

Consistent Distributed Commit: A Constant Overhead Solution

Baruch Awerbuch

Xiaoqi Lu

Ciprian Tutu

Johns Hopkins University
{baruch, xiaoqilu, ciprian}@cnds.jhu.edu
Technical Report
CNDS-2002-6
<http://www.cnds.jhu.edu>

December 8, 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.

We introduce a modular solution based on several well defined blocks with clear formal specifications. These blocks were extensively studied in the literature but their combined potential was never exploited in order to provide database commit. The blocks can be implemented in a variety of ways and we give simple examples of possible implementations. The algorithm gains its efficiency by aggregating the acknowledgments required for each transaction. Also, in contrast with existing solutions, our algorithm does not require any membership knowledge. Components are detected based on local information and the data is disseminated over an overlay spanning tree.

We provide an analytical evaluation of the algorithm performance as well as experimental results.

1 Introduction

We present an algorithm for persistent consistent distributed commit (distributed database commit) in a dynamic, asynchronous, peer to peer network. We introduce a modular solution based on several well defined blocks with clear formal specifications. These modules were each widely researched independently in the literature, but their combined power to provide distributed commit services in an efficient and straightforward algorithmic way was not realized until now.

Most of the existing distributed commit solutions require acknowledgments from every participant for each action. Our algorithm uses implicit, aggregated acknowledgements during normal operation, when the network is stable. We construct a stable overlay spanning tree over the set of active, connected nodes. The use of a spanning tree as the underlying data structure guarantees an efficient communication pattern and facilitates the deployment of simple algorithms with clear correctness proofs.

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, we can establish a global total order on the actions so all the nodes commit the actions in the same order. A pulse message p will implicitly acknowledge all messages sent during the previous pulse $p-1$ ¹. The idea of using the pulse mechanism as a synchronizer is derived from [Awe85]. The algorithm is enhanced to protect against network partitions/merges and maintain consistency even in their presence, thus bridging a gap that most existing atomic commit protocols do not address. The algorithm continues to make progress as long as a quorum of nodes remains connected.

Our solution consists of several independent modules whose interaction is well defined, based on their external signatures. The complexity of each module depends on the generality of the environment for which the solution is developed. We analyze the complexity bounds of the complete algorithm and compare it with existing methods. In contrast with previous analysis, we are concerned with the complete cost of the solution, including all levels of abstraction that other methods rarely consider when estimating the overall complexity bounds of the system. We also practically evaluate the performance of our system and compare it with several established methods, in both local and wide area network settings.

We show that simple design can yield an efficient solution to a problem that traditionally required very complex algorithmic solutions. The modular design can be used for a future detailed analysis of the trade-offs involved and finer tuning to the specific network environments in which the solution is deployed.

The rest of the paper is organized as follows: the following subsections describe the model and relate to existing work. Section 2 introduces the main component of the algorithm and informally proves its correctness. Section 3 presents the complete algorithm that supports dynamic networks. Section 4 discusses the generality of the algorithm. Section 5 evaluates the complexity bounds of the algorithm and compares it to existing solutions. Section 6 evaluates the practical performance of the algorithm. We conclude in section 7.

1.1 Model

We consider a dynamic asynchronous network susceptible to network partitions/re-merges and server crashes/recoveries that can occur independently or in cascade. 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 also assume that each node is capable of detecting failures of the physical network that disrupt the overlay topology. In order to guarantee liveness of the protocol, the failure detector needs to be in the class $\diamond S$ as defined by Chandra and Toueg [CT96]. Note that these assumptions do not reduce the generality of the solution; in practice, these properties can be satisfied using the standard TCP protocol as a link communication protocol between the nodes.

Each node holds a local permanent copy of the distributed database² and may issue read/write operations on behalf of the clients that connect to it. All the write operations must be eventually committed in agreement at all nodes. Read operations can be answered locally, after the causally preceding writes have been committed.

The correctness properties satisfied by the algorithm resemble those of a quorum based atomic commit protocol. The algorithm establishes a global total order of actions that is consistent with

¹Depending 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 4.

²We will use the word *database* leisurely in this paper, referring only to the atomic commitment aspect of distributed database operations. This model can be used in conjunction with a real database if we assume non-interactive, deterministic transactions.

FIFO specifications. If all the servers commit the actions in this agreed order, the commit consistency is guaranteed. The correctness specification below is similar to the one used in [AT02].

