CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Middleware Extensions that Trade Consistency for Availability

Mikael Asplund[1], Simin Nadjm-Tehrani[1], Klemen Zagar[2]

[1] *Linköping University, SE-581 83 Linköping, Sweden, {mikas,simin}@ida.liu.se*
[2] *Cosylab, SI-1000 Ljubljana, Slovenia, klemen.zagar@cosylab.com*

SUMMARY

Replicated distributed object systems are deployed to provide timely and reliable services to actors at distributed locations. This paper treats applications in which data updates are dependent on satisfaction of integrity constraints over multiple objects. We propose a means of achieving higher availability by providing partition-awareness in middleware. The general approach has been illustrated by implementing a number of CORBA extensions that trade consistency for availability during network partitions. This paper contains a thorough experimental evaluation that shows the gains and costs of our approach. The experiments clearly illustrate the benefit of our protocols in terms of significantly higher availability and number of performed operations.

KEY WORDS: middleware, fault tolerance, network partitions

1. INTRODUCTION

Our society relies on getting correct service from a large number of computer systems. These computer systems are often not stand-alone systems that operate in isolation. Rather, they are distributed across different geographical locations connected through the Internet. Unfortunately, Internet connectivity is not reliable to the extent we would like to expect. For example, router misconfigurations, router crashes, congestion due to high load, a cable being cut off can all cause a site to be temporarily disconnected from the Internet.

Therefore, it is important to find solutions to ensure that computer systems stay available even when there are failures in the network. The fact that one group of computers is unable to communicate with another group via the network is called a partition fault. What we would

*Correspondence to:
Contract/grant sponsor: Publishing Arts Research Council; contract/grant number: 98–1846389

Received 4 February 2008
Revised 28 June 2008

like to achieve is to allow services to remain available despite network partitions. However, this is not trivial since most applications have strict requirements on the consistency of distributed data.

The database community has shown that using integrity constraints is a useful and efficient way of automatically dealing with application specific consistency requirements. However, most protocols for maintaining consistency in databases are very pessimistic in their nature. Distributed object systems, on the other hand, usually have consistency checks embedded inside the object methods. In our approach we make these constraints explicit and categorise them as critical and non-critical constraints in order to act optimistically during network partitions.

As an example, consider a system for handling medical journals. Accessing a journal requires authentication from a server that might be temporarily unavailable. However, this requirement can be seen as non-critical since relaxing it allows the medical personnel to continue working during surgery and a possible violation can be investigated later. However, deleting the journal should not be possible with no authentication and this constraint can be labelled critical. Note that every read here is effectively a write due to logging requirements.

Due to the complexity of designing fault-tolerant distributed applications, it is reasonable to put as much support as possible in the middleware layer to relieve the application programmers. One of the first middleware standards specified by the OMG group and widely deployed is CORBA. In the context of a European project [11] we have built an extension to CORBA that allows applications to give service during network partitions and support restoration of consistency. Thus, we temporarily trade consistency for availability. Studies in the project have shown that the generic ideas for tolerating partitions in an optimistic way carry over to other middlewares (in particular EJB and .Net that were also studied in the project [14]). In this paper we illustrate the ideas through CORBA extensions.

Previous work in the area of partition-tolerant middleware have been either very pessimistic in the sense that serializability is required (e.g., [16]), or they do not allow system-wide integrity constraints ([34, 23]). Our work differs mainly in the sense that the post-partition reconciliation process is able to use application semantics (in the form of integrity constraints) to construct an acceptable state in bounded time. A more detailed overview of related work can be found in Section 5. The contributions of this paper are twofold: (1) implementation of CORBA extensions that provide partition tolerance and trading support, including a partition-aware transaction manager that can properly deal with nested transactions. (2) Performance measurements on our platform to assess the gains in terms of added availability as well as the costs in terms of overhead.

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 [3]. 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). An early evaluation [4] of the continuous service protocol in a simulated environment (J-Sim) acted as

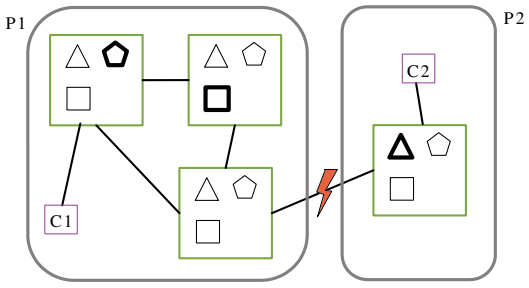


Figure 1. System Overview

a proof of concept. This paper extends the evaluation by embedding the reconciliation and replication mechanism in a CORBA platform and performance testing in a physical network.

The rest of the paper is organised as follows. Section 2 gives an overview of the system and the assumptions. In Section 3 the architecture and implementation of our middleware extension is described. The results of the performance evaluation are given in Section 4, followed by related work and conclusions (sections 5 and 6).

2. OVERVIEW AND SYSTEM MODEL

2.1. Overview

Consider the simple system depicted in Figure 1. There are four server nodes hosting three replicated objects. Each object is represented as a polygon with a primary replica marked with bold lines. There are also two clients, C1 and C2, that perform invocations on the objects in the system.

