Post-Partition Reconciliation Protocols for Maintaining Consistency

Mikael Asplund and Simin Nadjm-Tehrani
Department of Computer and Information Science
Linköping university SE-581 83 Linköping, Sweden
{ mikas, simin }@ida.liu.se

ABSTRACT

This paper addresses design exploration for protocols that are employed in systems with availability-consistency trade-offs. Distributed data is modelled as states of objects replicated across a network, and whose updates require satisfaction of integrity constraints over multiple objects. Upon detection of a partition, such a network will continue to provide delivery of services in parallel partitions; but only for updates with non-critical integrity constraints. Once the degraded mode ends, the parallel network partitions are reconciled to arrive at one partition.

Using a formal treatment of the reconciliation process, three algorithms are proposed and studied in terms of their influence on service outage duration. The longer the reconciliation time, the lower is system availability; since the interval in which no services are provided is longer. However, the reconciliation time in turn is affected by the time to construct the post-partition system state. The shorter the construction time the higher is the number of updates that took place in the degraded mode but that will not be taken up in the reconciled partition. This will lead to a longer interval for rejecting/redoing these operations and thereby increase reconciliation time.

Keywords

reconciliation protocol, availability, partition, consistency, trade-off

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

1. INTRODUCTION

Distributed provision of services is a natural consequence of global business operations and at the heart of modern complex command and control decision systems. Central to distributed services in a data-centric application is how to maintain data consistency, here modelled as consistency among the states of object replicas. As operations are performed at multiple nodes in a replicated system, a propagation mechanism is employed to update the other nodes and to maintain consistency among replicas. There is clearly an availability-consistency trade-off. The performance of the system during normal operation, as well as during failures, is affected by the replication and reconciliation protocols.

An excellent recent survey on optimistic replication reflects the need for determining the requirements of an application and selection of design parameters using multiple criteria [14]. The extreme ends of the spectrum are the provision of fastest possible services at occasional costs of consistency, and the upholding of full consistency at the cost of availability (and thereby performance). There are several points in between based on focusing on other criteria. While covering a massive body of work on the issues related to this trade-off, the survey does not emphasise failure models, in particular partitions.

This paper concentrates on applications where data consistency is strived for, and the standard data integrity constraints on distributed data are used when performing operations on objects with replicas on multiple nodes. The novelty of our approach is a formal treatment of the reconciliation policy for achieving integrity consistency after *network partitions*. Several instances of such applications are studied in the European DeDiSys project [18], among them are the Distributed Telecommunication Management System (DTMS) and the Advanced Control System (ACS).

Consider a network that suffers from partition failures from time to time. During normal operation there is one primary for every object as well as a number of replicas. Replicas of the object are kept for distributed access and better performance. This paper assumes a passive replication model and assumes that a replication protocol takes care of the replica updates during normal operation. Updates can typically be conditional on satisfaction of integrity constraints that can encompass other objects (potentially on other nodes). Once the system has detected a partition, it starts to operate in a degraded mode (see Figure 1). We assume that the network can partition several times and concurrently but that no faults occur during reconciliation.

We further assume the existence of a detection mechanism (e.g. membership service in the middleware) that supports the transition to the degraded mode. During this phase there will be one primary copy of every object per partition. For operations that need to satisfy *non-critical* integrity constraints, the services may proceed to be delivered

SAC'06 April 23-27, 2006, Dijon, France

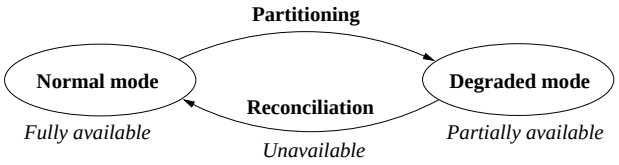

Figure 1: System modes

in multiple partitions, even though distributed updates can not be propagated to all relevant nodes. For operations that depend on *critical* integrity constraints, the updates will be blocked waiting for the partition to be repaired.

The final system state that is the result of the reconciliation can be achieved either by state transfer or by operation transfer. That is, the changes can either be propagated to the rest of the system by transferring the resulting state or the operations that led to the new state. The state transfer approach is attractive due to the fact that it allows for fast recovery. Write-write conflicts can be dealt with in such systems using time-stamps or version vectors [3]. Integrity constraints are hard to enforce in a state transfer system. If a system state is found to be inconsistent after merging a number of object states it might be hard to get the system back to a consistent state. Operation transfer systems, on the other hand, are more flexible in this respect at the cost of being more time consuming. This is the approach we adopt in two of our algorithms.