Property 1 (Global Total Order/Agreement) *If both servers s and r commit their i th action, then these actions are identical.*

Property 2 (Global FIFO Order) *If server r commits 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 commits action a .*

1.2 Related Work

Two-phase commit (2PC) [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 in order to recover from some simple fault scenarios. Quorum based three-phase commit protocols (3PC) [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, Lam01]. 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, allowing for different quorums to participate in the two phases of the algorithm. The algorithm however pays in efficiency the increase in fault-tolerance.

Keidar [Kei94] proposed 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 [AT02] also use group communication in order to improve communication performance. Their algorithm aggregates the acknowledgments required for each action, further improving 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. We note that 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 Basic Algorithm

The algorithm executed at each node progresses through several states. The procedures corresponding to each state assume certain preconditions are satisfied and enforce certain guarantees before switching to the next state. We sometimes refer to the algorithms executed in each state as *modules* since they can be designed completely independent of the other components as long as they enforce the requirements.

To give a better intuition of the role of each component of the algorithm, we develop our solution gradually, by relaxing the working network model from a theoretical static synchronous network to the

Input	Network_change(), Freeze
Precondition	Stable overlay network
Provides	Spanning tree over the overlay network
Output	Tree_change()

Table 1: Spanning Tree Module Specifications

internet-like partitionable, asynchronous network model. The properties that each module satisfies and its role are defined as each module is introduced.

We define an invariant³ which guarantees the correctness of the algorithm and we informally show that the algorithm maintains this invariant, reenforcing it after failures or partitions.

The algorithms are presented in a generic pseudocode 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.

2.1 Synchronous Network

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).

2.2 Static Asynchronous Network

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 present a simple algorithm that could be used for the Initialization procedure⁴. This procedure assumes a stable underlying network and builds a spanning tree over the connected nodes (Table 1).

Synchronization: The algorithm emulates 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 acknowledgements 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 subsequently introduce them into the system. Section 4 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

³We slightly abuse the term *invariant* to refer to an eventual property that holds in the operational states of the algorithm.

⁴The algorithm presented in the appendix does not have the good complexity bounds that the cited solutions have, but is presented for completeness of the solution

Input	Pulse_restart(), Freeze
Precondition	Pulse Invariant, stable spanning tree, identical state within connected component
Provides	Consistent Distributed Commit
Output	Client/Database interaction

Table 2: Pulse Module Specifications

message. These choices do not restrict in any way the generality of the model as we will show in section 4, but they significantly simplify the presentation and the understanding of the basic modules.

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 ,⁵ 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 pulse⁶.

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 will 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 (because of the FIFO property of the links). 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. Under all the models, the pulse module will work under the specification in table 2.

3 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.

⁵When there is no confusion about which node we are referring to, we will use the simplified notation β_p .

⁶The 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)

```
1: when: receive data_message from parent do
2:   store message in  $\beta_{localClock}$ 
3:   forward data_message to all children c in children_list
4: when: receive pulse(p) message from parent do
5:   commit, if not previously committed, all messages in  $\beta_{p-1}$ 
6:    $localClock \leftarrow p$ 
7:   for all children c in children_list do
8:      $pulseack[c] \leftarrow false$ 
9:     forward pulse(p) message to c
10: when: receive pulse_ack(p) message from c do
11:    $pulseack[c] \leftarrow true$ 
12:   if  $pulseack[c] = true$  for all children c then
13:     if parent  $\neq nil$  then
14:       send pulse_ack(p) message to parent
15:     else { if parent = nil , i.e. root of the tree }
16:       commit, if not previously committed, all messages in  $\beta_p$ 
17:       execute generate_next_pulse()
18: when: invoked generate_next_pulse() do { code for Procedure generate_next_pulse() }
19:    $localClock \leftarrow localClock + 1$ 
20:   generate pulse( $localClock$ ) message and send to all children
21:   send data messages for pulse  $localClock$  to all children
```

Input	Network_fail()
Precondition	Asynchronous network with Unreliable Failure Detector
Provides	Stable overlay network
Output	Network_change()

Table 3: Reset Module Specifications

Each node is enhanced with an unreliable failure detector that runs independently of the other components. When a change in the connectivity of the underlying network is detected, the failure detector sends a **freeze** signal to the running algorithm and a **network_change()** notification will start the Reset procedure.

