Formalising Reconciliation in Partitionable Networks with Distributed Services

Mikael Asplund and Simin Nadjm-Tehrani

Department of Computer and Information Science,
Linköping University SE-581 83 Linköping, Sweden
{mikas,simin}@ida.liu.se

1 Introduction

Modern command and control systems are characterised by computing services provided to several actors at different geographical locations. The actors operate on a common state that is modularly updated at distributed nodes using local data services and global integrity constraints for validity of data in the value and time domains. Dependability in such networked applications is measured through availability of the distributed services as well as the correctness of the state updates that should satisfy integrity constraints at all times. Providing support in middleware is seen as one way of achieving a high level of service availability and well-defined performance guarantees. However, most recent works [1, 2] that address fault-aware middleware cover crash faults and provision of timely services, and assume network connectivity as a basic tenet.

In this paper we study the provision of services in distributed object systems, with network partitions as the primary fault model. The problem appears in a variety of scenarios [3], including distributed flight control systems. The scenarios combine provision of critical services with data-intensive operations. Clients can approach any node in the system to update a given object, copies of which are present across different nodes in the system. A correct update of the object state is dependent on validity of integrity constraints, potentially involving other distributed objects. Replicated objects provide efficient access at distributed nodes (leading to lower service latency). Middleware is employed for systematic upholding of common view on the object states and consistency in write operations. However, problems arise if the network partitions. That is, if there are broken/overloaded links such that some nodes become unreachable, and the nodes in the network form disjoint partitions. Then, if services are delivered to clients approaching different partitions, the upholding of consistency has to be considered explicitly. Moreover, there should be mechanisms to deal with system mode changes, with service differentiation during degraded mode.

Current solutions to this problem typically uphold full consistency at the cost of availability. When the network is partitioned, the services that require integrity constraints over objects that are no longer reachable are suspended until the network is physically unified. Alternatively, a majority partition is assumed to continue delivering services based on the latest replica states. When the network is reunified the minority partition(s) nodes rejoin; but during the partition clients approaching the minority partition receive no service. The goal of our work is to investigate middleware support that enables distributed services to be provided at *all* partitions, at the expense of temporarily trading off some consistency. To gain higher availability we need to act optimistically, and allow one primary per partition to provisionally service clients that invoke operations in that partition.

The contributions of the paper are twofold. First, we present a protocol that after reunification of a network partition takes a number of partition states and generates a new partition state that includes a unique state per object. In parallel with creating this new state the protocol continues servicing incoming requests. Since the state of the (reconciled) post-reunification objects are not yet finalised, the protocol has to maintain virtual partitions until all operations that have arrived after the partition fault and provisionally serviced are dealt with.

Second, we show that the protocol results in a stable partition state, from which onwards the need for virtual partitions is no longer necessary. The proof states the assumptions under which the stable state is reached. Intuitively, the system will leave the reconciliation mode when the rate of incoming requests is lower than the rate of handling the provisionally accepted operations during reconciliation. The resulting partition state is further shown to have desired properties. A notion of correctness is introduced that builds on satisfaction of integrity constraints as well as respecting an intended order of performed operations seen from clients' point of view.

The structure of the paper is as follows. Section 2 i provides an informal overview of the formalised protocols in the paper. Section 3 introduces the basic formal notions that are used in the models. Section 4 describes the intuitive reasoning behind the choice of ordering that is imposed on the performed operations in the system and relates the application (client) expectations to the support that can reasonably be provided by automatic mechanisms in middleware. Section 5 presents the reconciliation protocol in terms of distributed algorithms running at replicas and in a reconciliation manager. Section 6 is devoted to the proofs of termination and correctness for the protocol. Related work are described in Sect. 7, and Sect. 8 concludes the paper.

2 Overview

We begin by assuming that middleware services for replication of objects are in place. This implies that the middleware has mechanisms for creating replica objects, and protocols that propagate a write operation at a primary copy to all the object replicas transparently. Moreover, the mechanisms for detecting link failures and partition faults are present in the middleware. The latter is typically implemented by maintaining a membership service that keeps an up to date view of which replicas for an object are running and reachable. The middleware also includes naming/location services, whereby the physical node can be identified given a logical address.

In normal mode, the system services read operations in a distributed manner; but for write operations there are protocols that check integrity constraints before propagating the update to all copies of the object at remote nodes. Both in normal and degraded mode, each partition is assumed to include a designated primary replica for each object in the system.

The integrity constraints in the system are assumed to fall in two classes: critical and non-critical. For operations with non-critical constraints different primary servers continue to service client requests, and provisionally accept the operations that satisfy integrity constraints. When the partition fault is repaired, the state of the main partition is formed by reconciling the operations carried out in the earlier disjoint partitions. The middleware supports this reconciliation process and guarantees the consistency of the new partition state. The state is formed by replaying some provisional operations that are accepted, and rejecting some provisional operations that should be notified to clients as "undone". It is obviously desirable to keep as many of the provisionally accepted operations as possible.

The goal of the paper is to formally define mechanisms that support the above continuous service in presence of (multiple) partitions, and satisfactorily create a new partition upon recovery from the fault. For a system that has a considerable portion of its integrity constraints classified as non-critical this should intuitively increase availability despite partitions. Also, the average latency for servicing clients should decrease as some client requests that would otherwise be suspended or considerably delayed if the system were to halt upon partitions are now serviced in a degraded mode.

Figure 1 presents the modes of a system in presence of partition faults. The system is available in degraded mode except for operations for which the integrity constraints are critical so that they cannot accept the risk of being inconsistent during partitions (these are not performed at all in degraded mode). The system is also partially available during reconciling mode; but there is a last short stage within reconciliation (installing state) during which the system is unavailable.

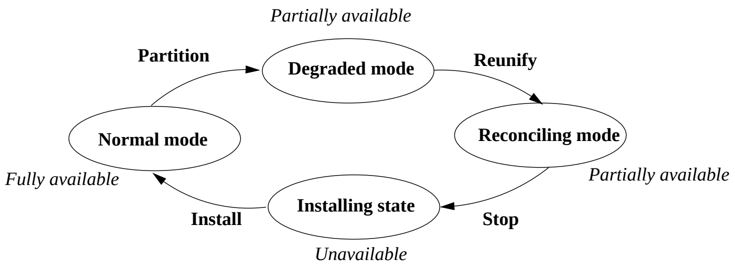

Fig. 1. System modes

In earlier work we have formalised the reconciliation process in a simple model and experimentally studied three reconciliation algorithms in terms of their influence on service outage duration [4]. A major assumption in that work was that no service was provided during the *whole* reconciliation process. Simulations showed that the drawback of the 'non-availability' assumption can be severe in some scenarios; namely the time taken to reconcile could be long enough so that the non-availability of services during this interval would be almost as bad as having no degraded service at all (thereby no gain in overall availability).

In this paper we investigate the implications of continued service delivery *during* the reconciliation process. This implies that we need to formalise a more refined protocol that keeps providing service to clients in parallel with reconciliation (and potential replaying of some operations). The algorithms are modelled in timed I/O automata, that naturally model multiple partition faults occurring in a sequence (so called cascading effects). More specifically, the fault model allows multiple partition faults in a sequence before a network is reunified, but no partitions occur during reconciliation. We also exclude crash faults during reconciliation in order to keep the models and proofs easier to convey. Crash faults can be accommodated using existing checkpointing approaches [5] with no known effects on main results of the paper. Furthermore, we investigate correctness and termination properties of this more refined reconciliation protocol. The proofs use admissible timed traces of timed I/O automata.

