ ms (Line 7), and by adding members suspected by other members (Line 12). Processes that are in the *suspected* list of a process p_i will no longer be added by p_i in a coalition (Line 17).

The above-described behavior requires some additional comments: the *heartbeat* protocol does not prevent *false suspicions*. On the contrary, once a member is suspected by some process p_i, it will eventually be suspected by all other coalition members. Nevertheless, if *suspected* lists were not propagated, coalitions would oscillate as long as one process falsely suspects another member. Moreover, propagating *suspected* lists is not a real issue since (1) timeouts can be set sufficiently large to prevent most cases of false suspicions and (2) it is possible

to remove processes from the *suspected* lists after some (long enough) period of time, in order to allow falsely suspected processes to re-integrate coalitions.

```
For each coalition member p_i

1: suspected := ∅
2: alive := ∅

3: task heartBeat every δ ms
4:    send⟨PING, info⟩ to all p_j ∈ coalition

5: upon receive⟨PING, info⟩ from p_j do
6:    alive := alive ∪ {p_j}
7:    suspected.add(info.suspected)
8:    coalitionUpdate(info)

For each coalition member p_i

9: task coalitionMaintenance every (2 * δ) ms
10:    info.epoch = epoch + 1
11:    if alive ≠ coalition then
12:       suspected.add(coalition\alive)
13:       info.coalition = alive
14:       coalitionUpdate(info)
15:    alive := ∅

16: procedure merge(c1, c2)
17:    c1 := (c1 ∪ c2)\suspected
```

Fig. 4. Extension for handling faults within a coalition.

3.4 Self-sizing coalitions

This section describes a mechanism in charge of improving the protocol's scalability. In our context, ensuring scalability consists in being able to handle a large number of nodes and to guarantee high throughput in message deliveries under high load. The protocol described so far already deals with scalability issues by (1) using a gossip protocol to disseminate messages, (2) distributing the sequencer role among several processes (coalition), and (3) balancing the load among coalition members by requesting sequence numbers in a round-robin fashion. Nevertheless, one limitation of the protocol is that it assumes a priori knowledge of the *optimal* coalition size.

We have extended the protocol with a *self-sizing* mechanism[4] that aims at dynamically computing the optimal coalition size. This mechanism is based on the fact that during a long enough period of time, all coalition members experience the same load (due to the round-robin load balancing mechanism). Therefore, computing the optimal size can be done by a specific member (i.e. the member that has rank 0, which we will call the "smallest member"), by simply looking at the load it experienced during the last *sizing period*. If the node is overloaded, it adds processes to the coalition; otherwise, it removes processes. This is the responsibility of the application deployer to decide the maximal load (in terms of request/seconds) a node in the system can support.

When two coalitions merge, the optimal size is set to the sum of the optimal sizes of both coalitions. This is the only case when the optimal size can be changed by a member other than the smallest one. Note that it is necessary to determine if the optimal size is the one set by the smallest member or by the process that executed the merger. This decision can easily be done by propagat-

[4] For space reasons, the pseudo-code of this extension is not shown.

ing a *sizing number* together with the optimal size sent in each message. This sizing number allows knowing if a sizing decision precedes or not another one.

4 Performance

In this section, we present the performance results obtained by simulating our algorithm. We start by describing the simulation settings and then give the actual performance measurements. The goal of the simulations is to show that the protocol is (1) self-stabilizing, (2) non-blocking, and (3) scalable.

4.1 Simulation Environment

We simulated our algorithm using the Peersim simulator [1]. Peersim allows cycle-based simulations of distributed algorithms in large-scale environments. Processes are connected using a random graph topology: every process knows a fixed number of random processes. Moreover, processes disseminate messages using an LPBCast-like broadcast protocol [8]. Note that we extended the simulator in order to be able to simulate asynchrony: we can vary the time (i.e. number of cycles) it takes for a message to be transferred from one process to another. In our experiments, this time is bounded by *maxLatency*, and every message transfer takes a random number of cycles ranging from 1 to *maxLatency*.