A link failure as indicated in the picture will result in a network partition. At first glance, this is not really a problem. The system is fully replicated so all objects are available at all server nodes. Therefore, client requests could in principle continue as if nothing happened. This is true for read operations, where the only problem is the risk of reading stale data. The situation with write operations is more complicated. If client C1 and C2 both try to perform a write operation on the square object then there is a write-write conflict when the network is reunified. Even if clients are only restricted to writing to primary replicas (e.g., C2 is only allowed to update the triangle) there is the problem of integrity constraints. If there is some constraint relating the square and the triangle, C2 cannot safely perform a write operation since the integrity constraint might be violated.

The most common solution to this problem is to deny service to the clients until the system is reunified (shown in Figure 2). This is a safe solution without any complications, and the

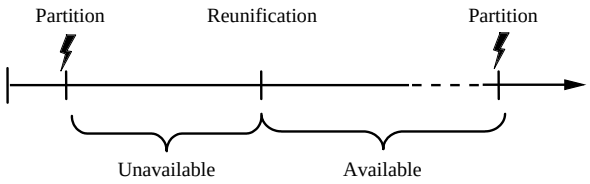


Figure 2. Pessimistic approach: No service during partitions

system can continue immediately after the network reunifies. However, it may result in low availability.

The second approach of letting a majority partition [18] continue operating is better. In our example this would allow at least partial availability in the sense that C1 gets serviced whereas C2 is denied service. However, there are two problems associated with this solution. First of all, there might not be a primary partition (consider the case where the network splits in two partitions with 2 nodes each). Secondly, it does not allow prioritising between clients or requests. It might be the case that it is critical to service C2 whereas C1's operations are not as important. Such requirements cannot be fulfilled using the majority partition approach.

This leaves us with the optimistic approach where all partitions are allowed to continue accepting update operations. We say that the system operates in *degraded mode*. However, we still have the problems of replica conflicts and integrity constraint violations. This is why we need reconciliation algorithms to solve the conflicts and install a consistent state in all nodes. In other words, we temporarily relax the consistency requirements but restore full consistency later.

The reconciliation algorithms that we use are based on the idea of replaying operations that were performed during the degraded mode. By starting from a pre-partition state and replaying operations following a certain order and checking integrity constraints, the system ends up in a consistent state.

The optimistic approach poses several challenges. First of all, not all of the operations that were accepted during degraded mode can be kept in the reconciled system state. They might violate integrity constraints forcing us to revoke the operation and notify the client that the operation was cancelled. This can be problematic if there are side-effects associated with an operation so that the operation cannot be undone. There are two ways to tackle this problem. Either these operations are not allowed during degraded mode, or the application writer must supply compensating actions for such operations. A typical example of this is billing. Some systems allow users to perform transactions off-line without being able to verify the balance on the user's account. A compensating action in this case is to send a bill to the user.

What remains is that somehow the effect of the operation is cancelled. This has an important impact on the way we measure the availability of our system. When the operation was tentatively accepted, it was reasonable to consider the system to be available. However, as it later turns out, that was only *apparent availability* since the change mandated by that

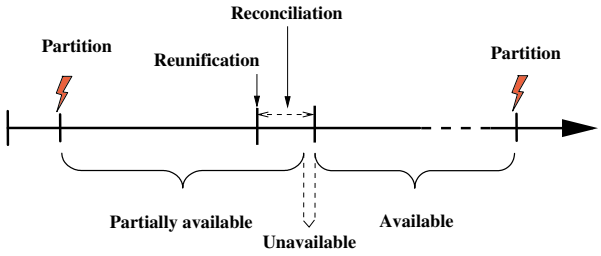


Figure 3. CS Optimistic approach: partial service during partitions

operation did not persist. Therefore, we need to distinguish apparent availability and devise metrics to account for revocations.

The second challenge with acting optimistically is that there might be integrity constraints that cannot be allowed to be violated, not even temporarily. This usually means that there is some side effect associated with the operation that cannot be compensated. To deal with this we differentiate between critical and non-critical constraints. During network partitions, only operations that affect non-critical constraints are allowed to continue. Therefore, the system is only partially available. However, our results still show that acting optimistically pays off in terms of number of performed operations.

Since this reconciliation procedure involves revoking or compensating some of the conflicting operations, some constraint-aware reconciliation protocols [3] stop incoming requests *during* reconciliation. We call this type of reconciliation algorithms stop-the-world algorithms since the system is totally unavailable during reconciliation. Since stopping service for this long has an adverse effect on availability, we have also constructed a reconciliation protocol that we call the Continuous Service (CS) reconciliation protocol. This protocol allows incoming requests during reconciliation (see Figure 3), except for a short interval in which the state is installed.

2.2. System model

As a fault model we assume that the network may partition multiple times, and that when the network reunifies, then it reunifies completely. That is, no partial network reunifications are allowed. Such cases can be dealt with but we have not implemented support for it in our middleware. Moreover, we assume that nodes never crash. The reason for this assumption is that a pause-crash fault (i.e. nodes are assumed to have a persistence layer) can be seen as a partition fault where a single node is disconnected for some time and then rejoins the network.