3 Preliminaries

This section introduces the concepts needed to describe the reconciliation protocol and its properties. We will define the necessary terms such as object, partition and replica as well as defining consistency criteria for partitions.

3.1 Objects

For the purpose of formalisation we associate data with objects. Implementation-wise, data can be maintained in databases and accessed via database managers.

Definition 1. *An* object *o is a triple $o = (S, \mathcal{O}, T)$ where S is the set of possible states, $\mathcal{O}$ is the set of operations that can be applied to the object state and $T \subseteq S \times \mathcal{O} \times S$ is a transition relation on states and operations.*

We assume all operation sets to be disjunct so that every operation is associated with one object.

Transitions from a state s to a state s' will be denoted by $s \overset{\alpha}{\rightsquigarrow} s'$ where $\alpha = \langle op, k \rangle$ is an operation instance with $op \in \mathcal{O}$, and $k \in \mathbb{N}$ denotes the unique invocation of operation op at some client.

Definition 2. *An* integrity constraint *c is a predicate over multiple object states. Thus, $c \subseteq S_1 \times S_2 \times \ldots \times S_n$ where n is the number of objects in the system.*

Intuitively, object operations should only be performed if they do not violate integrity constraints.

A distributed system with replication has multiple replicas for every object located on different nodes in the network. As long as no failures occur, the existence of replicas has no effect on the functional behaviour of the system. Therefore, the state of the system in the normal mode can be modelled as a set of replicas, one for each object.

Definition 3. *A* replica *r for object $o = (S, \mathcal{O}, T)$ is a triple $r = (L, s^0, s^m)$ where the log $L = \langle \alpha_1 \ldots \alpha_m \rangle$ is a sequence of operation instances defined over $\mathcal{O}$. The initial state is $s^0 \in S$ and $s^m \in S$ is a state such that $s^0 \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_m}{\rightsquigarrow} s^m$.*

The log can be considered as the record of operations since the last checkpoint that also recorded the (initial) state s^0.

We consider partitions that have been operating independently and we assume the nodes in each partition to agree on one primary replica for each object. This will typically be promoted by the middleware. Moreover, we assume that all objects are replicated across all nodes. For the purpose of reconciliation the important aspect of a partition is not how the actual nodes in the network are connected but the replicas whose states have been updated separately and need to be reconciled. Thus, the state of each partition can be modelled as a set of replicas where each object is uniquely represented.

Definition 4. *A* partition *p is a set of replicas r such that if $r_i, r_j \in p$ are both replicas for object o then $r_i = r_j$.*

The state of a partition $p = \{(L_1, s_1^0, s_1), \ldots, (L_n, s_n^0, s_n)\}$ consists of the state of the replicas $\langle s_1, \ldots, s_n \rangle$. Transitions over object states can now be naturally extended to transitions over partition states.

Definition 5. *Let $\alpha = \langle op, k \rangle$ be an operation instance for some invocation k of operation op. Then $\mathbf{s^j} \overset{\alpha}{\rightsquigarrow} \mathbf{s^{j+1}}$ is a* partition transition *iff there is an object o_i such that $s_i \overset{\alpha}{\rightsquigarrow} s_i'$ is a transition for o_i, $\mathbf{s^j} = \langle s_1, \ldots, s_i, \ldots, s_n \rangle$ and $\mathbf{s^{j+1}} = \langle s_1, \ldots, s_i', \ldots, s_n \rangle$.*

We denote by $\mathit{Apply}(\alpha, P)$ the result of applying operation instance α at some replica in partition P, giving a new partition state and a new log for the affected replica.

3.2 Order

So far we have not introduced any concept of order except that a state is always the result of operations performed in some order. When we later will consider the problem of creating new states from operations that have been performed in different partitions we must be able to determine in what (if any) order the operations must be replayed.

At this point we will merely define the existence of a strict partial order relation over operation instances. Later, in Sect. 4.2 we explain the philosophy behind choosing this relation.

Definition 6. *The relation $\rightarrow$ is a irreflexive, transitive relation over the operation instances obtained from operations $\mathcal{O}_1 \cup \ldots \cup \mathcal{O}_n$.*

In Definition 8 we will use this ordering to define correctness of a partition state. Note that the ordering relation induces an ordering on states along the time line whereas the consistency constraints relate the states of various objects at a given "time point" (a cut of the distributed system).

3.3 Consistency

Our reconciliation protocol will take a set of partitions and produce a new partition. As there are integrity constraints on the system state and order dependencies on operations, a reconciliation protocol must make sure that the resulting partition is correct with respect to both of these requirements. This section defines consistency properties for partitions.

Definition 7. *A partition state $\mathbf{s} = \langle s_1, \ldots, s_n \rangle$ for partition where $P = \{(L_1, s_1^0, s_1), \ldots, (L_n, s_n^0, s_n)\}$ is* constraint consistent*, denoted cc(P), iff for all integrity constraints c it holds that $\mathbf{s} \in c$.*

Next we define a consistency criterion for partitions that also takes into account the order requirements on operations in logs. Intuitively we require that there is some way to construct the current partition state from the initial state using all the operations in the logs. Moreover, all the intermediate states should be constraint consistent and the operation ordering must follow the ordering restrictions. We will use this correctness criterion in evaluation of our reconciliation protocol.

Definition 8. *Let $P = \{(L_1, s_1^0, s_1), \ldots, (L_n, s_n^0, s_n)\}$ be a partition, and let $\mathbf{s^k}$ be the partition state. The initial partition state is $\mathbf{s^0} = \langle s_1^0, \ldots s_n^0 \rangle$. We say that the partition P is* consistent *if there exists a sequence of operation instances $L = \langle \alpha_1, \ldots, \alpha_k \rangle$ such that:*

1. $\alpha \in L_i \Rightarrow \alpha \in L$
2. $\mathbf{s^0} \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_k}{\rightsquigarrow} \mathbf{s^k}$
3. *Every $\mathbf{s^j} \in \{\mathbf{s^0}, \ldots, \mathbf{s^k}\}$ is constraint consistent*
4. $\alpha_i \rightarrow \alpha_j \Rightarrow i < j$

4 Application-Middleware Dependencies

In Sect. 3 we introduced integrity constraints and an order relation between operations. These concepts are used to ensure that the execution of operations is performed according to the clients' expectations. In this section we will further elaborate on these two concepts, and briefly explain why they are important for reconciliation.

Due to the fact that the system continues to provisionally serve requests in degraded mode, the middleware has to start a reconciliation process when the system recovers from link failures (i.e. when the network is physically reunified). At that point in time there may be several conflicting states for each object since write requests have been serviced in all partitions. In order to merge these states into one common state for the system we will have to replay the performed operations (that are stored in the logs of each replica). Some operations may not satisfy integrity constraints when multiple partitions are considered, and they may have to be rejected (seen from a client perspective, undone). The replay starts from the last common state (i.e. from before the partition fault occurred) and iteratively builds up a new state. Note that the replay of an operation instance may potentially take place in a different state compared to that where the operation was originally applied in the degraded mode.

