Maintaining Database Consistency in Peer to Peer Networks

Baruch Awerbuch and Ciprian Tutu
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218, USA
{baruch, ciprian}@cnds.jhu.edu

Technical Report
CNDS-2002-2
http://www.cnds.jhu.edu

February 6th, 2002

Abstract

We present an algorithm for persistent consistent distributed commit (distributed database commit) in a dynamic, asynchronous, peer to peer network. The algorithm has constant overhead in time and space and almost constant communication complexity, allowing it to scale to very large size networks. Previous solutions required linear overhead in communication and space, making them unscalable.

We introduce a modular solution based on several well defined blocks with clear formal specifications. These blocks can be implemented in a variety of ways and we give examples of possible implementations. Most of the existing solutions require acknowledgments from every participant for each action. Our algorithm is highly efficient by aggregating these acknowledgments. Also, in contrast with existing solutions, our algorithm does not require any membership knowledge. Components are detected based on local information and the information is disseminated on an overlay spanning tree.

The algorithm may prove to be more suited for practical implementation than the existing ones, because of its simplicity.

1 Introduction

We present an algorithm for persistent consistent distributed commit (distributed database commit) in a dynamic, asynchronous, peer to peer network. The problem has been recognized as a very difficult, yet very important problem. Replicated databases are becoming an imperative necessity and the existing solutions trade strict semantics for efficiency or vice versa. We introduce a modular solution based on several well defined blocks with clear formal specifications.

We base our solution on the construction of a stable overlay spanning tree over the set of active, connected nodes. Using the spanning tree as the underlying data structure guarantees an efficient communication pattern and facilitates the deployment of simple algorithms with clear correctness proofs. This construction is equivalent in requirements to the construction employed by Lamport's Paxos algorithm [Lam98] which is based on the election of a leader or with the constructions employed by solutions based on group communication.

Most of the existing solutions require acknowledgments from every participant for each action. Our algorithm aggregates these acknowledgments during normal operation, when the network is stable. Intuitively, we run a virtual clock on a spanning tree by means of *pulse* messages and converging acknowledgments. Since the actions are disseminated on the same spanning tree used by the pulse messages and because of the FIFO guarantees on the links, a pulse message p will implicitly acknowledge all messages sent during the previous pulse *p-1*[1]. The pulse mechanism as a synchronizer is derived from [Awe85].

The rest of the paper is organized as follows: the following subsections describe the model and relate to existing work. Section 2 evaluates the complexity of the algorithm and compares it to existing solutions. Section 3 introduces the main component of the algorithm and proves its correctness. Section 4 presents the complete algorithm that supports dynamic networks. Section 5 discusses the generality of the algorithm and concludes the paper.

1.1 Model

We consider a dynamic asynchronous network susceptible to network partitions/re-merges and server crashes/recoveries. The network configuration is given by a graph G(V(t), E(t)) where V represents the set of servers in the system and E the set of overlay connections between servers. Communication on the edges is assumed reliable and FIFO. We assume the existence of a network monitor module that detects failures of the physical network that disrupt the overlay topology.

Each node holds a local copy of the distributed database and may issue read/write operations on behalf of the clients that connect to them. The distributed database must obey consistency guarantees equivalent to one-copy-serializability. All the operations are assumed to have deterministic outcome.

The algorithm will satisfy standard correctness and liveness properties. In order to guarantee correctness, the algorithm establishes a global total order of actions that is consistent with FIFO specifications. If all the servers commit the actions in this agreed order, the database consistency is guaranteed. The correctness specification below is similar to the one used in [AT01].

Property 1 (Global Total Order)
If both servers s and r committed their ith action, then these actions are identical.

Property 2 (Global FIFO Order)
If server r committed an action a generated by server s, then r already committed every action that s generated prior to a.

Property 3 (Liveness)
If server s commits action a and there exists a set of servers containing s and r, and a time from which on that set does not face communication or process failures, then server r eventually orders action a.