This paper focuses on the transition from degraded mode to the normal mode, i.e. the reconciliation of object states that have been subject to updates within multiple partitions. It demonstrates the choices that affect the design of a reconciliation algorithm and the influence of the design decisions on availability and performance of the system. In this model, the system does not accept new updates during the reconciliation process. Hence, it is unavailable during that period. We study the trade-off between saving time on constructing a post-partition state and the time needed to be spent on undoing the operations whose effect is not reflected in the resulting state. The trade-off is affected by parameters such as the time needed to fully deal with a rejected operation and the time the system has been in degraded mode (time to repair).

The contribution of this paper is a formal description of three algorithms for reconciliation, metrics for evaluating reconciliation algorithms and an experimental study of the trade-off between time needed for an elaborate reconciliation and the time spent on rejecting operations.

2. RELATED WORK

This section will give an overview of related works, in particular, optimistic replication. We will discuss how the problem of reconciliation after network partition has been dealt with in the literature. For more references on related topics there is, apart from the already mentioned survey by Saito and Shapiro [14], an earlier survey discussing consistency in partitioned networks by Davidson et al. [5].

Gray et al. [6] address the problem of update everywhere and their reconciliation approach is based on operation transfer. The authors demonstrate that the number of conflicts increases rapidly when the number of operations increases. They 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 [17] is a distributed storage system that is adapted for mobile environments. It allows updates to occur in a partitioned system. Reconciliation is operation based and conflict detection is done syntactically. However, the system does not supply automatic reconciliation in case of conflicts but relies on the application to do this. This is a common strategy for operation transfer systems. Conflict detection is more straightforward than conflict resolution and thus the application is required to sort out conflicts. Our approach is fully automatic and does not require application interaction during the reconciliation process. Most works on reconciliation algorithms after network partition focus on achieving a schedule that satisfies order constraints. Lippe et al. [9] try to order operation logs to avoid conflicts of the *Before* relation. However, their algorithm does not scale well as the number of operations increase. The IceCube system [7] also tries to order operations to achieve a consistent final state. Their approach requires the construction of many operation logs, which is a computationally intensive process. They deal with integrity constraints only in the sense that operations that violate an integrity constraint are aborted. However, no effort is made to minimise integrity constraint violations. Preguica et al. [11] extend the IceCube approach by differentiating between different types of constraints and improve the algorithms by dividing the work in subproblems that can be solved separately since they operate on disjoint data. However, they do not fully address the problem of integrity constraints that involve several objects.

Phatak et al. [10] propose an algorithm that provides reconciliation by either using multiversioning to achieve snapshot isolation [2] or using a reconciliation function given by the client. The system deals well with possible side effects since reconciliation is always performed. An interesting technique that has been applied in collaborative systems is that of operational transformation [15]. The idea is to keep the intent of each operation rather than the actual data operations. Thus, the order in which operations are applied can be changed even though the operations are not originally commutative. Unfortunately, the required amount of knowledge about the operation semantics reduces the applicability of this approach.

Several replicated file systems exist that deal with reconciliation in some way [13, 8]. Balasubramaniam and Pierce [1] specify a set of formal requirements on a file synchroniser. These are used to construct a simple state based synchronisation algorithm. Ramsey and Csirmaz [12] present an algebra for operations on file systems. Reconciliation can then be performed on operation level and the possible reorderings of operations can be calculated.

Performance studies for fully consistent systems tend to focus on recovery crash failures [16]. To our knowledge this is the first performance study of the consistency/availability trade-off for reconciliation after network partition with system wide integrity constraints.

3. PRELIMINARIES

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

Moreover, we will introduce utility as a metric to be used in Sections 4 and 5.

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* $o = (S, op, T)$ *where* S *is the set of possible states,* op *is the set of operations that can be applied to the object state and* $T \subseteq S \times op \times S$ *is a transition relation on states and operations.*

Transitions from a state s to a state s' will be denoted by $s \overset{\alpha}{\rightsquigarrow} s'$ where $\alpha \in op$.

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

Intuitively, object operations should only be performed if they do not violate integrity constraints. The results presented in this paper are not affected by whether integrity constraints are checked before or after operations.

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* $o = (S, op, T)$ *is a triple* $r = (L, s^0, s^m)$ *where the log* $L = \langle \alpha_1 \ldots \alpha_m \rangle$ *is a sequence of applied operations* $\alpha_i \in op$. *The initial state is* $s^0 \in S$ *and* $s^m \in S$ *is a state such that* $s^0 \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_m}{\rightsquigarrow} s^m$.