4.1 Integrity Constraints

Since some operations will have to be replayed we need to consider the conditions required, so that replaying an operation in a different state than that it was originally executed in does not cause any discrepancies. We assume that such conditions are indeed captured by integrity constraints.

In other words, the middleware expects that an application writer has created the needed integrity constraints such that replaying an operation during reconciliation is harmless as long as the constraint is satisfied, even if the state on which it is replayed is different from the state in which it was first executed. That is, there should not be any implicit conditions that are checked by the client at the invocation of the operation. In such a case it would not be possible for the middleware to recheck these constraints upon reconciliation.

As an example, consider withdrawal from a credit account. It is acceptable to allow a withdrawal as long as there is coverage for the account in the balance; it is not essential that the balance should be a given value when withdrawal is allowed. Recall that that an operation for which a later rejection is not acceptable from an application point of view should be associated with a critical constraint (thereby not applied during a partition at all). An example of such an operation would be the termination of a credit account.

4.2 Expected Order

To explain the notion of expected order we will first consider a system in normal mode and see what kind of execution order is expected by the client. Then we will require the same expected order to be guaranteed by the system when performing reconciliation. In our scenarios we will assume that a client who invokes two operations α and β in sequence without receiving a reply between them does not have any ordering requirements on the invocations. Then the system need not guarantee that the operations are executed in any particular order. This is true even if the operations were invoked on the same object.

Now assume that the client first invokes α and does not invoke β until it has received a reply for α confirming that α has been executed. Then the client knows that α is executed before β. The client process therefore assumes an ordering between the execution of α and β due to the fact that the events of receiving a reply for α precedes the event of invoking β. This is the order that we want to capture with the relation $\rightarrow$ from Definition 6. When the reconciliation process replays the operations it must make sure that this expected order is respected.

This induced order need not be specified at the application level. It can be captured by a client side front end within the middleware, and reflected in a tag for the invoked operations. Thus, every operation is piggybacked with information about what other operations must precede it when it is later replayed. This information is derived from the requests that are sent by the client and the received replies. Note that it is only necessary to attach the IDs of the immediate predecessors so the overhead will be small.

5 The Reconciliation Protocol

In this section we will describe the reconciliation protocol in detail using timed I/O automata. However, before going into details we provide a short overview of the idea behind the protocol. The protocol is composed of two types of processes: a number of replicas and one reconciliation manager.

The replicas are responsible for accepting invocations from clients and sending logs to the reconciliation manager during reconciliation. The reconciliation manager is responsible for merging replica logs that are sent during reconciling mode. It is activated when the system is reunified and eventually sends an install message with the new partition state to all replicas. The new partition state includes empty logs for each replica.

The reconciliation protocol starts with one state per partition is faced with the task of merging a number of operations that have been performed in different partitions while preserving constraint consistency and respecting the expected ordering of operations. In parallel with this process the protocol should take care of operations that arrive during the reconciliation phase. Note that there may be unreconciled operations in the logs that should be executed before the incoming operations that arrive during reconciliation.

The state that is being constructed in the reconciliation manager may not yet reflect all the operations that are before ($\rightarrow$) the incoming operations. Therefore the only state in which the incoming operation can be applied to is one of the partition states from the degraded mode. Or in other words, we need to execute the new operations as if the system was still in degraded mode. In order to do this we will maintain virtual partitions while the reconciliation phase lasts.

5.1 Reconciliation Manager

In Algorithm 1 the variable *mode* represents the modes of the reconciliation process and is basically the same as the system modes described in Fig. 1 except that the normal and degraded mode are collapsed into an idle mode for the reconciliation manager, which is its initial mode of operation.

When a reunify action is activated the reconciliation manager goes to reconciling mode. Moreover, the variable P, which represents the partition state, is initialised with the pre-partition state, and the variable *opset* that will contain all the operations to replay is set to empty. Now the reconciliation process starts waiting for the replicas to send their logs and the variable *awaitedLogs* is set to contain all replicas that have not yet sent their logs.

Next, we consider the action $receive(\langle \text{“}log'', L\rangle)_{iM}$ which will be activated when some replica r_i sends its operation log. This action will add logged operations to *opset* and to $ackset[i]$ where the latter is used to store acknowledge messages that should be sent back to replica r_i. The acknowledge messages are sent by the action $send(\langle \text{“}logAck'', ackset[i]\rangle)_{Mi}$. When logs have been received from all replicas (i.e. *awaitedLogs* is empty) then the manager can proceed and start replaying operations. A deadline will be set on when the next handle action must be activated (this is done by setting $last(handle)$).

The action $handle(\alpha)$ is an internal action of the reconciliation process that will replay the operation α (which is minimal according to $\rightarrow$ in *opset*) in the reconciled state that is being constructed. The operation is applied if it results in a constraint consistent state.

As we will show in Sect. 6.2, there will eventually be a time when *opset* is empty at which M will enable $broadcast(\text{“}stop'')_M$. This will tell all replicas to stop accepting new invocations. Moreover, M will set the mode to *installingState* and wait for all replicas to acknowledge the stop message. This is done to guarantee that no messages remain untreated in the reconciliation process. Finally, when the manager has received acknowledgements from all replicas it will broadcast an install message with the reconciled partition state and enter idle mode.

5.2 Replica Process

A replica process (see Algorithm 2) is responsible for receiving invocations to clients and for sending logs to M. We will proceed by describing the states and actions of a replica process. First note that a replica process can be in four different modes, normal, degraded, reconciling, and unavailable which correspond to the system modes of Fig. 1.

In this paper we do not explicitly model how updates are replicated from primary replicas to secondary replicas. Instead, we introduce two global shared variables that are accessed by all replicas, provided that they are part of the same group. The first shared variable $P[i]$ represents the partition for the group with ID i and it is used by all replicas in that group during normal and degraded mode. The group ID is assumed to be delivered by the membership service.

During reconciling mode the group-ID will be 1 for all replicas since there is only one partition during reconciling mode. However, as we explained in the beginning of Sect. 5 the replicas must maintain virtual partitions to service requests during reconciliation. The shared variable $VP[j]$ is used to represent the virtual partition for group j which is based on the partition that was used during degraded mode.