Finally, we model churn (i.e. continuous joining and leaving of processes) by periodically replacing a percentage of processes. All experiments are run with 1000 processes, with a PING period (δ) of 20 cycles and a sizing period of 40 cycles. All the experiments start with a warm-up phase (first 100 cycles) in which processes progressively join.

4.2 Self-stabilization

The first experiment illustrates the fact that the protocol is able to select stable processes. It consists in simulating 1000 processes that randomly broadcast messages. The self-sizing mechanism was disabled and the optimal coalition size was set to 8. The goal of the experiment is to show how the average number of stable members in each coalition evolve. For the sake of clarity, the average was only computed on coalitions that stayed in the system for longer than 20 cycles.

Figure 5 depicts the average number of stable processes in each coalition as a function of time (i.e. cycle number). We varied both the latency (through the *maxLatency* parameter) and the churn rate. The *maxLantecy* parameter ranges from 1 to 15; the churn rate ranges from 4% to 8% every 15 cycles. We observe that without any aging mechanism, the protocol does not reach stability (last plot). On the contrary, the aging mechanism ensures that stability is reached (first four plots), i.e. that eventually there will be 8 stable processes in the coalition. Nevertheless, the speed at which stability is reached depends on the level of asynchrony and churn.

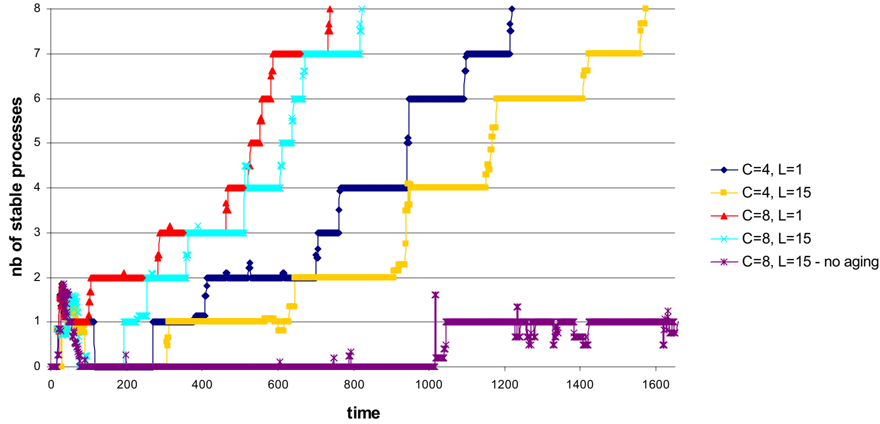

Fig. 5. Stable processes selection.

- The stability time increases with asynchrony for two reasons: (1) more time is necessary for coalitions to meet, and (2) asynchrony alters the knowledge that processes have about the age of other processes. Therefore, the protocol has a higher probability of selecting processes that are not stable.
- Increasing churn decreases the time it takes to reach stability. This result might seem surprising, but it can easily be explained by the fact that: (1) unstable members in the coalition have higher probability to fail (and thus to be replaced), and (2) stable processes are proportionally older (and thus have higher probability to be selected).

4.3 Non-Blocking Behavior

The second experiment illustrates the fact that our protocol is non-blocking. In particular, we show that it still provides service during periods when the network is partitioned. The experiment consists in simulating 1000 processes that randomly broadcast messages. The $maxLatency$ parameter is set to 10. Moreover, there is no churn. In order to simulate 3 network partitions, we group processes into 3 groups. The interconnection graph is built in such a way that each process has an equal number of (randomly chosen) neighbors in each group. A network partition is simulated by disconnecting the groups.

Figure 6 plots the average latency of a message broadcast as a function of the time at which the broadcast was initiated. The experiment starts with three network partitions that merge at cycle 300. As explained in Section 3.1, messages that are not delivered by the gossip primitive are retrieved using a *pull* mechanism. In the depicted experiment, this is the case of most messages sent between cycles 0 and 300. Indeed, our protocol keeps providing service, but the gossip primitive only delivers messages to processes belonging to the same partition as the one the message's broadcaster is in. Other processes wait until the partitions have merged to retrieve these messages using the pull mechanism.