The reset procedure (Algorithm 6) simply restarts the Spanning Tree construction algorithm when a topological change occurs, allowing the Spanning Tree construction to proceed under a seemingly static network. 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 complete algorithm lies in its simplicity and independent specifications of its modules.

3.1 Fully Connected Dynamic Network

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

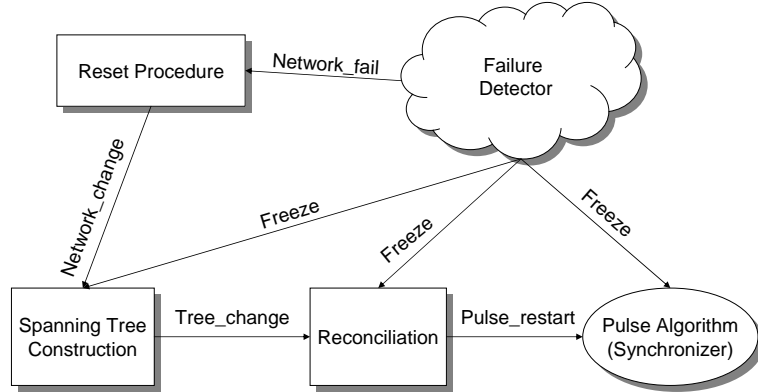


Figure 1: Conceptual Replication Algorithm

Input	Tree_change(), Freeze
Precondition	Stable spanning tree over the overlay network
Provides	Pulse Invariant, Identical state within connected component
Output	Pulse_restart()

Table 4: Reconciliation Procedure Specifications

rebuilding of the tree⁷.

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 $maxpulse$, 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 $maxpulse-1$ ⁸. 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 re-sending 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 failure detector **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:

⁷An optimal failure detector will notify the algorithm of a network change, only if that change affects the existing spanning tree.

⁸As no node disconnects we can assume that the new spanning tree has the same root as the old one.

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 node 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 $\beta_{maxpulse-1}$. □

3.2 Fail-Stop Network

We extend the model to support node failures (crashes). At this point we consider that failed nodes do not recover. We also consider 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. In the fail-stop model, the nodes that become disconnected from the primary component are considered failed and do not re-merge with the primary.

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 the fail-stop network model.*

Proof:

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 j already committed on the messages of pulse $maxpulse$. In this case, all the nodes in the primary will have $localClock = maxpulse$ and $\beta_{maxpulse} = \overline{\beta_{maxpulse}}$, therefore they will commit on the same set of messages as node j maintaining the consistency for pulse $maxpulse$. If however, outside the new primary component, there exists a node i , with $localClock = maxpulse - 1$, then the primary will continue the synchronizer algorithm from pulse $maxpulse$ which does not contradict the state at node i .

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. □

3.3 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.

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. 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 3 Reconciliation Algorithm

```
1: when: receive tree_change() notification from Spanning Tree Construction module do
2:   execute LinkUpdate() procedure
3:   if (parent = NIL) then
4:     send reconcile(localClock) message to all children
5:   when: receive reconcile(p) from parent do
6:     reconcile  $\leftarrow$  TRUE
7:     reconciledClock  $\leftarrow$  p
8:   when: (reconcile = TRUE) and (localClock = reconciledClock) do
9:     if (children_list  $\neq$  NIL) then
10:      send reconcile(localClock) to all children
11:    else
12:      send reconcile_ack to parent
13:   when: receive reconcile_ack from j do
14:     reconciled[j]  $\leftarrow$  TRUE
15:     if (reconciled[k] = TRUE for all k in children_list) then
16:       if (parent  $\neq$  NIL) then
17:         send reconcile_ack to parent
18:       else if (Quorum( $\bar{\quad}$ )) then
19:         send restart_pulse() notification
```

Algorithm 4 LinkUpdate procedure

```
1: send state(localClock) to all neighbours
2: when: receive state(clock) from j do
3:   clocks[j]  $\leftarrow$  clock
4:   execute UpdateLinks()
5: when: receive data_message(msg, p) from j do
6:   store msg in  $\beta_p$ 
7:   if (received all messages for pulse p) then
8:     commit messages in  $\beta_p$ 
9:     localClock  $\leftarrow$  p + 1
10:    execute UpdateLinks()
11: when: invoked UpdateLinks() do
12:   for all neighbours k do
13:     if (localClock > clocks[k]) then
14:       send in order all messages in  $\beta_p$  to k,  $p = \text{clocks}[k].\text{localClock} - 1$ 
15:       clocks[k]  $\leftarrow$  localClock
```

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

Proof:

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 use of a $\diamond S$ unreliable failure detector guarantees that a fault in the network will be detected and the reconfiguration procedure triggered. Liveness is then 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. 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 uniquely 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 □

It is important to remark that the root of the tree, although plays a special role during the normal execution of the algorithm, does not introduce a single point of failure into the system. A crash/disconnection of the root node is, from a fault-tolerance perspective, indistinguishable from a crash of a regular node of the primary component, the algorithm being able to continue correctly (after reconfiguring), as long as a quorum remains connected.