Algorithm 1 Reconciliation manager M

```
States
$mode \in \{idle, reconciling, installingState\} \leftarrow idle$
$P \leftarrow \{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_n^0, s_n^0)\}$ /* Output of protocol: Constructed partition */
$opset$ /* Set of operations to reconcile */
$awaitedLogs$ /* Replicas to wait for sending a first log message */
$stopAcks$ /* Number of received stop "acks"*/
$ackset[i] \leftarrow \emptyset$ /* Log items from replica $i$ to acknowledge*/
$now \in \mathbb{R}^{0+}$
$last(handle) \leftarrow \infty$ /* Deadline for executing handle */
$last(stop) \leftarrow \infty$ /* Deadline for sending stop */
$last(install) \leftarrow \infty$ /* Deadline for sending install */

Actions
Input  $reunify(g)_M$
Eff:   $mode \leftarrow reconciling$
       $P \leftarrow \{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_n^0, s_n^0)\}$
       $opset \leftarrow \emptyset$
       $awaitedLogs \leftarrow \{$All replicas$\}$

Input  $receive(\langle\text{"log"}, L\rangle)_{iM}$
Eff:   $opset \leftarrow opset \cup L$
       $ackset[i] \leftarrow ackset[i] \cup L$
       if $awaitedLogs \neq \emptyset$
          $awaitedLogs \leftarrow awaitedLogs \setminus \{i\}$
       else
          $last(handle) \leftarrow$
             $min(last(handle), now + d_{\text{han}})$

Output  $send(\langle\text{"logAck"}, ackset[i]\rangle)_{Mi}$
Eff:   $ackset[i] \leftarrow \emptyset$

Internal  $handle(\alpha)$
Pre:   $awaitedLogs = \emptyset$
       $mode = reconciling$
       $\alpha \in opset$
       $\nexists \beta \in opset \quad \beta \rightarrow \alpha$
Eff:   if $cc(Apply(\alpha, P))$
          $P \leftarrow Apply(\alpha, P)$
       $last(handle) \leftarrow now + d_{\text{han}}$
       $opset \leftarrow opset \setminus \{\alpha\}$
       if $opset = \emptyset$
          $last(stop) = now + d_{\text{act}}$

Output  $broadcast(\text{"stop"})_M$
Pre:   $opset = \emptyset$
       $awaitedLogs = \emptyset$
Eff:   $stopAcks \leftarrow 0$
       $mode \leftarrow installingState$
       $last(handle) \leftarrow \infty$
       $last(stop) \leftarrow \infty$

Input  $receive(\langle\text{"stopAck"}\rangle)_{iM}$
Eff:   $stopAcks \leftarrow stopAcks + 1$
       if $stopAck = mn$
          $last(install) = now + d_{\text{act}}$

Output  $broadcast(\langle\text{"install"}, P\rangle)_M$
Pre:   $mode = installingState$
       $stopAcks = m \cdot n$
Eff:   $mode \leftarrow idle$
       $last(install) = \infty$

Timepassage  $v(t)$
Pre:   $now + t \leq last(handle)$
       $now + t \leq last(stop)$
       $now + t \leq last(install)$
Eff:   $now \leftarrow now + t$
```

During normal mode replicas apply operations that are invoked through the $receive(\langle\text{“invoke”}, \alpha\rangle)_{cr}$ action if they result in a constraint consistent partition. A set *toReply* is increased with every applied operation that should be replied to by the action $send(\langle“reply'', \alpha\rangle)_{rc}$.

A replica leaves normal mode and enters degraded mode when the group membership service sends a partition message with a new group-ID. The replica will then copy the contents of the previous partition representation to one that will be used during degraded mode. Implicit in this assignment is the determination of one primary per partition for each object in the system (as provided by a combined name service and group membership service). The replica will continue accepting invocations and replying to them during degraded mode.

When a replica receives a reunify message it will take the log of operations served during degraded mode (the set L) and send it to the reconciliation manager M by the action $send(\langle“log'', L\rangle)_{rM}$. In addition, the replica will enter reconciling mode and copy the partition representation to a virtual partition representation. The latter will be indexed using virtual group-ID vg which will be the same as the group-ID used during degraded mode. Finally, a deadline will be set for sending the logs to M.

The replica will continue to accept invocations during reconciliation mode with some differences in handling. First of all, the operations are applied to a virtual partition state. Secondly, a log message containing an applied operation is immediately scheduled to be sent to M. Finally, the replica will not immediately reply to the operations. Instead it will wait until the log message has been acknowledged by the reconciliation manager and $receive(\langle“logAck'', L\rangle)_{Mr}$ is activated. Now any operation whose reply was pending and for whom a $logAck$ has been received can be replied to (added to the set *toReply*).

At some point the manager M will send a stop message which will make the replica to go into unavailable mode and send a *stopAck* message. During this mode no invocations will be accepted until an install message is received. Upon receiving such a message the replica will install the new partition representation and once again go into normal mode.

6 Properties of the Protocol

The goal of the protocol is to restore consistency in the system. This is achieved by merging the results from several different partitions into one partition state. The clients have no control over the reconciliation process and in order to guarantee that the final result does not violate the expectations of the clients we need to assert correctness properties of the protocol. Moreover, as there is a growing set of unreconciled operations we need to show that the protocol does not get stuck in reconciliation mode for ever.

In this section we will show that (1) the protocol terminates in the sense that the reconciliation mode eventually ends and the system proceeds to normal mode (2) the resulting partition state which is installed in the system is consistent in the sense of Definition 8.

Algorithm 2 Replica r

```
Shared vars
$P[i] \leftarrow \{(\langle\rangle, s_1^0, s_1^0), \dots, (\langle\rangle, s_n^0, s_n^0)\}$, for $i = 1 \dots N$ /* Representation for partition $i$,
                                                     before reunification */
$VP[i]$, for $i = 1 \dots N$ /* Representation for virtual partition $i$, after reunification */

States
$mode \in \{normal, degraded, reconciling, unavailable\} \leftarrow idle$
$g \in \{1 \dots N\} \leftarrow 1$ /* Group identity (supplied by group membership service) */
$vg \in \{1 \dots N\} \leftarrow 1$ /* Virtual group identity, used between reunification and install */
$L \leftarrow \emptyset$ /* Set of log messages to send to reconciliation manager M*/
$toReply \leftarrow \emptyset$ /* Set of operations to reply to */
$pending \leftarrow \emptyset$ /* Set of operations to reply to when logged */
$enableStopAck$ /* Boolean to signal that a stopAck should be sent */
$last(log) \leftarrow \infty$ /* Deadline for next $send(\langle``log'', \dots\rangle)$ action */
$last(stopAck) \leftarrow \infty$ /* Deadline for next $send(\langle``stopAck'', \dots\rangle)$ action */
$now \in \mathbb{R}^{0+}$

Actions
Input $partition(g')_r$
Eff:  $mode \leftarrow degraded$
      $P[g'] \leftarrow P[g]$
      $g \leftarrow g'$

Input $reunify(g')_r$
Eff:  $L \leftarrow L_r$ where $\langle L_r, s_r^0, s_r\rangle \in P$
      $mode \leftarrow reconciling$
      $vg \leftarrow g$
      $VP[vg] \leftarrow P[g]$
      $g \leftarrow g'$
      $last(log) \leftarrow now + d_{\text{act}}$

Input $receive(\langle``\text{invoke}", \alpha\rangle)_{cr}$
Eff:  $switch(mode)$
      $normal \mid degraded \Rightarrow$
         if $Apply(\alpha, P[g])$ is Consistent)
            $P[g] \leftarrow Apply(\alpha, P[g])$
            $toReply \leftarrow toReply \cup \{\langle\alpha, c\rangle\}$
      $reconciling \Rightarrow$
         if $Apply(\alpha, VP[vg])$ is Consistent)
            $VP[vg] \leftarrow Apply(\alpha, VP[vg])$
            $L \leftarrow L \cup \{\alpha\}$
            $last(log) \leftarrow min(last(log), now + d_{\text{act}})$
            $pending \leftarrow pending \cup \{\langle\alpha, c\rangle\}$

Output $send(\langle``log'', L\rangle)_{rM}$
Pre:  $mode \in \{reconciling, unavailable\}$
      $L \neq \emptyset$
Eff:  $L \leftarrow \emptyset$
      $last(log) \leftarrow \infty$

Input $receive(\langle``logAck'', L\rangle)_{Mr}$
Eff:  $replies \leftarrow \{\langle\alpha, c\rangle \in pending \mid \alpha \in L\}$
      $toReply \leftarrow toReply \cup replies$
      $pending \leftarrow pending \setminus replies$

Output $send(\langle``reply'', \alpha\rangle)_{rc}$
Pre:  $\langle\alpha, c\rangle \in toReply$
Eff:  $toReply \leftarrow toReply \setminus \{\langle\alpha, c\rangle\}$

Input $receive(``stop'')_{Mr}$
Eff:  $mode \leftarrow unavailable$
      $enableStopAck \leftarrow true$
      $last(stopAck) \leftarrow now + d_{\text{act}}$

Output $send(\langle``stopAck''\rangle)_{rM}$
Pre:  $enableStopAck = true$
      $L = \emptyset$
Eff:  $enableStopAck = false$
      $last(stopAck) \leftarrow \infty$

Input $receive(\langle``install'', P'\rangle)_{Mr}$
Eff:  $P[g] \leftarrow P'$ /* g = 1 */
      $mode \leftarrow normal$

Timepassage $v(t)$
Pre:  $now + t \leq last(log)$
      $now + t \leq last(stopAck)$
Eff:  $now \leftarrow now + t$
```