1.2 Related Work

Two-phase commit [GR93, LL93] algorithms are the most common approach to providing a consistent view in a distributed database system over an unreliable network. However these protocols impose a substantial communication cost on each transaction and may require connectivity of all replicas

[1]Depending on the way the actions are sent, pulse p may actually acknowledge actions belonging to pulse *p-2* or *p-3*. See details in Section 5.

Algorithm\Measure	Space	#Mesgs	disk writes	time	space	#mesgs	time
2PC[GR93]	O(1)	3	n+1	2	0	0	0
[Ami95, AT01]	0	$1+2/\Delta$	1/n	O(1)	$\Omega(n)$	$\Omega(n)$	$O(1)$
This paper	0	$1+2/\Delta$	1/n	2	$O(1)$	$O(\log n)$	$O(1)$
Overhead	per transaction				per topological change		

Table 1: Complexity Comparison

in order to recover from some simple fault scenarios. Three phase commit protocols [Ske82, KD95] overcome some of the availability problems paying the price of an additional communication round.

Lamport introduced a now famous algorithm, known as the Paxos part-time parliament [Lam98]. By using the analogy with a parliamentary system where legislators have to agree on the decrees that are passed, Lamport presents a solution which is basically a three phase commit variant with better survivability properties. This solution is later analyzed by Lampson [Lam01] which puts it in a formal framework. The *voting* and updating of permanent records is performed only in the *senate,* (the primary component).

Keidar [Kei94] introduces a protocol that uses group communication to reduce the communication price by using group multicast primitives instead of point to point communication. Amir and Tutu [AT01, Ami95] also use group communication in order to improve communication performance. Their algorithm aggregates the acknowledgments required for each action thus improving even further the efficiency of the solution. The resulting protocol is highly efficient during normal operation and pays a higher price only when network faults occur. However, since it relies on the membership services provided by the group communication, the protocol cannot scale with the number of participants. Fekete, Lynch and Shvartsman [FLS01] develop a similar algorithm based on a different specification of a partitionable group communication service. However, the complexity of the group communication specifications, which hide some of the solution costs behind various abstractions, makes these algorithms more difficult to analyze.

2 Complexity Analysis

To stress the analytical improvements that the current algorithm brings, we compare its complexity with some of the existing solutions. We can consider standard complexity measures for distributed algorithms, such as:

- Space per node per edge of the overlay network. We consider the size of the data structures that each node needs to store.
- Communication per edge of the overlay network. We consider this measure more relevant for peer to peer systems than the more frequently used "total number of messages" measure, because it captures the congestion that the algorithm induces on each link.
- Time elapsed from input to output, normalized over network diameter (number of roundtrips).
- Forced disk writes required per transaction in the system.

We can distinguish between complexity of storing and communicating the *data* and complexity of the (protocol-induced) overhead in messages and data structures. This overhead can be traced to responses to topological changes as well as to responses to transactions.

The current solution has smaller overhead per transaction, but pays more per topology change. This is however desirable as it is assumed that the transaction frequency is significantly higher than the rate of network changes. The number of messages sent per transaction, per link is $1 + \frac{2}{\Delta}$, where Δ is the aggregation coefficient (i.e the number of transactions per pulse in our algorithm, or the number of transactions per token cycle in the group communication implementation based on a ring protocol).

To measure the overhead paid for a topology change we consider as one topology change as any number of failures that are detected "simultaneously". The present algorithm has constant overhead in time and space, and $\log n$ overhead in number of messages sent per link per topological change. The logarithmic factor is the price paid to compute a spanning tree [GHS83, Awe87]. Previous solutions based on group communication require linear ($\Omega(n)$, where n is the number of nodes) overhead in communication and space, because of the inherent "membership learning" component, making them unscalable. Membership learning means that all the nodes must know the ID of all other nodes in the same connected component, which requires communicating linear ($\Omega(n)$) amount of information over a single edge, as well as storing linear ($\Omega(n)$) information at each node.

Prior solution due to Lamport [Lam98] is difficult to evaluate, but it appears to require linear space and communication overhead as well.

The time complexities of all these solutions are of the order of magnitude of cross-network latency, i.e. constant factor away from optimum.