In order to successfully handle client expectations we must also make some assumptions regarding ordering of operations. We recall that 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

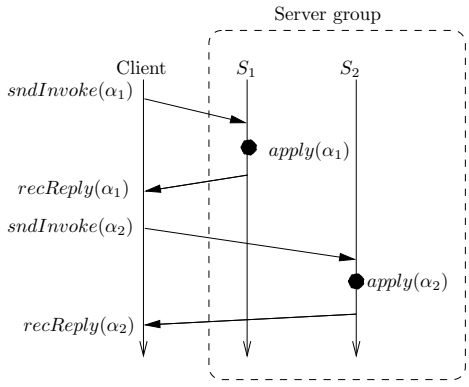


Figure 4. Client excluded ordering

common state for the system we can choose to replay the performed operations (that are stored in the logs of each replica). The replay starts from the last common state (i.e. from the state before the partition fault occurred) and iteratively builds up a new state. The question is only in what order should the operations be replayed? It would be tempting to answer this question with "the same order as they were originally applied". But there are two fallacies with this answer. First, it is hard to characterise the order they were originally applied in. Secondly, that order is not necessarily the best one in terms of maximising system availability.

Let's assume that we have an asynchronous system where a number of servers are joined in a logical group that orders events according to some scheme (e.g. causal order). Moreover, events taking place outside of this group (i.e., at the clients) are not covered by the ordering scheme. This is not an unreasonable approach. For crash tolerance it is enough with such a solution, since clients rarely need to be crash-tolerant. Now consider the scenario in Figure 4. Since no messages have passed between S_1 and S_2 in this scenario, there is no way for them to determine the temporal order between events $apply(\alpha_1)$ and $apply(\alpha_2)$.

Now imagine that S1 and S2 have performed these operations in degraded mode and reconciliation takes no account of client perceived order. Then α_1 and α_2 can be replayed in arbitrary order.

This completely violates what the client can reasonably expect from the system. The client *knows* that α_1 was executed before α_2 since the reply for α_1 was received even before α_2 was invoked. Therefore we use this expectation to define an ordering relation. $\alpha_1 \rightarrow \alpha_2$ iff $recReply(\alpha_1) < sndInvoke(\alpha_2)$, where $<$ denotes how the events are ordered by the local clock at the client node, $recReply(\alpha_1)$ denotes the event of receiving the reply for α_1, and $sndInvoke(\alpha_2)$ denotes the event of sending the invocation for α_2.

This induced order can be captured by a client side interceptor within the middleware, and reflected in a tag for the invoked operations. In Section 3.1.1 we describe the implementation of such a mechanism. Note that our proposed ordering is weaker than causal order. It builds

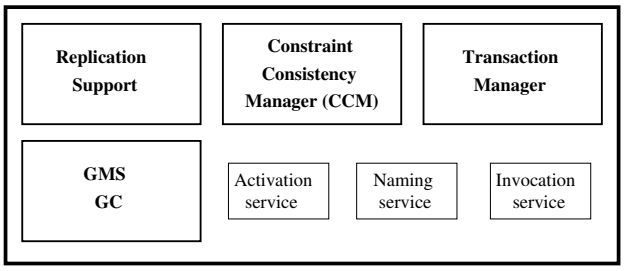


Figure 5. Architecture overview

on the assumption that clients are independent from each other. If clients communicate, causal ordering can replace the implemented order.

3. ARCHITECTURE AND COMPONENTS

Our additions to a generic middleware can be divided into seven distinct components (see figure 5). The smaller boxes represent components that presented fewer challenges in a CORBA setting. The four larger boxes will be the main focus of this section. It might be worth mentioning that the standard naming service provided with CORBA does not work in a partitionable setting. Instead, it is necessary to couple the naming service with the replication support to know where objects are replicated.

We implemented the group membership (GMS) and group communication (GC) component by encapsulating the Spread Toolkit [32] within our middleware. This service notifies topology changes to the other components in the system and provides ordered reliable multicast primitives to the replication and transaction managers. We now proceed by describing the three major components.

There are relatively few CORBA dependencies in our middleware extension. This allows it to be adopted for a wide range of middleware specifications. The main requirement on the middleware is that it must support some kind of interception mechanism (e.g., like those available in EJB, .NET, and through Java reflections), so that requests can be redirected, logged, and tagged with additional information.

3.1. Replication support

The replication service in the CORBA implementation consists of a replication manager, a replication protocol called P4Log (see [4, 7]) and a reconciliation protocol (CS) as shown in Figure 6. The figure also shows the control flow of a client invocation. The invocation is intercepted in a CORBA interceptor that passes the control to the replication manager. The replication manager just forwards the request to the replication protocol. The replication

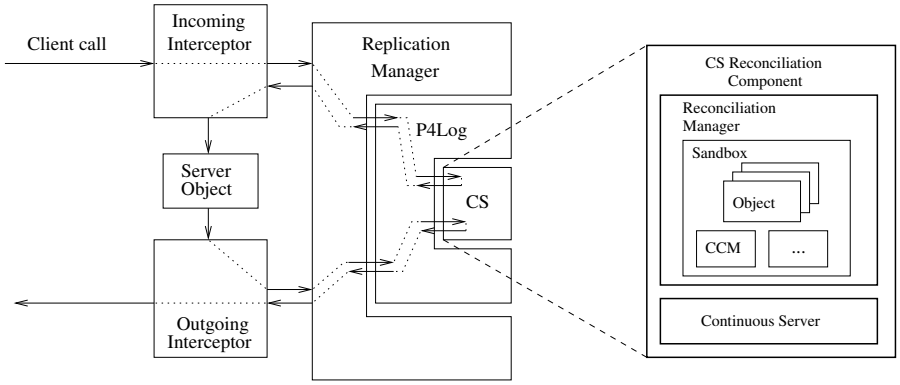


Figure 6. CORBA invocation

protocol can then decide if the request should be forwarded to the primary replica or to some replica that temporarily acts as primary during a network partition. Moreover, the object is registered as a transactional resource and the preconditions are checked. The request is also passed on to the CS protocol that may record the state of the target object if the operation is the first to be performed on that object in the degraded mode.