3.2 Partition

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. 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 $\langle s_1, \ldots, s_n \rangle$. Transitions over object states can now be naturally extended to transitions over partition states.

DEFINITION 5. $\mathbf{s^j} \overset{\alpha}{\rightsquigarrow} \mathbf{s^{j+1}}$ *is a* partition transition *if and only if 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* $\mathbf{s^{j+1}} = \langle s_1, \ldots, s_i', \ldots, s_n \rangle$.

3.3 Order

Operations can have dependencies on other operations. If the operations are executed in a transactional context most of these dependencies are made explicit. We will use a *before* relation as described by Preguica et al. [11].

DEFINITION 6. *Two operations can be related as* $\alpha \rightarrow \beta$ *meaning that operation* α *must be* before *the operation* β.

Note that this relation creates an ordering of operations. However, the existence of α in an operation log does not require that β is present and vice versa. Our algorithms can be extended to other ordering relations such as causality. It is also possible to extend with the "must have" relation ($\triangleright$ of [11]) ,which would allow more general dependencies. However, for the trade-offs that we study we consider the *before* ordering relation adequate.

Note also that the order relationship 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).

3.4 Consistency

Our reconciliation algorithms 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 algorithm 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* $P = \{(L_1, s_1^0, s_1), \ldots, (L_n, s_n^0, s_n)\}$ *is* constraint consistent *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.

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 an operation sequence* $L = \langle \alpha_1, \ldots, \alpha_k \rangle$ *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. $\alpha_i \rightarrow \alpha_j \Rightarrow i < j$

We will use the above correctness criterion in evaluation of our reconciliation algorithms.

3.5 Utility

Note that a partition with empty logs can still be a consistent partition. A goal of the reconciliation algorithms will be to create a new partition, and within it include as many operations as possible already performed within the existing partitions (during the time that the system operated in degraded mode). However, this might not be possible due to strict consistency requirements. Since some of the operations will have to be dropped it is useful to have a notion of usefulness for various operations. For now we will simply define a notion of utility and assume that it is dictated by the application. Later we will discuss how utility can be tied to execution times and what other utility functions could be used.

DEFINITION 9. *Let op_i be the operations for object o_i. A* utility function $U : \bigcup_{i=1}^{n} op_i \rightarrow \mathbb{R}$ *is a mapping from the operations to a real number.*

The definition can be extended to apply to a partition by letting the utility of a partition be the sum of the utilities of all the executed operations appearing in the logs of the partition.

4. RECONCILIATION ALGORITHMS

This section presents three reconciliation algorithms that deal with multi-object constraints. The first algorithm is based on state transfer whereas the other two use operation transfer. The operation based algorithms rely on the assumption that there is a desired partial ordering of operations. The partial order is based on the before relation defined in Section 3.3. We make this assumption to be able to concentrate on the problem of constraint consistency. If the "before" relation of the operations is such that there are cycles, then some operations must first be removed so that a partial order can be created. How such a cyclic dependency graph can be made acyclic was first investigated by Davidson [4]. Moreover, we assume that all the partitions that the algorithms take as input have a common initial state. So $\mathbf{s_i^0} = \mathbf{s_j^0}$ where i and j denotes different partitions.

The reconciliation starts when the network is no longer physically partitioned. This information is assumed to be available through a group membership service. The reconciliation is performed at one node. Therefore in the first phase replicas must transfer the partition logs to this coordinator node. Once the reconciliation algorithm has been performed at this node the post partition state is replicated back to all nodes.

4.1 Choose states

The reconciliation process aims at creating a resulting partition state from a number of partition states. If consistency is maintained in each partition during the degraded mode then each partition state is correct prior to reconciliation. However, as this paper does not aim to model the replication protocols during normal and degraded mode, we start by assuming that partition states are consistent.

One approach would be to select any of the partitions' states as the final state of the system. Another approach would be to construct the repaired partition state by combining the object states from several partitions. However, such a constructed state cannot be guaranteed to be consistent. Therefore, that approach will not be investigated here. In Algorithm 1 a partition choosing algorithm is described. What partition to choose as the resulting partition can depend on the application, but since this presentation is focused on maximising the utility and minimising reconciliation time we let the algorithm choose using the function *GetPartWithMaxUtility*. This function selects from a set of partitions the partition with the highest utility.