6.1 Assumptions

The results rely on a number of assumptions on the system. We assume a partially synchronous system with reliable broadcast. Moreover, we assume that there are bounds on duration and rate of partition faults in the network. Finally we need to assume some restrictions on the behaviour of the clients such as the speed at which invocations are done and the expected order of operations. The rest of the section describes these assumptions in more detail.

Network Assumptions. We assume that there are two time bounds on the appearance of faults in the network. T_{D} is the maximal time that the network can be partitioned. T_{F} is needed to capture the minimum time between two faults. The relationship between these bounds are important as operations are piled up during the degraded mode and the reconciliation has to be able to handle them during the time before the next fault occurs.

We will not explicitly describe all the actions of the network but we will give a description of the required actions as well as a list of requirements that the network must meet. The network assumptions are summarised in N1-N6, where N1, N2, and N3 characterise reliable broadcast which can be supplied by a system such as Spread[6]. Assumption N4 relates to partial synchrony which is a basic assumption for fault-tolerant distributed systems. Finally we assume that faults are limited in frequency and duration (N5,N6) which is reasonable, as otherwise the system could never heal itself.

N1 A receive action is preceded by a send (or broadcast) action.
N2 A sent message is not lost unless a partition occurs.
N3 A sent broadcast message is either received by all in the group or a partition occurs and no process receives it.
N4 Messages arrive within a delay of d_{msg} (including broadcast messages).
N5 After a reunification, a partition occurs after an interval of at least T_{F}.
N6 Partitions do not last for more than T_{D}.

Client Assumptions. In order to prove termination and correctness of the reconciliation protocol we need some restrictions on the behaviour of clients.

C1 The minimum time between two invoke actions from one client is d_{inv}.
C2 If there is an application-specific ordering between two operations, then the first must have been replied to before the second was invoked. Formally, admissible timed system traces must be a subset of $ttraces(C2)$. $ttraces(C2)$ is defined as the set of sequences such that for all sequences σ in $ttraces(C2)$: $\alpha \rightarrow \beta$ and $(send(\langle \text{“}invoke'', \beta\rangle)_{cr}, t_1) \in \sigma \Rightarrow$ $\exists (receive(\langle \text{“}reply'', \alpha\rangle)_{r'c}, t_0) \in \sigma$ for some r' and $t_0 < t_1$.

In Table 1 we summarise all the system parameters relating to time intervals that we have introduced so far.

Table 1. Parameter summary

T_{F}	Minimal time before a partition fault after a reunify
T_{D}	Maximal duration of a partition
d_{msg}	Maximal message transmission time
d_{inv}	Minimal time between two invocations from one client
d_{han}	Maximal time between two handle actions within reconciliation manager
d_{act}	Deadline for actions

Server Assumptions. As we are concerned with reconciliation and do not want go into detail on other responsibilities of the servers or middleware (such as checkpointing), we will make two assumptions on the system behaviour that we do not explicitly model. First, in order to prove that the reconciliation phase ends with the installment of a consistent partition state, we need to assume that the state from which the reconciliation started is consistent. This is a reasonable assumption since normal and degraded mode operations always respect integrity constraints. Second, we assume that the replica logs are empty at the time when a partition occurs. This is required to limit the length of the reconciliation as we do not want to consider logs from the whole life time of a system. In practice, this has to be enforced by implementing checkpointing during normal operation.

A1 The initial state $\mathbf{s^0}$ is constraint consistent (see Definition 7).
A2 All replica logs are empty when a partition occurs.

We will now proceed to prove correctness of the protocol. First we give a termination proof and then a partial correctness proof.

6.2 Termination

In this section we will prove that the reconciliation protocol will terminate in the sense that after the network is physically healed (reunified) the reconciliation protocol eventually activates an install message to the replicas with the reconciled state. As stated in the theorem it is necessary that the system is able to replay operations at a higher rate than new operations arrive (reflected in the ratio q).

Theorem 1. *Let the system consist of the model of replicas, and the model of reconciliation manager. Assume the conditions described in Sect. 6.1. Assume further that the ratio q between the minimum handling rate $\frac{1}{d_{\mathrm{han}}}$ and the maximum interarrival rate for client invocations $C \cdot \frac{1}{d_{\mathrm{inv}}}$, where C is the maximum number of clients, is greater than one. Then, all admissible system traces are in the set $ttraces(Installing)$ of action sequences such that for every $(reunify(g)_M, t)$ there is a $(broadcast(\langle \text{“install”}, P\rangle)_M, t')$ in the sequence, with $t < t'$, provided that $T_{\mathrm{F}} > \frac{T_{\mathrm{D}}+7d}{q-1} + 9d$, where d exceeds d_{msg} and d_{act}.*

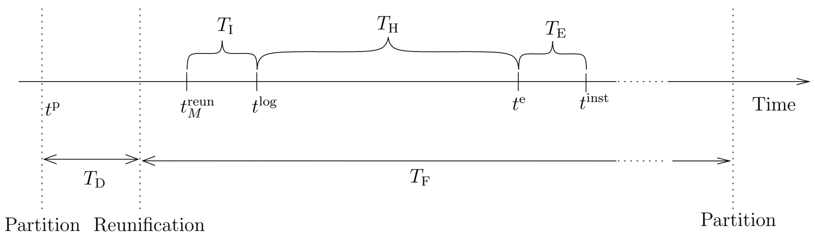

Fig. 2. Reconciliation timeline

Proof. Consider an arbitrary admissible timed trace γ such that $(reunify(g)_M, t_M^{\mathrm{reun}})$ appears in γ. Let all time points t^i below refer to points in γ. The goal of the proof is to show that there exists a point t^{inst} after t_M^{reun}, at which there is an install message appearing in γ.