When the operation has been applied by the server object, it is once again passed through all the replication components. This time to perform logging (and possible interaction with the reconciliation manager which is described in Section 3.1.1) and update propagation. Updates are propagated via spread messages.

The replication manager is responsible for keeping track of the system modes, i.e. normal, degraded, reconciliation and the final installation of the state after full recovery from the partition. The manager is registered as a listener to events from Spread. When the changes in the network occur, the replication manager determines if the replication protocol should be notified that it should enter degraded mode, or in the case of a node re-joining, it should commence the reconciliation process.

3.1.1. Continuous service (CS) reconciliation component

The reconciliation component is composed of two sub-components: a continuous server and a reconciliation manager. The continuous server encapsulates the application objects and is responsible for logging all requests that are accepted during degraded mode. The reconciliation manager performs the actual reconciliation work based on the collected logs. We will proceed to give an informal description of the behaviour of these components (shown in algorithms 1 and 2, respectively). In [2] the algorithms are formally presented as timed I/O automata, and their correctness is shown.

Algorithm 1 Continuous Server

On reunify:
Send all logs to Reconciliation Manager(s)

On operation invocation:
If not stopped, apply operation
Check consistency, abort if not consistent
Send log to Reconciliation Manager
Suspend reply until later

On receive getState:
Send last stored object state
(from normal mode)

On receive logAck:
Send suspended replies to client

On receive stop:
Stop accepting new operations
Send stopAck

On receive install:
Change state of local objects to received state

Algorithm 2 Reconciliation Manager

On reunify:
Elect which node acts as recon. manager
Determine which objects to reconcile
Send *getState* request to servers

On receive log:
Add log to *opset*
Send *logAck* to server

On receive state:
Create object in sandbox environment

If opset not empty and states received:
Replay first operation in *opset* in sandbox
Check consistency, abort if not consistent

If opset empty and states received:
Send stop message to all servers

On receive stopAck:
Wait for opset to become empty
Send out new state to all servers

When the reconciliation begins, the continuous servers send the collected logs to the reconciliation manager. The manager answers these log messages by sending *logAck*, and puts the logs in an ordered set *opset* (according to the order discussed in Section 2.2). During the reconciliation phase the servers allow new operation requests and apply them provided that the constraint consistency requirements are met. However, the reply to the client is not sent until the a log message has been received by the reconciliation manager.

The manager exists in every node, but at each reconciliation there is an election process to make sure that only one reconciliation manager is active. After being elected it sets up copies of all the application objects in a sandbox environment. This is done by sending *getState* messages to the continuous servers. Once all states have been received, the manager starts replaying the log messages in the sandbox environment.

Consistency is maintained in the sandbox environment by checking the same constraints as are normally active in the system. When all logs have been replayed, the manager sends out a *stop* message, indicating that the reconciliation process is to end soon. The servers stop accepting new requests as soon as they receive a *stop* message from the manager, this block lasts until a new state is received and the system reenters normal mode.

Since the CS reconciliation protocol needs to keep track of the order in which operations are to be replayed, a mechanism is needed that makes ordering explicit. This ordering can easily be implemented by tagging each operation at the client side with the operations that must be executed before it. Since the operations "before" are those that the client has received a reply for when the operation was invoked, we must keep track of operation invocations as well as replies. We achieve this by introducing an *order manager* that is called from the interceptors at the client side. That is, there is an order manager at each client responsible for creating

the operation tags for each operation. It must also be called each time a reply is received to compute the tags correctly.

Note that the order manager only needs to maintain a local state for each client. Therefore, no communication is required between different managers. The state contains information on sent and received messages for one particular client.

For the application that runs inside the sandbox it appears as the real environment but no changes in the sandbox are committed until the end of the reconciliation phase. All objects that are to be involved in the reconciliation are registered in the sandbox. The arguments passed in this call are the state of the object, the object reference and the class name. The latter is needed to create new instances of the object inside the sandbox environment.

The sandbox is initialised with its own ORB and interceptors are used to check integrity constraints in the same way as in the normal system. However, the mechanisms in the sandbox interceptors are much simpler since everything is performed on one node and information does not need to be propagated throughout the system. A simple transaction mechanism was used so that operations could be aborted if integrity constraints are violated.

During the reconciliation phase the reconciliation manager replays all the logged operations by invoking them in the sandbox environment. The sandbox is responsible for checking consistency constraints and an operation is applied to the internal state only if all constraints are satisfied. When the reconciliation manager has finished invoking requests it will fetch the resulting object states. These states are then installed at the continuous servers and the system can go back to normal mode.