This approach generates a consistent partition. In a way this approach is very similar to having only one partition (e.g. the majority) operating in the partitioned phase. Although the system proceeds to service in different partitions in degraded mode, there are requests that the clients believe to be accepted and will later be undone.

4.2 Merging operation sequences

Algorithm 1 CHOOSE1

Input: $p_1, \ldots, p_m$ /* m partitions */
Output: $\langle P, Reject \rangle$ /* P: A new partition
Reject: rejected operations */

```
P = GetPartWithMaxUtility(p_1, ..., p_m)
for each p in {p_1, ..., p_m}
do if p ≠ P
      then op ← GetAllOps(p)
           for each α in op
           do Reject ← Reject ∪ {α}
return ⟨P, Reject⟩
```

Instead of just choosing one state, merging can be done on an operation level. That is, to start from the initial state and re-execute the operations from the different partitions so that as many operations as possible are accepted. This means aiming for the highest possible utility. One way to do this is by simply merging the sequences of operations that have been stored in the replicas at each partition and apply them in order. The algorithm MERGE implements this idea and uses the order specified by the predefined operation ordering.

Before describing the algorithm we present a function called *AddSuccs* that takes a partially ordered set and an element α and adds to the set the sole successors of α (successors that are not successors of another element in the set). Moreover we introduce the function *Apply*, which is defined as follows $Apply(\alpha, p) = p \setminus \{r\} \cup r'$ where $r = (L, s^0, s^m)$ is a replica for object $o = (S, op, T)$ and $\alpha \in op$. $r' = (L + \langle \alpha \rangle, s^0, s^{m+1})$ is also a replica for o and $(s^m \overset{\alpha}{\rightsquigarrow} s^{m+1}) \in T$.

Algorithm 2 starts by creating a set of candidate elements from which the next operation to execute is chosen. The first candidate contains the elements that have no predecessor in the set of operations. When an operation is executed it is removed from the candidate set and replaced by its sole successors. If the result is consistent then the new resulting partition is updated. Otherwise the operation is put in the set of rejected operations. This procedure is repeated until all the elements have been considered.

Algorithm 2 MERGE

Input: $p_1, \ldots, p_m$ /* m partitions */
$\rightarrow$ /*a partial ordering relation for all the operations in the domain */
Output: $\langle P, Reject \rangle$ /* P: A new partition
Reject: rejected operations */

```
P = {(⟨⟩, s_1^0, s_1^0), ..., (⟨⟩, s_n^0, s_n^0)}
op ← ⋃_{p∈{p_1,...,p_m}} GetAllOps(p)
SortedOp ← Sort(op, →)
op_cand ← GetLeastElements(SortedOp)
while op_cand ≠ ∅
do α ← ChooseArbitraryElement(op_cand)
   op_cand ← AddSuccs(op_cand, {α}) \ {α}
   if Apply(α, P) is Constraint Consistent
      then P ← Apply(α, P)
      else Reject ← Reject ∪ {α}
return ⟨P, Reject⟩
```

Note that this algorithm selects the elements nondeterministically. Hence, there are potentially several different results that can be obtained from using this algorithm on the same partition set.

4.3 Greatest Expected Utility (GEU)

The previous algorithm does not try to order operations to maximise the utility because it has no way of comparing operation sequences and decide on one operation ordering in preference to another. This section presents an algorithm that tries to maximise the expected utility. To explain this concept we will first introduce probability of violation.

4.3.1 Probability of Violation

A reconciliation process must make decisions about the order that operations are to be performed at each replica. Recall that operation ordering requirements and integrity constraints are orthogonal. Even if the execution of an operation is acceptable with regard to the operation ordering ($\rightarrow$) it can put the partition in an inconsistent state after execution. Moreover, it could be the case that applying a particular operation from another partition may hinder the execution of operations that have taken place within this partition. Trying out the operations to see if an inconsistency is caused can turn out to be very time consuming. So the ideal situation would be to decide whether a given operation should be executed or not without actually executing it. Suppose that for each operation we could determine the probability that we would get an inconsistency if we were to apply it. Then this *probability of violation* could be used to selectively build execution sequences for the resulting partition.

One may wonder how such a parameter would be obtained. In a real application we must be practical and consider if we can calculate this probability from past history or if it is possible to measure. We will explore an approach that is based on system profiling. When the system is functioning in normal mode the consistency violations are immediately detected after applying an operation. By keeping track of how often a particular operation causes an inconsistency we get a rough measure of the probability of violation for that particular operation.