4 Discussion

In section 2.2 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 these messages it is possible that all the nodes that committed on this set are disconnected from the nodes that didn't yet receive the complete set. This second set of nodes can go on and form the next primary. The invariant is then violated as the new primary commits on a smaller set and incorrectly advances 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 $\overline{\beta}_{\text{maxpulse}}$ 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 it⁹. 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 [AT02] 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. One other apparent drawback of the algorithm is the fact that any topology change that affects the constructed spanning tree will cause the execution of a reconciliation phase and the rebuilding of a tree. This is a design choice, based on the assumption that failures are rare enough that significant amount of actions can be committed. Furthermore, this situation can be improved by appropriately selecting the quorum nodes (their number and their location) either based on stability measures or on randomized, probabilistic principles.

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 version of the algorithm presented in this paper does not have this property, we believe that a self-stabilizing version of the algorithm can be designed.

5 Complexity Analysis

We now compare the analytical complexity of our algorithm with some of the existing solutions. Since our algorithm is fully specified starting from the network level, our analysis includes *all* the communication that the algorithm entails. Because of this, we attempt to analyze the existing solutions at the same level, rather than using standard analysis methods that hide the overall complexity in the use of various abstractions - such as totally ordered multicast messages employed by group communication solutions. We consider that an all-inclusive analysis is valuable in the understanding

⁹In the previous sections we referred to β_p as storing the messages received in pulse p , rather than those created in pulse p . In the current setup there is a difference between the two and the correctness follows more naturally if we consider the buffers storing messages based on their creation pulse.

Algorithm\Measure	Space	#Mesgs	disk writes	time	space	#mesgs	time
2PC[GR93]	O(1)	3	n+1	2	0	0	0
Paxos [Lam98]	O(1)	4	2n+1	3	0	0	0
[Ami95, AT02]	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 5: Complexity Comparison

of the various trade-offs between methods.

We introduce a set of complexity measures and argue their relevance for distributed algorithms.

- Space *per node* 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 (latency), *normalized* over network diameter (number of roundtrips).
- Forced disk writes required per transaction in the system.

Table 5 compares our solution with the standard 2PC and with a solution that uses group communication to improve performance during normal operation while paying a higher cost per topological change. Note again that our analysis considers **all** the traffic that the protocol generates, including the messages that the group communication layer uses in order to provide ordering guarantees and the messages needed to re-establish the underlying graph when a topological change occurs.

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). The 2PC/3PC based solutions pay a larger communication cost because they do not define an explicit efficient underlying communication structure. They also pay a heavy price in disk writes, in order to protect from crashes and topological changes.

To measure the overhead paid for a topology change we consider as one topology change 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 set of topological changes¹⁰. 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

¹⁰Note that this is a worst-case analysis. If only one failure occurs at a given moment, the communication overhead of our algorithm will be O(1) per edge per topological change.

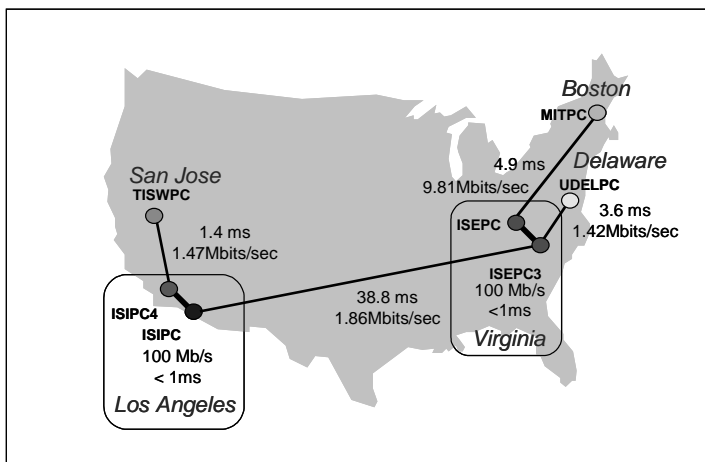


Figure 2: Wide Area Network Setup

unscalable. Membership learning means that all the nodes must know the identifier 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. The additional cost paid per transaction by the 2PC/3PC methods pays off in the presence of topological changes which are supported at no additional cost.