3.1.1.1. Complexity The above algorithms are online algorithms, since new input continuously arrives during their execution. However, we can characterise the time taken to perform the reconciliation to be $O(n \cdot m \cdot c)$, where n is the number of log elements, m is the maximum number of predecessors to a given operation, c is the maximum number of constraints for each operation. This is of course assuming that each constraint takes constant time to evaluate. However, the number of operations is in turn dependent on the length of the reconciliation, since new operations are accepted continuously. In [2] we characterised an upper bound on the time required for reconciliation. The linear complexity in the size of the logs is mainly due to the fact that the algorithms do not try to optimise the number of accepted operations. An optimal algorithm would need to either have knowledge of the application semantics, or need to perform an exhaustive search of all possible execution interleavings. In [3] we presented an algorithm that used operation profiling to heuristically achieve slightly better results than just randomly choosing the order.

3.2. Transaction manager

In our system we have used the concept of transactions to ensure integrity of data. Of particular importance was the all-or-nothing atomicity property of transactions, which guarantees that either updates on all objects succeed, or none take effect.

Transactions are used by the constraint consistency manager. Before any update takes place, a transaction is initiated. When updates have been performed, consistency of the state is

Algorithm 3 Group-per-transaction strategy

On begin transaction (at node i):
Generate globally unique transaction ID *TxID*.
Join group *TxID*.

On commit transaction:
Broadcast message commit to group *TxID*.

On receipt of message commit to group *TxID*:
Notify transactional resources associated with *TxID* to commit.

On rollback transaction:
Broadcast message rollback to group *TxID*.

On receipt of message rollback to group *TxID*:
Notify transactional resources associated with *TxID* to roll back.

On group *TxID* membership change (partition):
Notify transactional resources associated with *TxID* to roll back.

checked and if it is found to be inconsistent the transaction is rolled back. Otherwise, the transaction is committed.

One approach to implement distributed transactional behaviour would be using an off-the-shelf implementation of a transaction manager. However, this design would not have been partition-tolerant, since the traditional two-phase commit calls for a single transaction manager. In case of partitions the transaction manager would by definition end up in one of the partitions. In other partitions, transactions would be unable to continue until the transaction manager is available again.

Another issue with an off-the-shelf transaction manager is that its failure detector is not the same as the failure detector used for the rest of the system, in particular components that use GC/GMS for communication. Thus, even if fault detector timeouts were equally configured, race conditions would lead to situations where transaction manager would commit, whereas group communication messages would not be delivered, and vice-versa.

Therefore, we have decided to implement a transaction manager that makes use of the GC/GMS communication infrastructure to overcome the issues presented in the above two paragraphs. In this section, design and protocol of the transaction manager are informally explained.

In our implementation, transaction manager is a distributed service executing on all nodes of the system. Instances of the transaction manager exchange messages through group communication to decide the outcome of a transaction. This design addresses both of the aforementioned weaknesses of the off-the-shelf transaction managers: since transaction manager is distributed across all nodes, there wont be partitions with no transaction manager; and because GC/GMS is used for communication, the fault detector embedded in GC/GMS is used by the transaction manager as well, which is consistent with the other components in the system.

Algorithm 4 Single group strategy

On begin transaction (at node i):
Generate globally unique transaction ID $TxID$.
Associate running thread with the transaction.
Broadcast a message $join(i, TxID)$

On receipt of message $join(i, TxID)$**:**
Add $TxID$ to the set of transactions executing at node i:
$transactions(i) := transactions(i) \cup \{TxID\}$

On commit transaction:
Broadcast message commit($TxID$)

On receipt of message $commit(TxID)$**:**
Notify transactional resources associated with $TxID$ to commit.

On rollback transaction:
Broadcast message $rollback(TxID)$:

On receipt of message rollback($TxID$**):**
Notify transactional resources associated with $TxID$ to roll back.

On group membership change (partition):
Initialise a list of transactions to rollback: $toRollback := \emptyset$
For each node j that is no longer present in the view:
    Add all transactions in $transactions(j)$ to the list $toRollback$:
For each $TxID$ in toRollback:
    Notify transactional resources associated with $TxID$ to roll back.

Two strategies for associating transactions with communication groups have been designed and implemented: (1) Group-per-transaction: for every transaction, a separate communication group is created (shown in Algorithm 3). All nodes that are involved in the transaction join its respective group. (2) Single group: a single communication group is used for exchanging messages related to all transactions (shown in Algorithm 4).

The two protocols differ mainly in their use of the group communication infrastructure: whereas the second approach generates more network traffic in cases where not all nodes have joined a transaction, the first approach results in a large number of groups with a relatively short life cycle. In our experiments we have used the second policy since our scenarios involved a small number of nodes that were almost always all involved in the transaction.

3.2.1. Commit protocol

The protocol consists of the following steps:

Begin transaction. When the transaction begins, it generates a transaction ID, and associates the running thread with the transaction. It then signals other transaction managers that it has joined the transaction.

In case of *group-per-transaction* strategy, the transaction manager joins the transaction's communication group. With the *single group* strategy, the transaction manager broadcasts

a message to the group, stating that it has joined the transaction. Transaction managers to which this message is delivered must then explicitly keep track of the transaction manager nodes that have joined a particular transaction.

Commit. When the transaction is ready to commit, one of the nodes (typically the one that initiated the transaction) broadcasts a *commit* message to the group. The broadcast is delivered to the sending transaction manager as well. When a *commit* message is delivered to a transaction manager, the transaction manager notifies all the transactional resources registered with the transaction to commit the work they have done in the context of the transaction. Atomicity of the commit is guaranteed by the group communication semantics of a broadcast, which delivers the message either to all nodes, or to none.