4.3.2 Expected Utility

There are now two ways of measuring the appropriateness of executing a given operation. The utility gives information on how much there is to gain from executing the operation. The probability of violation, on the other hand, tells us what are the chances of getting a result that cannot be used.

The *expected utility* is a measure of what we can expect to achieve given the utility and the violation probability. We quantify the gain of a successful application of the operation α with its utility $U(\alpha)$. The expected utility of operation α is $EU(\alpha) = (1 - P_{viol}(\alpha)) \cdot U(\alpha) - P_{viol}(\alpha) \cdot U(\alpha) \cdot C$ where $P_{viol}(\alpha)$ is the probability of violation for operation α. The scaling factor C reflects the fact that the cost of a violation can be more expensive than just the loss of the utility of the operation. Increasing C increases the importance of P_{viol}. In all the experiments shown later we equate C with 10.

The success of the algorithm will depend on how well the the expected utility can be estimated. The difficult part of this estimate is to estimate probability of violation. The $U(\alpha)$ part, that is the utility of an operation, can to begin with be uniform for all actions. But profiling will punish those operations whose constraints were often violated in the lifetime of the system.

4.3.3 The Algorithm

In Algorithm 3 the expected utility is used to choose operation orderings. The algorithm starts by collecting the operations and chooses a set of candidates in the same way as MERGE. But instead of choosing arbitrarily among the candidates, the operation with the highest expected utility is chosen.

If an operation is successfully executed (leads to a consistent state) a new set of candidates is created. If, on the other hand, the execution of an operation leads to an inconsistent state then the state is reverted to the previous state and the operation is put in the set of tried operations. The number of times an operation can be unsuccessfully executed due to inconsistencies is bounded by the input parameter $TryBound$. When the number of tries exceeds this bound the operation is abandoned and put in the set of rejected operations.

If the set of tried elements is equal to the set of candidates (meaning that none of the operations could be applied without causing an inconsistency) then the operation with the lowest expected utility is replaced by its successors. This ensures that the algorithm terminates and all operations are tried.

Algorithm 3 GEU

Input: $p_1, \ldots, p_m$ /* m partitions */
$\rightarrow$ /*a partial ordering relation for all the operations in the domain */
$TryBound$ /* Maximum number of times an operation is tried */
Output: $\langle P, Reject \rangle$ /* P: A new partition
Reject: rejected operations */

$P = \{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_n^0, s_n^0)\}$
$op \leftarrow \bigcup_{p \in \{p_1, \ldots, p_m\}} \textsc{GetAllOps}(p)$
$SortedOp \leftarrow \textsc{Sort}(op, \rightarrow)$
$op_{cand} \leftarrow \textsc{GetLeastElements}(SortedOp)$
$Tried \leftarrow \emptyset$/* Set of tried elements */
$\forall \alpha \in op: \ TryCount(\alpha) \leftarrow 0$
while $op_{cand} \neq \emptyset$
do $\alpha \leftarrow \textsc{ChooseHighestEU}(op_{cand} \setminus Tried)$
  if $\textsc{Apply}(\alpha, P)$ is Constraint Consistent
    then $P \leftarrow \textsc{Apply}(\alpha, P)$
      $op_{cand} \leftarrow \textsc{AddSuccs}(op_{cand}, \alpha) \setminus \{\alpha\}$
      $Tried \leftarrow \emptyset$
    else $TryCount(\alpha) \leftarrow TryCount(\alpha) + 1$
      if $TryCount(\alpha) > TryBound$
        then $op_{cand} \leftarrow \textsc{AddSuccs}(op_{cand}, \alpha) \setminus \alpha$
          $Reject \leftarrow Reject \cup \{\alpha\}$
        else $Tried \leftarrow Tried \cup \{\alpha\}$
  /* If all operations in op_{cand} have been tested, */
  if $Tried = op_{cand} \neq \emptyset$
    then /* remove the element with lowest EU */
      $\alpha \leftarrow ChooseLowestEU(op_{cand})$
      $op_{cand} \leftarrow \textsc{AddSuccs}(op_{cand}, \alpha) \setminus \{\alpha\}$
      $Tried \leftarrow Tried \setminus \{\alpha\}$
      $Reject \leftarrow Reject \cup \alpha$
return $\langle P, Reject \rangle$

5. EVALUATION

The algorithms have been evaluated both by a theoretical study and through experimental studies.

5.1 Correctness

Using the definitions of consistency in Section 3 we have shown that Algorithms 2 and 3 are correct, in the sense of producing consistent states.

THEOREM 1. *MERGE and GEU lead to consistent partitions.*

The proof is omitted due to space limitations.