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

6 Performance Evaluation

We have implemented a prototype of the algorithm presented in this paper and we have evaluated its performance in both local and wide-area network settings and compared it with two existing solutions: an *upper-bound* version of two-phase commit (2PC) and a group communication based solution ([AT02]). The 2PC implementation that we used assumes that all the locks are granted instantly, thus ensuring the maximum level of concurrency the 2PC method can support. The group communication solution was implemented using the Spread group communication toolkit ([AS98]).

In the first set of experiments we compared the maximum throughput of the three solutions in a 100Mbit Ethernet LAN setup consisting of 14 PIII 667MHz replicas with 256MB RAM and 9GB SCSI disks running Linux. The second set of experiments was run on an emulated Wide-Area network setup consisting of 7 PIII 850MHz computers with 512MB RAM and 40G IDE hard drives, running Linux. The wide-area network setup was designed to emulate the CAIRN US-wide network depicted in Figure 2. Each client submits one action at a time. Once the action is committed the client receives an acknowledgement from the server it was connected to and can generate a new action.

The left-side graph in Figure 3 compares the throughput of the three algorithms on the local area network. The upper-bound 2PC algorithm can only sustain a combined throughput of 60 actions per second. Our modular solution reaches an aggregate throughput comparable to the one reached by the group communication approach (800 and 1000 actions per second respectively), proving the similarity of the two approaches and their efficiency.

The wide-area experiments show a similar behaviour. The 2PC algorithm quickly reaches its limit of about 80 actions per second while the other algorithms are able to reach 350 actions per second.