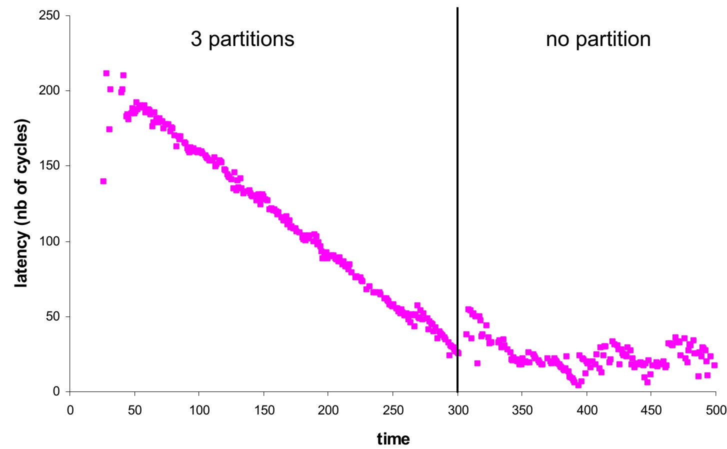

Fig. 6. Average message latency.

Messages broadcast after cycle 300 have an average latency ranging from 5 to 40 cycles. This is reasonable considering that the maximum latency of a point-to-point communication is equal to 10 cycles.

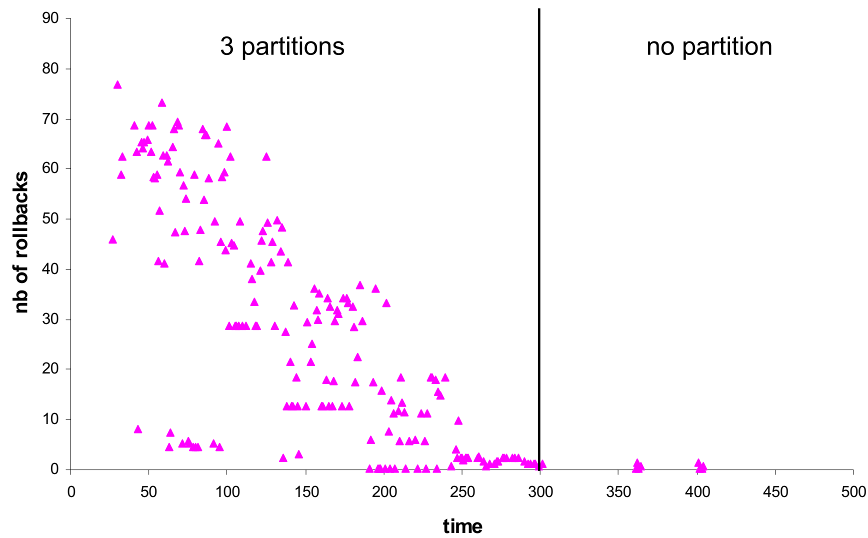

Fig. 7. Average number of rolled-back messages.

Figure 7 plots the average number of rollbacks that were done before delivering a message as a function of the time at which the broadcast was initiated. The experiment is the same as the previously described one. We observe that

messages broadcast between cycles 0 and 300 require rollbacks before being delivered. This can be explained by the fact that these messages were previously delivered in the partition of their respective broadcasters. After the network merger, these messages are retrieved using the pull mechanism. Their delivery requires rolling-back part of messages that were delivered during the network partition. We also observe that messages sent after cycle 300 do (almost) not require any rollback before being delivered. This shows that our protocol behaves like a traditional total ordering protocol when the network is not partitioned. As a consequence, it is possible in such periods to truncate the memory, while still ensuring eventual consistency with a very high probability.

4.4 Scalability

The last experiment we present demonstrates that the protocol is scalable. In particular, we show that the protocol ensures (almost) constant throughput even during periods when the number of initiated broadcasts drastically increases.