The timing relation between two partitions and the time line for manager M can be visualised in Fig. 2 (see N5 and N6). Let t_i^{reun} denote the time point at which the reunification message arrives at process i. The reconciliation activity is performed over three intervals: initialising (T_{I}), handling (T_{H}), and ending (T_{E}). The proof strategy is to show that the reconciliation activity ends before the next partition occurs, considering that it takes one message transmission for the manager to learn about reunification. That is, $d_{\mathrm{msg}} + T_{\mathrm{I}} + T_{\mathrm{H}} + T_{\mathrm{E}} < T_{\mathrm{F}}$.

Let t^{log} be the last time point at which a log message containing a pre-reunification log is received from some replica. This is the time point at which handling (replaying) operations can begin. The handling interval (T_{H}) ends when the set of operations to replay (*opset*) is empty. Let this time point be denoted by t^e.

Initialising:

$$T_{\mathrm{I}} = t^{\mathrm{log}} - t_M^{\mathrm{reun}}$$

The latest estimate for t^{log} is obtained from the latest time point at which a replica may receive this reunification message (t_r^{reun}) plus the maximum time for it to react (d_{act}) plus the maximum transmission time (d_{msg}).

$$T_{\mathrm{I}} \leq \max_r(t_r^{\mathrm{reun}}) + d_{\mathrm{act}} + d_{\mathrm{msg}} - t_M^{\mathrm{reun}}$$

By N4 all reunification messages are received within d_{msg}.

$$T_{\mathrm{I}} \leq t_M^{\mathrm{reun}} + d_{\mathrm{msg}} + d_{\mathrm{act}} + d_{\mathrm{msg}} - t_M^{\mathrm{reun}} \leq 2d_{\mathrm{msg}} + d_{\mathrm{act}} \qquad (1)$$

Handling: The maximum handling time is characterised by the maximum number of invoked client requests times the maximum handling time for each operation (d_{han}, see Algorithm 1), times the maximum number of clients C. We divide client invocations in two categories, those that arrive at the reconciliation manager before t^{log} and those that arrive after t^{log}.

$$T_{\mathrm{H}} \leq \big([\text{pre-}t^{\mathrm{log}} \text{ messages}] + [\text{post-}t^{\mathrm{log}} \text{ messages}]\big) \cdot C \cdot d_{\mathrm{han}}$$

The maximum time that it takes for a client invocation to be logged at M is equal to $2d_{\mathrm{msg}} + d_{\mathrm{act}}$, consisting of the transmission time from client to replica and the transmission time from replica to manager as well as the reaction time for the replica. The worst estimate of the number of post-$t^{\log}$ messages includes all invocations that were initiated at a client prior to $t^{\log}$ and logged at M after $t^{\log}$. Thus the interval of $2d_{\mathrm{msg}} + d_{\mathrm{act}}$ must be added to the interval over which client invocations are counted.

$$T_{\mathrm{H}} \leq \left(\frac{T_{\mathrm{D}} + d_{\mathrm{msg}} + T_{\mathrm{I}}}{d_{\mathrm{inv}}} + \frac{T_{\mathrm{H}} + 2d_{\mathrm{msg}} + d_{\mathrm{act}}}{d_{\mathrm{inv}}} \right) \cdot C \cdot d_{\mathrm{han}} \tag{2}$$

using earlier constraint for T_{I} in (1). Finally, together with the assumption in the theorem we can simplify the expression as follows:

$$T_{\mathrm{H}} \leq \frac{T_{\mathrm{D}} + 5d_{\mathrm{msg}} + 2d_{\mathrm{act}}}{q-1} \tag{3}$$

Ending: According to the model of reconciliation manager M an empty *opset* results in the sending of a stop message within d_{act}. Upon receiving the message at every replica (within d_{msg}), the replica acknowledges the stop message within d_{act}. The the new partition can be installed as soon as all acknowledge messages are received (within d_{msg}) but at the latest within d_{act}. Hence T_{E} can be constrained as follows:

$$T_{\mathrm{E}} = t^{\mathrm{inst}} - t^{e} \leq 3d_{\mathrm{act}} + 2d_{\mathrm{msg}} \tag{4}$$

Final step: Now we need to show that $d_{\mathrm{msg}} + T_{\mathrm{I}} + T_{\mathrm{H}} + T_{\mathrm{E}}$ is less than T_{F} (time to next partition according to N5). From (1), (3), and (4) we have that:

$$T_{\mathrm{I}} + T_{\mathrm{H}} + T_{\mathrm{E}} \leq 2d_{\mathrm{msg}} + d_{\mathrm{act}} + \frac{T_{\mathrm{D}} + 5d_{\mathrm{msg}} + 2d_{\mathrm{act}}}{q-1} + 3d_{\mathrm{act}} + 2d_{\mathrm{msg}}$$

Given a bound d on delays d_{act} and d_{msg} we have:

$$d_{\mathrm{msg}} + T_{\mathrm{I}} + T_{\mathrm{H}} + T_{\mathrm{E}} \leq \frac{T_{\mathrm{D}} + 7d}{q-1} + 9d$$

Which concludes the proof according to theorem assumptions. □

6.3 Correctness

As mentioned in Sect. 3.3 the main requirement on the reconciliation protocol is to preserve consistency. The model of the replicas obviously keeps the partition state consistent (see the action under $receive(\langle \text{“}invoke'', \alpha \rangle)_{cr}$. The proof of correctness is therefore about the manager M withholding this consistency during reconciliation, and specially when replaying actions. Before we go on to the main theorem on correctness we present a theorem that shows the ordering requirements of the application (induced by client actions) are respected by our models.

Theorem 2. *Let the system consist of the model of replicas, and the model of reconciliation manager. Assume the conditions described in Sect. 6.1. Define the set* $ttraces(Order)$ *as the set of all action sequences with monotonically increasing times with the following property: for any sequence* $\sigma \in ttraces(Order)$*, if* $(handle((\alpha), t)$ *and* $(handle((\beta), t')$ *is in* σ*,* $\alpha \rightarrow \beta$*, and there is no* $(partition(g), t'')$ *between the two handle actions, then* $t < t'$*. All admissible timed traces of the system are in the set* $ttraces(Order)$*.*

Proof. We assume $\alpha \rightarrow \beta$, and take an arbitrary timed trace γ belonging to admissible timed traces of the system such that $(handle(\alpha), t)$ and $(handle(\beta), t')$ appear in γ and no partition occurs in between them. We are going to show that $t < t'$, thus γ belongs to $ttraces(Order)$. The proof strategy is to assume $t' < t$ and prove contradiction.

By the precondition of $(handle(\beta), t')$ we know that α cannot be in the *opset* at time t' (see the **Internal** action in M). Moreover, we know that α must be in *opset* at time t because $(handle(\alpha), t)$ requires it. Thus, α must be added to *opset* between these two time points and the only action that can add operations to this set is $receive(\langle \text{``}log'', \ldots\rangle)_{rM}$. Hence there is a time point $t^l$ at which $(receive(\langle \text{``}log'', \langle \ldots, \alpha, \ldots\rangle\rangle)_{rM}, t^l)$ appears in γ and

$$t' < t^l < t \tag{5}$$

Next consider a sequence of actions that must all be in γ with $t^0 < t^1 < \ldots < t_8 < t'$.