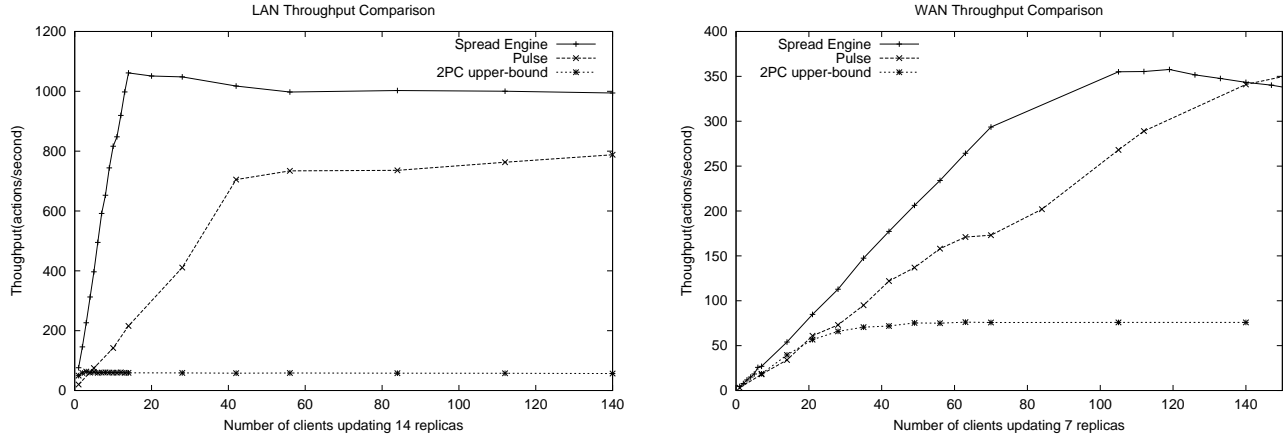


Figure 3: Aggregate Throughput Comparison

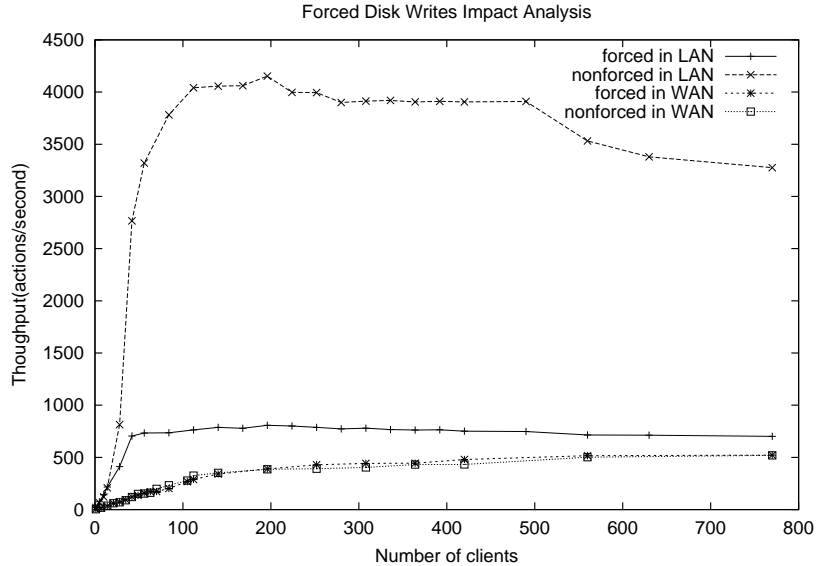


Figure 4: Impact of Forced Disk Writes on Aggregate Throughput

We have ran more experiments with the pulse-based algorithm and we reached its saturation point at approximately 500 actions per second when 560 clients push actions into the system, as can be noted in figure 4. These results were not plotted on the comparison graph since we did not have sufficient experimental data for the other algorithms.

In both experiments we notice that the pulse-based solution experiences a higher latency per transaction than the group communication solution. This is reflected in the slower increasing throughput curve - more clients are needed in order to produce the same aggregated throughput. However, in the wide-area network setup, the modular solution reaches a higher aggregated throughput.

We have also evaluated the performance of the pulse-based algorithm when we allow asynchronous disk-writes instead of the forced disk writes that are necessary to guarantee the algorithm correctness in the presence of faults (Figure 4). This evaluation shows the actual potential of the synchronization method and could be practically recreated in a system that employs specialized, fast persistent storage as used in critical high-performance database systems. In these experiments the pulse algorithm reached its peak at over 4000 actions/second on LAN and 520 actions/second in the WAN setup.

The results illustrate how the performance bottleneck shifts from the disk operations, in the local area setup, to the available bandwidth in the wide-area experiments. In wide-area settings it becomes increasingly important to use an optimized communication pattern.

7 Conclusions

We have presented a consistent distributed commit algorithm designed by combining a number of simple, well-defined modules. The algorithm functions correctly in the presence of any combination of network/server faults. The modular approach allows for an easier understanding of the solution as well as proving its correctness. We analytically compare the solution with existing methods and we validate the analysis through practical experiments.

We show that simple design can yield an efficient solution to a problem that traditionally required very complex algorithmic solutions. The modular design can be used for a future detailed analysis of the trade-offs involved and finer tuning to the specific network environments in which the solution is deployed.

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, Israel, 1995.
- [AS98] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS 98-4, Johns Hopkins University, Center for Networking and Distributed Systems, 1998.
- [AT02] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.
- [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.
- [CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
- [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. *ACM Trans. Programming Language Syst.*, 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 Lamport. 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, Massachusetts 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 Committee 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  $\leftarrow$  self
  send ID(self) to all neighbours
when: receive ID(id) from n do
  if leader < id then
    leader  $\leftarrow$  id
    parent  $\leftarrow$  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  $\neq$  NIL) then
      send ACK(1) to parent
    else
      send tree_change() notification to Reconciliation Algorithm
```

Algorithm 6 Reset Procedure

```
when: receive network_fail() from failure detector do
  changeNo  $\leftarrow$  changeNo + 1
  execute BroadcastChange()
when: receive change(change) do
  if (change > changeNo) then
    changeNo  $\leftarrow$  change
when: invoked BroadcastChange() do
  send change(changeNo) to all neighbours
  send network_change(changeNo)
```

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 7 Quorum Procedure (part of Reconciliation Algorithm)

```
1: current_weight  $\leftarrow w_i$ 
2: if (children_list = NIL) then
3:   send message weight( $w_i$ ) to parent
4: when: receive weight( $w$ ) from child do
5:   current_weight  $\leftarrow$  current_weight +  $w$ 
6:   if received weight message from all children then
7:     if (parent  $\neq$  nil) then
8:       send weight(current_weight) to parent
9:     else if (current_weight >  $\frac{qweight}{2}$ ) then
10:      return TRUE
11:   else
12:     return FALSE
```