3 Basic Algorithm

To give a better intuition of the role of each component of the algorithm, we will develop our solution gradually, by relaxing the working network model from a theoretical static synchronous network to the internet-like partitionable, asynchronous network model. The final algorithm can be viewed as consisting of several well-defined modules (Figure 1). The modules are designed to be oblivious to the changes in the model; a module is designed to run under certain preconditions and the subsequent additions to the algorithm will ensure the preservation of those preconditions despite the changes in the network model. We establish an invariant which will guarantee the correctness of the algorithm and we will show that, as we add more modules to the algorithm, the invariant is preserved.

The algorithms are presented in a generic pseudo code form which uses standard notation conventions: **some_message**(*data*) denotes a message passed through the network containing the relevant information *data; some_variable* denotes a local variable.

3.1 Synchronous Network Solution

If we consider the static network model given by G(V, E) where all communication is synchronous and no server or network failure is possible, the problem is trivially solved. Each update is propagated through the network to all the nodes. Since the system is synchronous all the updates can be ordered and executed based on the ordering of the local timestamps attached to them and the "ID" of the update (which can be the contents of the update, or the ID of the issuing node, if available).

3.2 Static Asynchronous Network Solution

Let's consider a static network (no server or network failures) with asynchronous communication. We distinguish two steps in this situation.

Initialization: Construct a spanning tree over the given network. This is a well-studied problem with a wide range of solutions [GHS83, Awe87]. In the appendix we give a simple possible solution.

Synchronization: The algorithm will simply emulate a synchronous environment, by maintaining a running "virtual clock". Each "pulse" of the clock is generated by flooding a message over the fixed tree constructed in the previous step and converge-casting acks over subtrees (see Algorithm 1).

Actions are sent in FIFO order on each link, along the same tree. For the simplicity of the analysis, but without reducing generality, we can assume that actions are initiated only by the root of the tree. Obviously, any node can first send their actions to the root which can introduce them into the system. Section 5 will analyze the impact of transparently allowing nodes to inject actions in the system. Furthermore we will consider that all the **actions are sent together with the pulse message.** These choices do not restrict in any way the generality of the model as we will show in section 5, but they significantly simplify the presentation and the understanding of the solution.

Each node keeps track of the virtual clock using the local variable *localClock* which is updated based on the **pulse** messages. Each node i will also maintain a buffer β_p^i,[2] containing the ordered set of messages received during pulse p (when $localClock = p$). We denote $\overline{\beta_p^i}$ the ordered content of β_p^i at the end of pulse p (when the local clock is incremented). A new pulse is generated by the root only after receiving **pulse_ack** messages from every node. When receiving pulse p+1, each node commits the messages belonging to the previous pulse[3].

We introduce the following invariant:

Pulse invariant: $\overline{\beta_p^i} = \overline{\beta_p^j} \equiv \overline{\beta_p}, \forall i, j, p$

Lemma 1

Algorithm 1 maintains the Pulse Invariant on a static asynchronous network.

Proof:

The invariant is trivially satisfied when the algorithm starts as all the buffers will be empty.

Since the links of the tree are reliable and FIFO, every node will receive all the messages generated by the root, in the same order. When the root initiates the message **pulse**$(p{+}1)$, it wil not send any more messages belonging to pulse p. Since **pulse** and **pulse_ack** messages are sent on the same tree, the causal relationship between the reception of the messages **pulse**(p), data for pulse p and **pulse**$(p{+}1)$ will be maintained at each node. This guarantees the inviolation of the invariant.

□

Note: On a static network where only the root sends messages on a spanning tree, there is no need for pulse messages in order to guarantee the distributed commit. The FIFO and reliability guarantees on the links are sufficient. The pulse will become useful however when used as a piece of a more general algorithm, under the model of a dynamic network, in the following sections.

[2]When there is no confusion about which node we are referring to, we will use the simplified notation β_p.

[3]The remark *if not previously committed* in the algorithm code is meant to protect from situations that will be described later, where actions will be retransmitted after network changes have occurred and a node may receive the same pulse message again as we see later in the presentation. To enforce this the algorithm either queries the database or writes to private permanent storage the last committed pulse.