The experiment consists in simulating 1000 processes that have a probability to broadcast messages that varies over time. In this experiment, the $maxLatency$ parameter is set to 10 and there is no churn. Moreover, the warm-up phase is not represented for the sake of clarity. Figure 8 plots both the average number of sequence number (SN) requests received by each coalition member at the start of each round (first Y axis) and the average number of broadcasts initiated at the start of each round (second Y axis). Each "coalition X" plot depicts the life cycle of a coalition (i.e. the cycle at which it is created/destroyed) and the average number of SN requests received by each member.

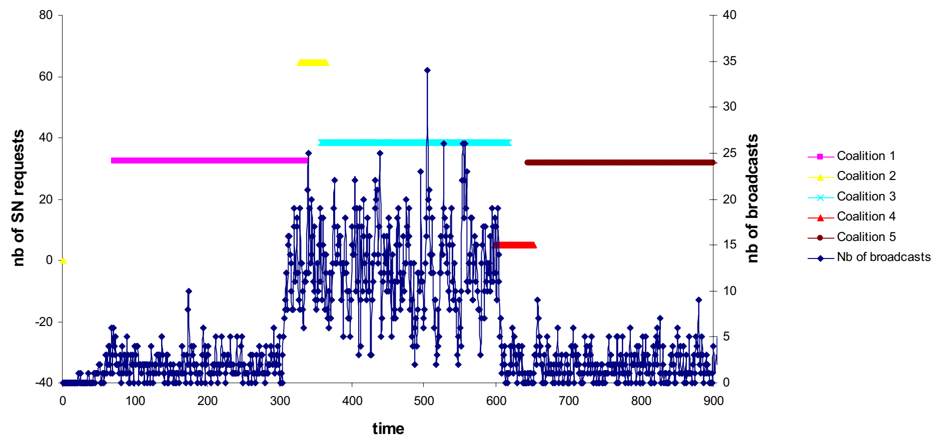

Fig. 8. Self-sizing mechanism.

The self-sizing mechanism was parameterized to maintain the average number of SN requests by member between 30 and 40. From cycle 0 to cycle 300,

processes have a low probability to initiate a new broadcast. During this period, messages are sequenced by coalition 1, which contains 3 members that handle (on average) 32,5 SN requests per cycle. Then, the broadcast rate significantly increases between cycles 300 and 600. Coalition 1 is first replaced by coalition 2 (6 members and 64,7 SN requests per cycle). Thus coalition 2 does not yet have enough members to handle the load. Consequently, coalition 2 is replaced by coalition 3 (12 members and 38,5 SN requests per cycle) after a short period of time. At time 600, the broadcast rate suddenly decreases. Coalition 3 is first replaced by coalition 4 (7 members and 15 SN requests per cycle), and then by coalition 5 (4 members and 31,3 SN requests per cycle). This experiment shows that the self-sizing ensures that coalitions can sustain a constant throughput, regardless of the broadcast rate.

5 Related Work

Update ordering for eventual consistency can be ensured by using total order protocols like the ones described in [6]. However, only optimistic total order protocols can efficiently support eventual consistency in a large scale setting [20, 18]. Other approaches to total ordering are too strong and would decrease responsiveness.

A interesting work is the one presented in [9] that presents a formalization of a related problem (eventual serializability) and an algorithm that solves it. Nevertheless, targeted environments are much smaller scale than the one we target and it is assumed that each replica is able to know if an update is stable (i.e. has been applied to every replica). Thus, the algorithm would not work correctly in highly asynchronous systems. Another work related to our work is the one done by Golding who proposes protocols for weak consistency group communications [11]. Proposed protocols assume a knowledge of the number of replicas in the system. Thus, they are not usable in the environments we target.