5.2 Experimental Evaluation

This section contains an experimental evaluation of the three algorithms. First we will explain what metrics we have used and how we expect the algorithms to behave. We have written a test bed application for evaluation purposes, which will be described shortly. An explanation of how the simulations were done is finally followed by the simulation results.

5.2.1 Performance metrics

We have introduced utility as a metric to compare the usefulness for each operation and we can therefore talk about the utility of a reconciliation function as the accumulated utility from the applied operations minus the utility of the rejected operations. The utility of an operation can be dependent on a number of factors, the lost benefit if it is not applied, or the economic gain for a business that gets paid by its customers for delivered service. Also, the time taken to execute an operation is important if we are considering undoing/redoing the operation; thus, the higher the execution time the lower utility. However, we will in this paper let the utility be a random number, which is assigned to each operation.

The reason that there are several divergent partitions to reconcile is that they have been accepting updates as a way to increase availability of the system. As the system does not accept new updates during the reconciliation process, all the time spent performing reconciliation will reduce the availability. The time that the system is partially unavailable is the sum of the time to repair from network partition plus the time taken to reconcile the system[1], $T_{Unavailable} = T_{Repair} + T_{Reconcile}$. Note that the reconciliation time $T_{Reconcile}$ can also be divided in two parts, $T_{Construct}$, which is the time to construct a new partition state and T_{Reject}, which is the time taken to perform all the necessary actions for the operations that have been rejected. This rejection mechanism potentially involves sending a notification to the client, which will then try to redo the operation. So $T_{Reconcile} = T_{Construct} + T_{Reject}$. If no notification of the clients are necessary or rejecting an operation for some other reason takes little time to perform then there is little use in spending time constructing an elaborate partition state. If, on the other hand, the undo operation is expensive in time then minimising T_{Reject} is the main goal.

With the experimental evaluation we aim to demonstrate that for short time to repair and a short reject time for operations the CHOOSE1 algorithm is good enough. Since the system does not stay long in degraded mode there are not many operations lining up and therefore the loss of utility is outweighed by the gain of a short construction time $T_{Construct}$. However, we expect the MERGE and GEU algorithms to achieve much better utility for longer repair times and thus also shorter reconciliation times. Moreover, we will demonstrate that the key to choosing between MERGE and GEU is the time needed to reject operations. If undoing an operation is time consuming then the GEU algorithm should perform better than MERGE as it aims at minimising the number of rejected operations.

[1]For some reconciliation algorithms it may be possible to accept new operations during reconciliation but this complicates maintaining expectations of consistency at clients.

5.2.2 Test Bed Application

A demo application has been developed to serve as a test bed for trade-off studies. It is composed of a set of real number objects. Possible operations are the addition, multiplication and division. An operation is applied to the current value with a random constant. The constant is an integer uniformly distributed between $0 < |m| < 11$. This creates a total of 60 distinct operations. There are also integrity constraints in the system expressed as: $\sum n_i + c_1 < \sum n_j + c_2$ where n_i and n_j are object values and c_1, c_2 are constants. This allows for a wide variety of constraints to be expressed. Although the application is very simple it is enough to give an indication of how the algorithms perform. Moreover the application is interesting in the sense that although the current state is known it is hard to predict how the execution of a certain operation will affect the consistency two or three operations in the future.

5.2.3 Experimental setup

The application has been implemented in Python together with the surrounding framework needed to perform the simulations. Eight objects were initialised with random numbers, uniformly distributed between -100 and 100. The results in this paper are based on five integrity constraints created between two randomly chosen objects so that the initial partition state was constraint consistent. [2]

Each application instance was profiled during a phase where 10000 operations were scheduled and statistics could be gathered about whether the operations succeeded or not. This data was then used to calculate P_{viol}. The system was then split up into three partitions in which operations were scheduled for execution with a load of 10 operations per second. Each object received the same load of requests. For each measurement point 100 samples were obtained and averaged.

Each operation was assigned a utility with a normal distribution $N(1.0, 0.1)$. The execution times of the algorithms were measured by assigning execution times for different steps of the algorithm, more specifically a step that does not require disk access takes $100ns$, disk operations take $1\mu s$ and the task of rejecting an operation takes $1ms$. Constraint checking is a step that is assumed not to require disk access. Rejecting an element is assumed to require notifying clients over the network and is therefore far more time consuming than other types of operations.