Algorithm 1 Pulse Algorithm (Synchronizer)

```
when: receive data_message from parent do
  store message in β_{localClock}
  forward data_message to all children c in children_list
when: receive pulse(p) message from parent do
  commit, if not previously committed, all messages in β_{p−1}
  localClock ← p
  for all children c in children_list do
    pulseack[c] ← false
    forward pulse(p) message to c
when: receive pulse_ack(p) message from c do
  pulseack[c] ← true
  if pulseack[c] = true for all children c then
    if parent ≠ nil then
      send pulse_ack(p) message to parent
    else { if parent = nil , i.e. root of the tree }
      commit, if not previously committed, all messages in β_p
      execute generate_next_pulse()
when: invoked generate_next_pulse() do { code for Procedure generate_next_pulse()}
  localClock ← localClock + 1
  generate pulse(localClock) message and send to all children
  send data messages for pulse localClock to all children
```

4 Complete Algorithm

In this section we will consider an asynchronous network whose connectivity may change in time; the network is described by the dynamic graph G(t) = (V(t), E(t)).

We can now introduce all the components of our algorithm model. Figure 1 illustrates the event flow between the various modules of the algorithm. The figure also maps to the state machine of the whole algorithm, as the different modules correspond to code executed in different states. The overlay network wrapper contains an edge failure/recovery detection mechanism that runs independently of the other components. When a change in the connectivity of the underlying network is detected, the wrapper sends a **freeze** signal to all the components, stopping their execution. The reset procedure (Algorithm 6) is triggered, which simply restarts the Spanning Tree construction algorithm when a topological change occurs. This mechanism has been studied in detail by Afek, Awerbuch and Gafni [AAG87]. When the spanning tree is complete the Reconciliation Algorithm is notified. The Reconciliation Algorithm is basically a brute force reinstatement of the Pulse Invariant after a network change has occurred. The power of the algorithm lies in its simplicity and strict specifications of its modules.

4.1 Fully Connected Dynamic Network Solution

For the moment we assume that global connectivity is maintained even when some links fail. Nodes do not fail. There is always a spanning tree for the whole network, but link failures may force the rebuilding of the tree.

Consider the node i with highest pulse number *maxpulse*. The highest completed pulse in the system is *maxpulse-1*. Even though some messages may have been sent during pulse *maxpulse*,

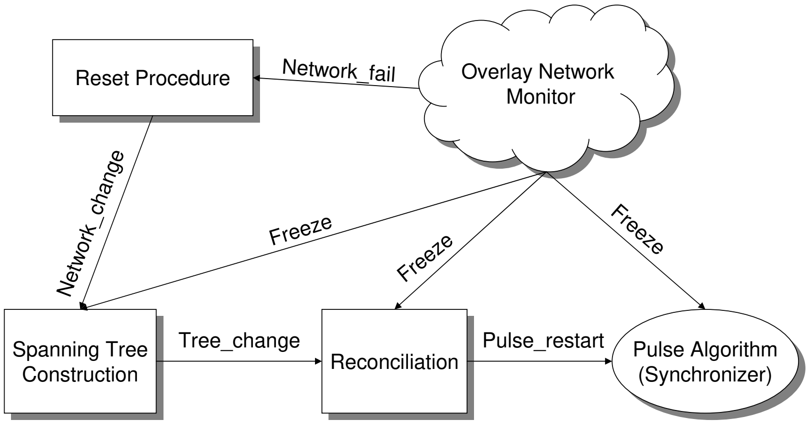

Figure 1: Conceptual Replication Algorithm

since that pulse was not completed, it needs to be re-executed in order to guarantee the invariant preservation. The Reconciliation algorithm will reset the *localClock* of the tree root to *maxpulse-1*[4]. In a practical implementation this value can be trivially determined and set while building the spanning tree. The reconciliation algorithm will send a **pulse_restart**() notification to the Synchronizer upon its termination. The pulse algorithm resumes with the tree root generating pulse *maxpulse* and resending all the messages corresponding to that pulse. The Synchronizer Algorithm is enhanced with the following wrapper in order to handle the two notifications.