Moreover, several optimistic total order protocols have been proposed. They distinguish between tentative delivery and committed delivery of messages. This approach has been proposed by Kemme et al. in [14] to improve the responsiveness of the system in a LAN. The optimistic approach in this case is based on the spontaneous total ordering in LANs. The protocol proposed by Vincente and Rodrigues in [20, 18] guarantees that the tentative order is equal to the committed one during synchrony periods of the network. During periods of asynchrony rollbacks might occur. Finally, the protocol proposed by Sousa et al. in [18] does its best to guarantee that the tentative order is equal to the committed by artificially delaying messages received at a process before delivery through a mechanism called delay compensation. This delay based approach aims at creating the right conditions for spontaneous total ordering in WANs. All these protocols deterministically guarantee eventual consistency by relying on strong reliable update dissemination. As a consequence, they do not scale and cannot be employed in weakly connected environments. This is contrary to our protocol that uses epidemic dissemination.

There exist other examples of protocols relying on epidemic dissemination [19, 15, 16, 2]. For instance, Bayou [19] is a storage system designed for a weakly connected computing environment. In Bayou, one server, designated as the primary, takes responsibility for totally ordering updates and thus for deciding the committed order. Each secondary replica executes updates in a tentative order while the committed order is being decided. Update propagation follows an anti-entropy [7] mechanism: pairs of replicas periodically exchange information to update their states. This pair-wise communication copes with arbitrary network connectivity and after an arbitrary number of communication exchanges, replicas converge to an identical state.

Oceanstore [15] targets extremely wide distributed environments with huge numbers of users. Consistency is reached using a two-tier architecture: a specific small set of untrusted servers, called the inner ring of the object, store the primary object replicas (primary tier). Other replicas, called secondaries, are deployed on a large number of nodes, mostly for caching reasons (secondary tier). The inner-ring totally orders updates coming from any node hosting a replica using a Byzantine agreement protocol [5]. Contrarily to our protocol, in Oceanstore and Bayou, consistency is achieved in a *conscious* manner. Note that a similar notion of unconsciousness has been introduced in the context of self-stabilizing communication protocols [12].

6 Concluding Remarks

This paper combines various self-stabilization techniques within a replication protocol that ensures *unconscious* eventual consistency. The protocol is stable, non-blocking, and scalable. Our simulations convey the reasonable latency of the protocol during synchronous periods, and its high throughput under load.

In contrast to a conscious notion of eventual consistency, where the replicas would know when they reached a stable consistent state, the guarantee we provide can be implemented in permanently asynchronous environments, while still supporting important classes of distributed applications such as interactive applications based on continuous shared data.

References

1. PeerSim: A Peer-to-Peer Simulator, 2006. `http://peersim.sourceforge.net/`.
2. K. Aberer, M. Punceva, M. Hauswirth, and R. Schmidt. Improving data access in p2p systems. *IEEE Internet Computing*, 6(1):58–67, 2002.
3. S. Bhola and M. Ahamad. 1/k phase stamping for continuous shared data. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 181–190, 2000.
4. K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999.
5. M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

6. X. Defago, A. Schiper, and P. Urban. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.
7. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 1–12, 1987.
8. P. Eugster, R. Guerraoui, S. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight Probabilistic Broadcast. *ACM Transanctions on Computer Systems*, 21(4):341–374, 2003.
9. A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1):113–156, 1999.
10. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.
11. R. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4):379–405, 1992.
12. M. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Trans. Comput.*, 40(4):448–458, 1991.
13. I. Gupta, K. Birman, and R. van Renesse. Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. *Journal of Quality and Reliability Engineering International*, 2002.
14. B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proceedings of International Conference on Distributed Computing Systems*, 1999.
15. J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ASPLOS*, November 2000.
16. Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming Aggressive Replication in the Pangaea Wide-area File System. *ACM SIGOPS Operating Systems Review*, 36, 2002.
17. Y. Saito and M. Shapiro. Optimistic Replication. *ACM Computing Survey*, 37(1):42–81, 2005.
18. A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic Total Order in Wide Area Networks. In *Symposium on Reliable Distributed Systems*, pages 190–199, October 2002.
19. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Symposium on Operating Systems Principles*, pages 172–182. ACM Press, 1995.
20. P. Vicente and L. Rodrigues. An Indulgent Uniform Total Order Algorithm with Optimistic Delivery. In *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, pages 92–101, Osaka, Japan, 2002.