5.2.4 Results

In Figure 2 the utility of the reconciliation process is plotted against the time to repair (that is, the time spent in

[2]Simulations have also been performed with more objects and constraints with similar results.

degraded mode) which in turn decides the number of operations that are queued up during the degraded mode.

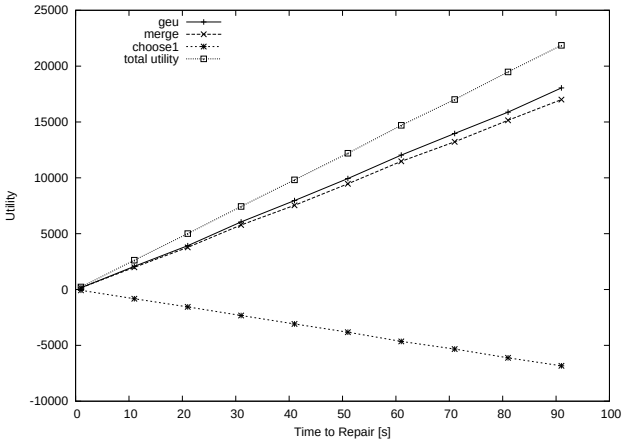

Figure 2: Utility vs. Time to Repair [s]

It is clear that just choosing the state from one partition will result in a low utility. The operations of all the other partitions will be undone and this will result in a negative utility given that the partitions are approximately of the same size. If the utility of one partition would heavily outweigh the utility of the other partitions (or there would be only two partitions) then the CHOOSE1 algorithm performs significantly better. However, there would have to be a very large imbalance for it to outperform the other algorithms. Further experiments indicate that the gap remains even if the time to reject for an operation is reduced by a factor of 10. As the MERGE algorithm is nondeterministic we have also studied how it behaves over 100 runs with the same input parameters. For 31s as time to repair the resulting mean utility is 5881.1 and the standard deviation 31.5.

The upper curve shows the total utility which is the sum $\sum_{\alpha \in op} U(\alpha)$ where *op* is the set of all operations appearing in the partition logs. The optimal utility is less than or equal to this sum. For the optimal utility to be equal to the total utility it must be possible to execute the operations in such an order that inconsistencies do not occur. Since operations are chosen randomly this seems very unlikely. The GEU algorithm performs slightly better than the MERGE and the difference increases somewhat as the time to repair grows (meaning more operations to reconcile).

In Figure 3 the reconciliation times of the algorithms (including the time needed to deal with the rejected operations) are compared. The graphs illustrate that the three algorithms execute in linear time over the repair time (the longer the duration of partition the higher the number of performed operations). Essentially the results follow the same pattern as before. The GEU algorithm performs slightly better that the MERGE and both are significantly better than the CHOOSE1 algorithm[3]. As the rejection of operations is the most time consuming task this is not surprising. However, since the MERGE and GEU show very similar results we will study in more detail how the cost of rejection affects the performance comparison between the two.

In Figure 4 the reconciliation times of MERGE and GEU

[3]The CHOOSE1 algorithm could benefit from parallelising the task of undoing operations.

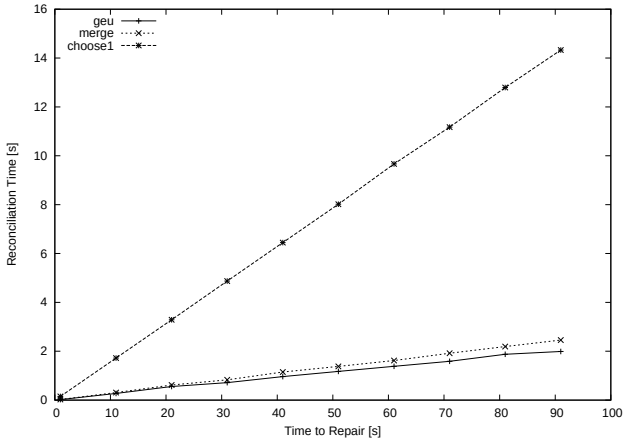

Figure 3: Reconciliation Time [s] vs. Time to Repair[s]

are plotted against the rejection time for one operation. As the MERGE algorithm has a shorter construction time $T_{Construct}$ than GEU it will result in a shorter reconciliation time when it is inexpensive to undo operations. When the time to reject an operation increases the effect of the longer construction time for GEU diminishes and the shorter total rejection time T_{Reject} results in a total reconciliation time that is shorter.