Algorithm 2 Wrapper for Pulse Algorithm

1: **when:** receive **freeze**() notification from reset procedure **do**
2:    freeze execution
3: **when:** receive **pulse_restart**() notification from reconciliation module **do**
4:    **if** $(parent = NIL)$ **then**
5:       execute **generate_next_pulse**()

Lemma 2
Algorithms 2,1 maintain the Pulse invariant under a fully connected dynamic network

Proof:

A new pulse is generated only if the algorithm executes line PA_{17} (line 17 of the Pulse Algorithm) or lines W_5 (line 5 of the Wrapper).

At both points the *localClock* of the root is incremented and the next consecutive pulse is generated. If the new pulse is created in line PA_{17}, then the invariant is maintained following Lemma 1 as it means the network was static for the duration of the pulse. If the maximum pulse number detected in the system is *maxpulse* then there was a node i that previously generated pulse number *maxpulse*. That could have happened only in lines PA_{17} or W_5, as mentioned before. By induction, since, according to the algorithm, line W_5 will never create a pulse number that wasn't already in the system, it follows that pulse number *maxpulse* was initially originated by a note j executing line PA_{17}. When j created **pulse**(*maxpulse*) all the nodes in the tree have already received the data corresponding to pulse *maxpulse-1* (the data is piggy-backed on the **pulse**(*maxpulse-1*) message). When the algorithm restarts the pulse all the nodes will be able to commit on $\overline{\beta_{maxpulse-1}}$.

[4]As no node disconnects we can assume that the new spanning tree has the same root as the old one.

□

4.2 Fail-Stop Network Solution

Let's consider now the situation where link failures can partition the network into separate components which are unable to communicate with each other. If, for example, we have two such components, A and B, they are completely disjoint and a node in A cannot communicate with any node in B, but can communicate with any other node from A.

Among the set of different connected components we will identify exactly one component as being the *primary* component. The primary component is determined using a *quorum* system. The synchronizer algorithm will continue to execute only in the primary component. We also extend the model to support node failures (crashes). At this point we consider that failed nodes do not recover; nodes that become disconnected from the primary component are also considered failed and do not rejoin the system.

As mentioned, the primary component is determined using a quorum system. Quorum systems and their properties have been largely studied in the literature [AW96, JM90, Woo98]. Examples of quorum systems include majority, monarchy, tree-based and dynamic linear voting.

The algorithm remains basically the same, but the reconciliation phase will also collect the data necessary to determine the primary component. This information could be collected by the tree construction algorithm, thus improving the efficiency. For clarity purposes however, we prefer to keep the algorithm blocks independent. The algorithm stops in all non-primary components. In Appendix A we show a sample implementation of a quorum detection mechanism based on weighted majority.

The fail-stop model exhibits the famous Consensus impossibility problem, presented by Fischer, Lynch and Paterson [FLP85]. Consider *maxpulse* to be the maximum localClock value in a component. It is possible that some members (now in a different component) completed the pulse *maxpulse* and moved on to the following pulse, committing on the messages from pulse *maxpulse.* Alternatively, it is possible that there are nodes that didn't advance to pulse *maxpulse* and therefore didn't even commit on pulse *maxpulse-1.* If all the nodes in the current component have the same local clock, they cannot tell which of the above scenarios is present in the system. However, the key element in overcoming this problem is the fact that the two described situations cannot occur simultaneously!

Lemma 3
The algorithm maintains the Pulse Invariant under a fail-stop network model.

Proof:(Sketch)

Let *maxpulse* be the maximum pulse number in the primary component. According to Lemma 2, the invariant is maintained for all *p¡maxpulse.* As the pulse algorithm resumes, all the messages belonging to pulse *maxpulse* will be retransmitted. If there exists (outside the component) a node *j* with *localClock=maxpulse+1*, it means that it already committed on the messages of pulse *maxpulse.* However, in this case all the nodes in the primary will have *localClock=maxpulse* and will have $\beta_{maxpulse} = \overline{\beta_{maxpulse}}$ and they will commit on the same set of messages as node *j*.

According to the quorum guarantees, at most one component will be chosen as primary and will continue the execution of the Pulse algorithm. According to the algorithm all the other components will never commit any action from this moment on. Thus, the invariant is preserved.