1. $(handle((\beta), t')$
2. $(receive(\langle \text{``}log'', \langle \ldots, \beta, \ldots\rangle\rangle, t_8)_{r_1 M}$ for some r_1
3. $(send(\langle \text{``}log'', \langle \ldots, \beta, \ldots\rangle\rangle, t_7)_{r_1 M}$
4. $(receive(\langle \text{``}invoke'', \beta\rangle, t_6)_{cr_1}$ for some c
5. $(send(\langle \text{``}invoke'', \beta\rangle, t_5)_{cr_1}$
6. $(receive(\langle \text{``}reply'', \alpha\rangle, t_4)_{cr_2}$ for some r_2
7. $(send(\langle \text{``}reply'', \alpha\rangle, t_3)_{r_2 c}$
8. $(receive(\langle \text{``}logAck'', \langle \ldots, \alpha, \ldots\rangle\rangle, t_2)_{Mr_2}$
9. $(send(\langle \text{``}logAck'', \langle \ldots, \alpha, \ldots\rangle\rangle, t_1)_{Mr_2}$
10. $(receive(\langle \text{``}log'', \langle \ldots, \alpha, \ldots\rangle\rangle, t^0)_{r_2 M}$

We show that the presence of each of these actions requires the presence of the next action in the list above (which is preceding in time).

– (1⇒2) is given by the fact that β must be in *opset* and that $(receive(\langle \text{``}log'', \langle \ldots, \beta, \ldots\rangle\rangle, t_8)_{r_1 M}$ is the only action that adds operations to *opset*.
– (2⇒3), (4⇒5), (6⇒7) and (8⇒9) are guaranteed by the network (N1).
– (3⇒4) is guaranteed since β being in $L = \langle \ldots, \beta, \ldots\rangle$ at r_1 implies that some earlier action has added β to L and $(receive(\langle \text{``}invoke'', \beta\rangle, t_6)_{cr_1}$ is the only action that adds elements to L at r_1.
– (5⇒6) is guaranteed by C3 together with the fact that $\alpha \rightarrow \beta$.

- (7⇒8) Due to 7 α must be in *toReply* at r_2 at time t^3. There are two actions that set *toReply*: one under the normal/degraded mode, and one upon receiving a *logAck* message from the manager M.
  First, we show that r_2 cannot be added to *toReply* as a result of $receive(\langle''invoke'', \alpha\rangle)_{cr_2}$ in normal mode. Since α is being replayed by the manager ($(handle(\alpha), t)$ appears in γ) then there must be a partition between applying α and replaying α. However, no operation that is applied in normal mode will reach the reconciliation process M as we have assumed (A2) that the replica logs are empty at the time of a partition. And since α belongs to *opset* in M at time t, it cannot have been applied during normal mode.
  Second, we show that r_2 cannot be added to *toReply* as a result of $receive(\langle''invoke'', \alpha\rangle)_{cr_2}$ in degraded mode. If α was added to *toReply* in degraded mode then the log in the partition to which r_2 belongs would be received by M shortly after reunification (that precedes handle operations). But we have earlier established that $\alpha \notin opset$ at t', and hence α cannot have been applied in degraded mode. Thus α is added to *toReply* as a result of a *logAck* action and (7⇒8).
- (9⇒10) is guaranteed since α must be in $ackset[r_2]$ and it can only be put there by $(receive(\langle``log'', \langle\ldots, \alpha, \ldots\rangle\rangle, t^0)_{r_2 M}$

We have in (5) established that the received log message that includes α appeared in γ at time point t^l, $t' < t^l$. This contradicts that $t^0 = t^l < t'$, and concludes the proof. □

Theorem 3. *Let the set ttraces(Correct) be the set of action sequences with monotonically increasing times such that if* $(broadcast(\langle``install'', P\rangle)_M, t^{\text{inst}})$ *is in the sequence, then P is consistent according to Definition 8. All admissible timed executions of the system are in the set ttraces(Correct).*

Proof. Consider an arbitrary element σ in the set of admissible timed system traces. We will show that σ is a member of the set $ttraces(Correct)$. The strategy of the proof is to analyse the subtraces of σ that correspond to actions of each component of the system. In particular, the sequence corresponding to actions in the reconciliation manager M will be of interest.

Let γ be the sequence that contains all actions of σ that are also actions of the reconciliation manager M ($\gamma = \sigma|M$). It is trivial that for all processes $C \neq M$ it holds that $\sigma|C \in ttraces(Correct)$ as there are no install messages broadcasted by any other process. Therefore, if we show that γ is a member of $ttraces(Correct)$ then σ will also be a member of $ttraces(Correct)$.

We will proceed to show that γ is a member of $ttraces(Correct)$ by performing induction on the number of actions in γ.

Base case: Let P be the partition state before the first action in γ. The model of the reconciliation manager M initialises P to $\{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_n^0, s_n^0)\}$. Therefore, requirements 1,2 and 4 of Definition 8 are vacuously true and 3 is given by A1.

Inductive step: Assume that the partition state resulting from action i in γ is consistent. We will then show that the partition state resulting from action $i+1$ in γ is consistent. It is clear that the model of the reconciliation manager M does not affect the partition state except when actions $reunify(g)_M$ and $handle(\alpha)$ are taken. Thus, no other actions need to be considered. We show that reunify and handle preserve consistency of the partition state.

The action $(reunify(g)_M, t)$ sets P to the initial value of P which has been shown to be consistent in the base case.

The action $(handle(\alpha), t)$ is the interesting action in terms of consistency for P. We will consider two cases based on whether applying α results in an inconsistent state or not. Let P^i be the partition state after action i has been taken.

(1) If $Apply(\alpha, P^i)$ is not constraint consistent then the if-statement in the action handle is false and the partition state will remain unchanged, and thus consistent after action $i+1$ according to the inductive assumption.

(2) If $Apply(\alpha, P^i)$ is constraint consistent then the partition state P^{i+1} will be set to $Apply(\alpha, P^i)$. By the inductive assumption there exists a sequence L leading to P^i. We will show that the sequence $L' = L + \langle\alpha\rangle$ satisfies the requirements for P^{i+1} to be consistent.

Consider the conditions 1-4 in the definition of consistent partition (Def. 8).

1. By the definition of *Apply* we know that all replicas in P remain unchanged except one which we denote r. So for all replicas $\langle L_j, s_j^0, s_j\rangle \neq r$ we know that $\beta \in L_j \Rightarrow \beta \in L \Rightarrow \beta \in L'$. Moreover the new log of replica r will be the same as the old log with the addition of operation α. And since all elements of the old log for r are in L, they are also in L'. Finally, since α is in L' then all operations for the log of r leading to P^{i+1} is in L'.
2. Consider the last state $\mathbf{s^k} = \langle s_1, \ldots, s_j, \ldots s_n\rangle$ where s_j is the state of the replica that will be changed by applying α. Let s_j' be the state of this replica in P^{i+1} which is the result of the transition $s_j \overset{\alpha}{\rightsquigarrow} s_j'$. By the inductive assumption we have that $\mathbf{s^0} \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_k}{\rightsquigarrow} \mathbf{s^k}$. Then $\mathbf{s^0} \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_k}{\rightsquigarrow} \mathbf{s^k} \overset{\alpha}{\rightsquigarrow} \mathbf{s^{k+1}}$ where $\mathbf{s^{k+1}} = \langle s_1, \ldots, s_i', \ldots s_n\rangle$ is a partition transition according to Definition 5.
3. By the inductive assumption we know that P^i is consistent and therefore $\forall j \leq k$ $\mathbf{s^j}$ is constraint consistent. Further since $Apply(\alpha, P^i)$ is constraint consistent according to (2), $\mathbf{s^{k+1}}$ is constraint consistent.
4. The order holds for L according to the inductive assumption. Let t be the point for $handle(\beta)$ in γ. For the order to hold for L' we need to show that $\alpha \nrightarrow \beta$ for all operations β in L. Since β appears in L there must exist a $handle(\beta)$ at some time point t' in γ. Then according to Theorem 2 $\alpha \nrightarrow \beta$ (since if $\alpha \rightarrow \beta$ then $t < t'$ and obviously $t < t'$). □

7 Related Work

In this section we will discuss how the problem of reconciliation after network partitions has been dealt with in the literature. For more references on related topics there is an excellent survey on optimistic replication by Saito and Shapiro [7]. There is also an earlier survey discussing consistency in partitioned networks by Davidson et al. [8].