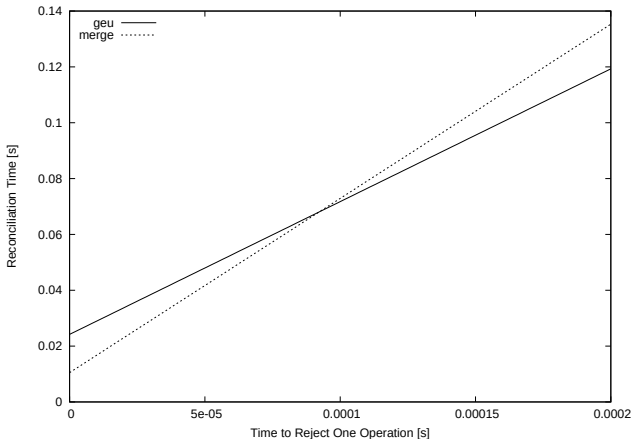

Figure 4: Reconciliation Time [s] vs. Time to Reject One Operation [s]

6. CONCLUSION

In this paper we have studied the reconciliation process with a focus on integrity constraint conflicts and the decisions that affect the design of reconciliation algorithms. To compare different algorithms we have both defined a theoretical correctness criterion and experimental metrics. Three reconciliation algorithms have been introduced.

We have studied the correctness properties of the algorithms as well as an experimental evaluation on the performance of the algorithms. With this evaluation we have demonstrated how the choice of reconciliation algorithm should be affected by parameters such as how long the average time is for the system to repair from network partitions and time

to reject an operation. CHOOSE1 should be considered in systems where the cost of rejecting operations is low and time to repair is short. This is because of the very short time required to create the post-partition state. However, when the time to repair a partition is long then MERGE and GEU will give higher utility than the CHOOSE1 algorithm. Which of the two algorithms performs best depends on the time it takes to reject(undo) one operation. MERGE requires more operations to be rejected but is better at quickly constructing a consistent partition state than the GEU algorithm.

More work needs to be done in considering the interaction between replication protocols and reconciliation protocols. Future work includes applying a selected reconciliation algorithm in a scenario with a realistic application and studying the implications of the choice of replication algorithm. Moreover, other utility functions relating to the state of the system are interesting to study as well as a more fine-grained metric to estimate the probability of violation.

The study of scale-up properties of the algorithms has not been relevant for the simple model in which constraint checking is assumed to be constant. Recall that the number of nodes and objects is not relevant as long as it does not affect the complexity of the constraints. Hence, scale-up studies would need to take into account the impact of size on complexity of constraints. Another direction for future work is the study of uneven load amongst objects and bursty request patterns that characterise hot-spots.

7. ACKNOWLEDGMENTS

This work has been supported by the FP6 IST project DeDiSys on Dependable Distributed Systems. We would like to thank all members of the DeDiSys project for their input and valuable ideas. A special thanks to Jaka Mocnik, Francesc Muñoz-Escoí and Stefan Beyer for commenting on the paper. Moreover, the CHOOSE1 protocol is a version of a replication and reconciliation protocol developed at Instituto Tecnologico de Informatica. We also thank anonymous reviewers for their comments.

8. REFERENCES

[1] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 98–108, New York, NY, USA, 1998. ACM Press.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM Press.

[3] J. D S Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. *Detection of mutual inconsistency in distributed systems*, pages 306–312. Artech House, Inc., Norwood, MA, USA, 1986.

[4] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Trans. Database Syst.*, 9(3):456–481, 1984.

[5] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Comput. Surv.*, 17(3):341–370, 1985.

[6] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM Press.

[7] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 210–218, New York, NY, USA, 2001. ACM Press.

[8] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[9] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 78–87, New York, NY, USA, 1992. ACM Press.

[10] S. H. Phatak and B. Nath. Transaction-centric reconciliation in disconnected client-server databases. *Mob. Netw. Appl.*, 9(5):459–471, 2004.

[11] N. Preguica, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. *Lecture Notes in Computer Science*, 2888:38–55, October 2003.

[12] N. Ramsey and E. od Csirmaz. An algebraic approach to file synchronization. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185, New York, NY, USA, 2001. ACM Press.

[13] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Conference Proceedings*, pages 183–195, Boston, MA, June 1994. USENIX.

[14] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[15] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68, New York, NY, USA, 1998. ACM Press.

[16] D. Szentiványi and S. Nadjm-Tehrani. Aspects for improvement of performance in fault-tolerant software. In *PRDC '04: Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04)*, pages 283–291, Washington, DC, USA, 2004. IEEE Computer Society.

[17] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM Press.

[18] Dedisys project. http://www.dedisys.org.