Rollback. Transaction can be rolled back because of several reasons:

- If the initiator of the transaction decides to rollback the transaction. For example, in our system this occurs if not all constraints are satisfied.
- If one of the nodes in the transaction can not perform its work.
- If a network partition is detected.

When a transaction manager instance is explicitly told to roll back, it broadcasts a *rollback* message to the transaction's group (or the single group). Upon receiving this message it notifies the transactional resources registered with the transaction to discard the work. If a fault occurs, the group communication infrastructure delivers a group membership change message. This message states which nodes are still remaining in the group. The transaction manager then finds all the transactions which are also running on the nodes not present in the current view any more. Transaction manager notifies transactional resources to abort these transactions.

3.2.2. Scalability

The algorithms of the transaction manager's commit protocol described above have a time complexity of at worst $O(N)$, where N is the number of nodes participating in the transaction.

Such is the case with the single group transaction strategy, where each node must keep track of all the other nodes that have joined the transaction: first N nodes must be added to a set (e.g., a hash-table, which is an $O(1)$ operation), and in case of a partition the set must be checked if any of the nodes that have left the view participated in the transaction (up to N repetitions of a $O(1)$ operation, which is $O(N)$).

However, reliable broadcasts provided by GC/GMS infrastructure are not $O(1)$ operations themselves! Furthermore, in terms of actual performance it is these broadcasts that are responsible for most of the latency. Therefore, it is important to look at scalability properties of the GC/GMS infrastructure, in our case Spread.

In a technical report [1], the authors of Spread report on the following behaviour of a Spread-based system with up to 20 physical nodes and up to 1000 participating processes:

- Due to the use of a skip list data structure that provides $log(N)$ access, the number of groups in the system does not affect the latency of message delivery *much*. (Due to limited amount of data presented in the technical report, a more precise statement is difficult to make.)

- The size of membership notification message increases linearly, as does the number of notifications per Spread daemon. Consequently, the dependency of join/leave latency on the number of groups is $O(N^2)$.

These observations have the following effect on scalability of our transaction manager implementation:

- In the group-per-transaction strategy, the commit protocol is simpler ($O(1)$), but the Spread's join/leave mechanism exhibits $O(N^2)$ behaviour.
- In the single group strategy, the commit protocol is more complex ($O(N^2)$), but the expense of Spread's join/leave mechanism is avoided.

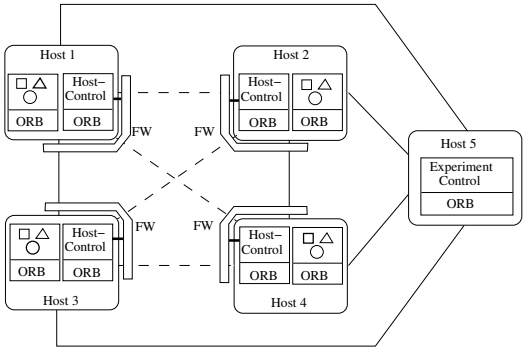
In the text above, only latency was considered. It should be noted that the number of messages exchanged over the network might grow up to factor N faster than the latency, as some of the messages might need to be delivered to many or all nodes. Though this puts additional stress on the routers in the network, it does not incur additional CPU cost or latency.

3.3. Constraint consistency manager

The Constraint Consistency Manager (CCM) is responsible for checking constraint consistency during operation invocations and making sure that no operations with critical constraints are accepted during degraded mode. Moreover, the reconciliation protocol needs to use the CCM while restoring consistency. The constraints managed by the CCM can be classified in different ways. Recall that there are critical and non-critical constraints, but one can also categorise the constraints according to when they are checked as pre- and post-conditions, and invariants. Depending on the type of constraint and whether the constraint is associated with a given object, method or class, the CCM should correctly evaluate the appropriate constraints.

The component contains a constraint repository in which all constraints are stored. The constraint repository is replicated on all nodes. Moreover, for each constraint additional information such as affected objects and methods are stored. This way, the CCM only needs to check a small subset of the constraints for every method invocation, thereby reducing unnecessary computations. If a constraint is violated, or if it is critical and evaluated in degraded mode, then the CCM throws a CORBA exception. This is propagated back to the replication protocol that aborts the current transaction, and to the client informing that the operation could not be applied.

When an operation with a non-critical operation is invoked in degraded mode, it is not necessarily a good idea to accept it. For example, a booking system where there is a non-critical constraint saying that the number of bookings should not exceed the number of seats. If there are many free seats, then it is probably safe to make a booking. But if the number of booked seats are close to the limit, then the risk of violation is high. Therefore, it might not be worth the risk to accept any more bookings for this flight. This type of reasoning can be done as part of the constraint management and is localised in the Negotiation Handler subcomponent. The implementation allows the application to register an application specific negotiation handler.



Number of runs	20
Number of objects	50
Number of constraints	50
Critical constraints	10 %
Number of nodes	4
Number of clients	2
Number of operations	200
Experiment duration	68-87 [s]
Load	6 [ops/s]

Figure 7. Deployment of the test environment.

Having described the components of our CORBA extension to trade consistency for availability, we now go on to present the results of performance evaluation of the system.