Gray et al. [9] address the problem of update everywhere and propose a solution based on a two-tier architecture and tentative operations. However, they do not target full network partitions but individual nodes that join and leave the system (which is a special case of partition). Bayou [10] is a distributed storage system that is adapted for mobile environments. It allows updates to occur in a partitioned system. However, the system does not supply automatic reconciliation in case of conflicts but relies on an application handler to do this. This is a common strategy for sorting out conflicts, but then the application writer has to figure out how to solve them. Our approach is fully automatic and does not require application interaction during the reconciliation process.

Some work has been done on partitionable systems where integrity constraints are not considered, which simplifies reconciliation. Babaouglu et al. [11] present a method for dealing with network partitions. They propose a solution that provides primitives for dealing with shared state. They do not elaborate on dealing with writes in all partitions except suggesting tentative writes that can be undone if conflicts occur. Moser et al. [12] have designed a fault-tolerant CORBA extension that is able to deal with node crashes as well as network partitions. There is also a reconciliation scheme described in [13]. The idea is to keep a primary for each object. The state of these primaries are transferred to the secondaries on reunification. In addition, operations which are performed on the secondaries during degraded mode are reapplied during the reconciliation phase. This approach is not directly applicable with integrity constraints.

Most works on reconciliation algorithms dealing with constraints after network partition focus on achieving a schedule that satisfies order constraints. Fekete et al. [14] provide a formal specification of a replication scheme where the client can specify explicit requirements on the order in which operations are to be executed. This allows for a stronger requirement than the well-established causal ordering [15]. Our concept of ordering is weaker than causal ordering, as it is limited to one client's notion of an expected order of execution based on the replies that the client has received. Lippe et al. [16] try to order operation logs to avoid conflicts with respect to a *before* relation. However, their algorithm requires a large set of operation sequences to be enumerated and then compared. The IceCube system [17, 18] also tries to order operations to achieve a consistent final state. However, they do not fully address the problem of integrity constraints that involve several objects. Phatak et al. [19] propose an algorithm that provides reconciliation by either using multiversioning to achieve snapshot isolation [20] or using a reconciliation function given by the client. Snapshot isolation is more pessimistic than our approach and would require a lot of operations to be undone.

8 Conclusions and Future Work

We have investigated a reconciliation mechanism designed to bring a system that is inconsistent due to a network partition back to a consistent state. As the reconciliation process might take a considerable amount of time it is desirable to accept invocations during this period.

We have introduced an order relation that forces the reconciliation protocol to uphold virtual partitions in which incoming operations can be executed. The incoming operations cannot be executed on the state that is being constructed. Since the protocol would then have to discard all the operations that the client expects to have been performed. However, maintaining virtual partitions during reconciliation will make the set of operations to reconcile larger. Thus, there is a risk that the reconciliation process never ends.

We have proved that the proposed protocol will indeed result in a stable partition state given certain timing assumptions. In particular, we need time bounds for message delays and execution time as well as an upper bound on client invocation rate. Moreover, we have proved that the result of the reconciliation is correct based on a correctness property that covers integrity consistency and ordering of operations.

The current work has not treated the use of network resources by the protocol and has not characterised the middleware overheads. These are interesting directions for future work. Performing simulation studies would show how much higher availability is dependent on various system parameters, including the mix of critical and non-critical operations. Another interesting study would be to compare the performance with a simulation of a majority partition implementation.

An ongoing project involves implementation of replication and our reconciliation services on top of a number of well-known middlewares, including CORBA [3]. This will allow evaluation of middleware overhead in this context, and a measure of enhanced availability compared to the scenario where no service is available during partitions.

References

1. Szentivanyi, D., Nadjm-Tehrani, S.: Middleware Support for Fault Tolerance. In: Middleware for Communications. John Wiley & Sons (2004)
2. Felber, P., Narasimhan, P.: Experiences, strategies, and challenges in building fault-tolerant corba systems. IEEE Trans. Comput. **53**(5) (2004) 497–511
3. DeDiSys: European IST FP6 DeDiSys Project. http://www.dedisys.org (2006)
4. Asplund, M., Nadjm-Tehrani, S.: Post-partition reconciliation protocols for maintaining consistency. In: Proceedings of the 21st ACM/SIGAPP symposium on Applied computing. (2006)
5. Szentivanyi, D., Nadjm-Tehrani, S., Noble, J.M.: Optimal choice of checkpointing interval for high availability. In: Proceedings of the 11th Pacific Rim Dependable Computing Conference, IEEE Computer Society (2005)
6. Spread: The Spread Toolkit. http://www.spread.org (2006)

7. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. **37**(1) (2005) 42–81
8. Davidson, S.B., Garcia-Molina, H., Skeen, D.: Consistency in a partitioned network: a survey. ACM Comput. Surv. **17**(3) (1985) 341–370
9. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (1996) 173–182
10. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in bayou, a weakly connected replicated storage system. In: SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM Press (1995) 172–182
11. Babaoglu, Ö., Bartoli, A., Dini, G.: Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. IEEE Trans. Comput. **46**(6) (1997) 642–658
12. Moser, L.E., Melliar-Smith, P.M., Narasimhan, P.: Consistent object replication in the eternal system. Theor. Pract. Object Syst. **4**(2) (1998) 81–92
13. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: Replica consistency of corba objects in partitionable distributed systems. Distributed Systems Engineering **4**(3) (1997) 139–150
14. Fekete, A., Gupta, D., Luchangco, V., Lynch, N., Shvartsman, A.: Eventually-serializable data services. In: PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM Press (1996) 300–309
15. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565
16. Lippe, E., van Oosterom, N.: Operation-based merging. In: SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments, New York, NY, USA, ACM Press (1992) 78–87
17. Kermarrec, A.M., Rowstron, A., Shapiro, M., Druschel, P.: The icecube approach to the reconciliation of divergent replicas. In: PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM Press (2001) 210–218
18. Preguica, N., Shapiro, M., Matheson, C.: Semantics-based reconciliation for collaborative and mobile environments. Lecture Notes in Computer Science **2888** (2003) 38–55
19. Phatak, S.H., Nath, B.: Transaction-centric reconciliation in disconnected client-server databases. Mob. Netw. Appl. **9**(5) (2004) 459–471
20. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. In: SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (1995) 1–10