□

4.3 Partitionable Network

We finally consider the most general network model G(V(t), E(t)) where nodes can fail and subsequently recover; link failures/recoveries can cause the network graph to partition into separate components and partitioned components can re-merge. All these events can occur independently, in cascade or simultaneously, in any combination. As before, at all times only one component will be identified as the primary component and the pulse algorithm will be executed only in the primary component. However, the nodes in the non-primary components are not forever stopped; instead, they may exchange information with other nodes, as the network connectivity changes, and even rejoin the primary component.

This model introduces a new complication. When several previously separated components merge, they will have different knowledge about the state of the system and will need to be updated to the most recent information available in the component. For simplicity, but without restricting the generality, we can assume that the root of the new tree is one of the "most updated" nodes in the component (it has the maximum pulse number). This can be trivially enforced upon the construction of the spanning tree.

Algorithm 3 Reconciliation Algorithm

```
when: receive tree_change() notification from Spanning Tree Construction module do
  execute LinkUpdate() procedure
  if (parent = NIL) then
    send reconcile(localClock) message to all children
when: receive reconcile(p) from parent do
  reconcile ← TRUE
  reconciledClock ← p
when: (reconcile = TRUE) and (localClock = reconciledClock) do
  if (children_list ≠ NIL) then
    send reconcile(localClock) to all children
  else
    send reconcile_ack to parent
when: receive reconcile_ack from j do
  reconciled[j] ← TRUE
  if (reconciled[k] = TRUE for all k in children_list) then
    if (parent ≠ NIL) then
      send reconcile_ack to parent
    else if (Quorum()) then
      send restart_pulse() notification
```

The Reconciliation procedure will bring all nodes to the same *localClock* value and update the local buffers correspondingly in order to re-establish the invariant. The more updated nodes will pass their information to the less updated ones. Actions belonging to completed pulses were already committed by the primary component and therefore can be safely committed by the less updated members at this time. This phase proceeds locally, in parallel on each link of the overlay network. At the same time the root will start a termination detection wave by sending a **reconcile** message. When a node is reconciled with its parent, the **reconcile** message is propagated on the tree towards the leaves. When finished reconciling, the leaves of the tree will generate **reconcile_ack** messages marking the completion of the procedure. These acknowledgments are propagated back towards the root once each node has received a **reconcile_ack** message from all its children. All this process is executed for any changes in the connectivity, and will be triggered even when non-primary components merge. When the reconciliation phase is finished, if the newly formed component has the quorum, it will restart the pulse algorithm.

The *reconciledPulse* variable keeps track of the maximum pulse number in the component in order to detect the termination of the **LinkUpdate** procedure. The *clocks*[] array in the **LinkUpdate** procedure keeps track of the current state of each neighbour as far as this node knows, in order to verify the completion of the updating procedure. Note that the updating mechanism is not restricted to the tree links but can use all available connections in order to speed the process. However, as it is presented, the **LinkUpdate** procedure is inefficient because several nodes may redundantly attempt to update the same node. This can be avoided with more careful information exchange, but this method was preferred to illustrate the simplicity of the basic process.

Algorithm 4 LinkUpdate procedure

```
send state(localClock) to all neighbours
when: receive state(clock) from j  do
  clocks[j] ← clock
  execute UpdateLinks()
when: receive data_message(msg, p) from j  do
  store msg in β_p
  if (received all messages for pulse p)  then
    commit messages in β_p
    localClock ← p + 1
    execute UpdateLinks()
when: invoked UpdateLinks()  do
  for all neighbours k  do
    if (localClock ¿ clocks[k])  then
      send in order all messages in β_p to k, p=clocks[k]..localClock-1
      clocks[k] ← localClock
```

Lemma 4
The algorithm preserves the pulse invariant under a partitionable network model.

Proof: (Sketch)

Let's consider two disjoint components C_1 and C_2. Let's assume that none of these components experienced joins or merges since they were disconnected from the primary (possibly they experienced several leaves/partitions). The reconciliation procedure will only update the nodes to the most current state, and only messages already committed by some node will be committed, therefore preserving the invariant. Due to the quorum properties only one component continues executing the pulse after the reconciliation is completed. As guaranteed by Lemma 3, the invariant will be maintained in this component for the incompleted *maxpulse* pulse as well as for the previous pulses.