4. EVALUATION

Avizienis et al. [5] define availability as "readiness for correct service". Classic availability metrics [18] state that in order to measure the availability of system one should use the following formula: $A(t) = \frac{\sum_i u_i}{t}$. Here, u_i are the periods of time where the system is available for operation. However, even in traditional systems it is not always trivial to decide what constitutes a mode where the system is available for operation. Sieworek and Swarz [30] identified up to eight stages in response to a failure that a system may go through. These are: fault confinement, fault detection, diagnosis, reconfiguration, recovery, restart, repair, and reintegration.

We have in our work identified the need to differentiate between full and partial availability as well as apparent availability where operations are seemingly accepted but later revoked. A different approach to measuring availability, which we also employ in this work, is based on counting successful operations [36]. This approach allows differentiation between different types of operations.

4.1. Experimental Setup

Figure 7 shows the framework that was used for performing measurements. Four nodes were set up with a CORBA object (*host control*) that controlled the activities on the node. These could be remotely controlled from a fifth server that starts up the middleware, initialises server objects and starts up client objects. Each node was running Linux and by using the `iptables` firewall (FW in the figure) capability in Linux, the *host control* object could inject network

partition faults by dropping inbound packets. In the figure we illustrate the links that were blocked with dashed lines. The CORBA ORB implementation was JacORB 2.2.3. For the availability and reconciliation duration measurements, the experiment was run 20 times, each time with 200 operations. For the overhead measurements, there was a single run, with 100 operations (except for the measurements during reconciliation where more operations where needed to ensure enough data points). For each measurement we have indicated the the size of the 95% confidence interval relative to the mean.

Four dedicated workstation computers with the same hardware configuration were set up. Each computer had an *AMD Athlon*^{TM}*64 3000+* processor running at 1.8GHz (3620 BogoMIPS) and 1 GB of RAM. The computers were interconnected with a 1 Gbps Ethernet network. The installed operating system was *Scientific Linux 4* (Linux kernel version 2.6.9-34.0.1.EL). *Sun Microsystem*'s Java$^{\text{TM}}$version 1.6.0_01. The transaction manager was run with the single group policy.

4.2. Test application

To validate the algorithms and the middleware implementation we needed a test application. We constructed a synthetic application so that we could easily change the mix of constraint types and to perform trade-off studies. The application is composed of a set of objects, each managing an integer number. Possible operations are addition, subtraction, multiplication, division, and setting of a value. An operation is applied to the current value with a random constant. There are also integrity constraints in the system expressed as: $n_1 + c < n_2$ where n_1 and n_2 are object values and c is a constant. In a sense this application resembles a distributed sensor network where the numbers correspond to sensors values.

Although the application is very simple, it is complex enough to give an indication of how the algorithms perform. One of the properties is that even if 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. This makes the reconciliation realistic in the sense that it is not trivial to construct a consistent state while retaining as many performed operations as possible. Moreover, the application allows key system parameters (e.g. ratio of critical constraints and load) to be changed and thus provides means to experimentally investigate their effect on availability.

In all of the experiments the application starts with initialising a number of servers with some default value. A set of constraints are created and added to the constraint store. The constraint parameters are chosen randomly with a uniform distribution. However, all constraints are created so that the initial state of the system is consistent.

4.3. Baseline

As a baseline in our experiments we have used a pessimistic replication protocol. By pessimistic we mean that it rejects all invocations during a network partition since it cannot make sure that nothing is changed in the other partitions. The reason for not using a majority partition approach as a baseline is that the performance of such a replication protocol is highly dependent

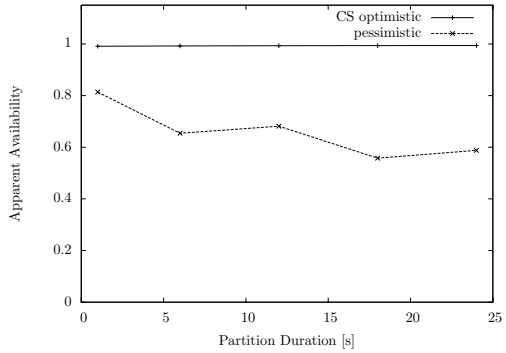


Figure 8. Apparent Availability vs. Partition Duration

on whether there exists a majority partition or not. If a majority partition exists, then the performance is very good, but when no majority exists it is equal to the pessimistic protocol.

4.4. Results

In this section we present the results of the performance measurements. First, we show how much is gained from acting optimistically in presence of partitions using the CS reconciliation protocol, both in terms of apparent availability as well as number of accepted operations. Secondly, we have measured the overhead incurred by our solution by measuring the throughput. Finally, we investigate the effect of load on the reconciliation protocol.

4.4.1. Apparent Availability

Apparent availability is a metric we use to reflect the client's conception of the system to be apparently available. We measure this by repeating the following scenario 20 times and measuring the duration of service over the experiment interval (irrespective of later revocations). First, a number of operations were performed in the normal mode. Then a partition fault was injected followed by further operations being applied in both partitions. After a predetermined time interval the injected fault was removed. This causes the reconciliation process to start. Invocations were continued during the whole reconciliation stage and a while into the new mode. Figure 7 shows the parameters used. The duration of the experiment depends on the duration of the fault.