□

Theorem 1
The algorithm satisfies the Correctness and Liveness properties specified in Section 1.1.

Proof: (Intuition)

The liveness is intuitively guaranteed as none of the components of the algorithm can block indefinitely, as long as the network is stable for a sufficient period of time. A formal analysis is possible, but we have to omit it for space considerations. The global total order is guaranteed by the two-layered ordering that the algorithm provides: within each pulse messages are strictly ordered preserving FIFO properties. The pulses form a unique increasing sequence. When a pulse is repeated, only messages that were originally part of that pulse will be retransmitted and their order will remain unaltered. Finally, the quorum mechanism guarantees the uniqueness of the pulse number sequence which therefore translates in the unique, global ordering of actions

□

5 Discussion

In section 3.2 we chose to only allow the root of the spanning tree to send messages and only at the beginning of a pulse (piggy-backed on the pulse message). This is not a restriction in the model, but merely a simplification to allow a clearer presentation of the algorithm.

Let's consider the model where still only the root is allowed to send data messages, but messages can be sent at any time through the course of a pulse. In this situation, when a node j receives **pulse**(*p+1*), it doesn't have the guarantee that all the other nodes have received all the messages generated during pulse p and cannot yet commit on $\overline{\beta_p}$ (although j has all these messages). If j would commit it is possible that all the nodes that committed on this set are disconnected from the nodes that didn't receive the complete set which will go on and form the next primary. The invariant will then be violated as the primary will commit on a smaller set and incorrectly advance to the next pulse. Therefore the algorithm has to be changed to commit $\overline{\beta_{p-1}}$ only when it receives **pulse**(*p+1*). The reconciliation procedure will also need to be changed to retransmit messages from pulse *maxpulse-1* to all members of the component, before restarting the pulse *maxpulse* as there may be participants that do not have all these messages; however, the nodes with *localClock=maxpulse* are guaranteed to have $\overline{\beta_{maxpulse}}$ and can therefore update the others.

It is quite straightforward now to see that any node can actually generate messages and disseminate them through the tree. Let m be a message generated by node i during pulse p; m is disseminated through the tree and each node will store it in β_p as it receives it. Since m is generated before the **pulse_ack**(p+1) message it will reach the root of the tree before the beginning of pulse p+*2*. Any node j in the component is guaranteed to receive m before it receives **pulse**(*p+2*) and will be able to commit on it upon reception of **pulse**(*p+3*).

Another aspect that was not discussed explicitly in the algorithm is the need for logging to permanent storage in order to protect the algorithm from node crashes. Each action that is injected into the system needs to be written to disk by at least one node, the one creating the action. This is needed in order to guarantee that the action is not completely lost in case of a quorum crash, when some nodes committed the action while others were about to.

Also each node that becomes part of the primary component needs to mark this membership on disk. When a member of the quorum recovers after a crash, it cannot become part of a new quorum unless it finds out that the quorum he was part of was superseded by subsequent quorums, or unless it talks with all the members of the quorum he was a member of (this is needed when the whole quorum crashed), as the node needs to guarantee that he has the complete information about the actions committed in the pulse that he was part of before it can restart executing the algorithm. These requirements are identical to the ones needed by [AT01] and are further detailed there.

The algorithm appears to suffer in comparison to other solutions as it commits "together" all the actions that were generated during a given pulse. Group communication based solutions do, in reality, the same thing: they commit an action when the total order primitive has established the order of the action. An efficient group communication implementation will actually use a method (ring protocol, tree) that also aggregates the ordering of all the messages that occur during a roundtrip time which is what we do as well. Two-phase commit does commit each action individually, but it takes just as much time as our algorithm needs to aggregate and commit several actions.

This analysis shows the generality of our algorithm despite its apparent simplicity. The simplicity, however, is a great bonus as it allows the design of a clean, provably correct implementation. Furthermore, although the algorithms presented in this paper do not have this property, we believe that a self-stabilizing version of the algorithm can be designed.

References

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *IEEE Symposium on Foundations of Computer Science*, pages 358–370, 1987.

[Ami95] Y. Amir. *Replication Using Group Communication over a Partitioned Network.* PhD thesis, Hebrew University of Jerusalem, Jerusalem, Israel, 1995. http://www.cnds.jhu.edu/publications/yair-phd.ps.

[AT01] Y. Amir and C. Tutu. From total order to database replication. Technical Report CNDS 2001-6, Johns Hopkins University, November 2001.

[AW96] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *Symposium on Fault-Tolerant Computing*, pages 26–35, 1996.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985.

[Awe87] Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems (detailed summary). In *ACM Symposium on Theory of Computing*, pages 230–240, 1987.

[FLP85] M. H. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FLS01] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001.

[GHS83] R. Gallager, P. Humblet, and P. Spira. A distributed algorithm for minimum-weight spanning trees. 5(1):66–77, 1983.

[GR93] J. N. Gray and A. Reuter. *Transaction Processing: concepts and techniques.* Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[JM90] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.

[KD95] I. Keidar and D. Dolev. Increasing the resilience of atomic commit at no additional cost. In *Symposium on Principles of Database Systems*, pages 245–254, 1995.

[Kei94] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lam01] Butler Lampson. The ABCDs of paxos. Presented at Principles of Distributed Computing, 2001.

[LL93] B. Lampson and D. Lomet. A new presumed commit optimization for two phase commit. Technical Report CRL 93/1, Digital Cambridge Research Laboratory, One Kendall Square, Cambridge, Massachussets 02139, February 1993.

[Ske82] D. Skeen. A quorum-based commit protocol. Berkley Workshop on Distributed Data Management and Computer Networks, February 1982.

[Woo98] A. Wool. Quorum systems in replicated databases science or fiction. *Bulletin of the Technical Commitee on Data Engineering*, 21(4):3–11, December 1998.

A Classic Algorithms

The appendix contains simple sample implementations of spanning tree construction, reset procedure and quorum detection algorithm. They can be replaced by any other standard or more performant implementation.

Algorithm 5 Static Spanning Tree Algorithm

```
when: receive network_change() from Reset procedure do
  leader ← self
  send ID(self) to all neighbours
when: receive ID(id) from n do
  if leader ¡ id then
    leader ← id
    parent ← n
    forward ID(id) to all neighbours
  else
    send ACK(0) to n
when: receive ACK(b) from j do
  if (b = 1) then
    add j to children_list
  if received ACK messages from all neighbours except parent then
    if (parent ≠ NIL) then
      send ACK(1) to parent
    else
      send tree_change() notification to Reconciliation Algorithm
```

The quorum algorithm implements a simple weighted majority algorithm. Each node i will write to permanent storage its weight w_i . The distributed procedure will add these values for the members of the current component. The root of the tree can decide if the quorum condition is satisfied and will be able to start the pulse algorithm. This particular quorum method requires that everybody knows apriori the total weight of the system *qweight* in order to detect the presence of a majority. This limitation is not inherent to the whole algorithm, but only to the particular quorum algorithm that we use in this example.

Algorithm 6 Reset Procedure

when: receive **network_fail**() from network monitor **do**
  $changeNo \leftarrow changeNo + 1$
  execute **BroadcastChange**()
when: receive **change**($change$) **do**
  $changeNo \leftarrow change$
when: invoked **BroadcastChange**() **do**
  send **change**($changeNo$) to all neighbours
  send **network_change**($changeNo$)

Algorithm 7 Quorum Procedure (part of Reconciliation Algorithm)

current_weight ¡- w_i
if (*children_list* = NIL) **then**
  send message weight(w_i) to *parent*
when: receive weight(w) from child **do**
  current_weight ← *current_weight* + w
  if received weight message from all children **then**
    if ($parent \neq nil$) **then**
      send weight(*current_weight*) to *parent*
    else if (*current_weight* ¿ $\frac{qweight}{2}$) **then**
      return TRUE
    else
      return FALSE