In Figure 8 the apparent availability is plotted against the partition duration. The 95% confidence interval of these values are within 0.1% from the mean. As expected, the optimistic protocol maintains a constant high availability of approximately 98%. The unavailability is caused by the short period in which the reconciliation protocol awaits the final operations before installing a new state.

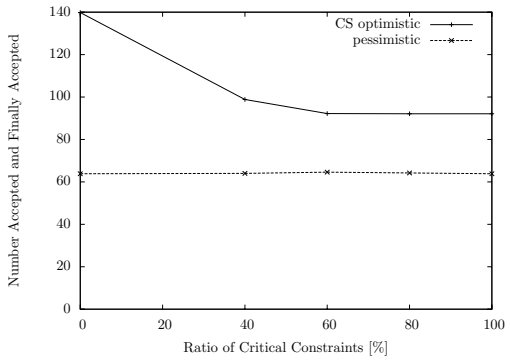


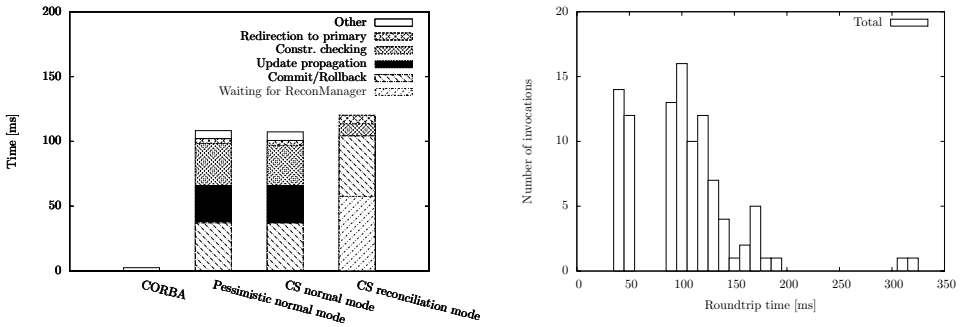
Figure 9. Accepted Operations vs. Ratio of Critical Constraints

The availability of the pessimistic protocol is a bit more interesting. We would expect the availability to start at 100% and then drop off to 70% for 24 second partition durations since the experiment time is then 87 seconds (i.e., 24s/87s). However the performance is worse than that. The reason for this is that Spread takes some time before registering that the network has reunified. The timeout in the Spread implementation that we used was set to 10 seconds. This timeout is also the cause of the stair-like shape of the graph. The availability does not drop continuously as the fault duration increases. Instead there is a drop every 10-12 seconds that corresponds to the configured timeout interval of Spread. We have shortened this interval from the default which is 60s. A too short interval would lead to high overhead.

4.4.2. Operation-based availability

Apart from the apparent availability it is equally important that the work performed during the period of degradation is saved (i.e., performed operations not later revoked) when the system is later reunified. The more provisionally accepted operations are revoked, the lower is the number of finally accepted operations.

Figure 9 shows the number of operations accepted in normal mode plus the number of finally accepted operations from the degraded mode plotted against the ratio of critical constraints. The parameters are the same as in the previous experiment except that the fault duration is kept constant at 10 seconds. The 95% confidence intervals are within 7% for all measurement points except for CS optimistic at 40% where the interval is within 21% of the mean. Remember that if a critical constraint needs to be checked for a given operation, then that operation cannot be performed optimistically. Thus the higher the ratio on the x-axis is, the less likely is it that acting optimistically pays off. This fact is reflected in the dropping of total number of accepted operations for the optimistic protocol. However, it still performs better than the pessimistic protocol. Here we observe an unexpected behaviour of the implementation. The more spread messages are sent the faster can Spread detect that the network is reunified. Thus, where the



(a) Average roundtrip times

(b) Distribution of roundtrip times for CS normal mode

Figure 10. Overhead

CS protocol benefits from faster detection of reunification the pessimistic approach loses since it does not send any spread messages during the degraded mode.

4.4.3. Overhead

Fault tolerance almost always comes with a cost in increased overhead. This is true also for our solution. However, the optimistic continuous service reconciliation approach turns out to be not very expensive at all compared to the pessimistic approach. As can be seen in Figure 10(a) the roundtrip time of the CS optimistic protocol and the pessimistic protocol are quite close to each other. The bar marked with "CORBA" is the roundtrip time of a plain CORBA middleware without any fault tolerance or constraint checking at all. So it is clear that the big cost associated with our middleware extension is associated with replication (update propagation), constraint checking, and transaction handling. These are quite complex operations that require several nodes to cooperate. The reason that constraint checking is an expensive operation is that reads are directed to the primary replica. This could obviously be improved by employing local reads. During reconciliation, the main cost is caused by having to wait for acknowledgement from the reconciliation manager before sending a reply.

Figure 10(a) only shows the average roundtrip times. The 95% confidence intervals of the total roundtrip time are within 39% (corresponding to 2.3ms), 21%, 26% and 13% of the average for the respective bars. These relatively high deviations were investigated by studying the worst case. When we studied the results from the previous experiment in more detail it became apparent that while most of the invocations completed within 150ms there were some exceptions that took a significantly longer time to complete. In Figure 10(b) one can clearly see this phenomenon. There are no operations taking more than 200ms except for a small number that require more than 300ms. The cause is that some spread messages take significantly longer time to be delivered.

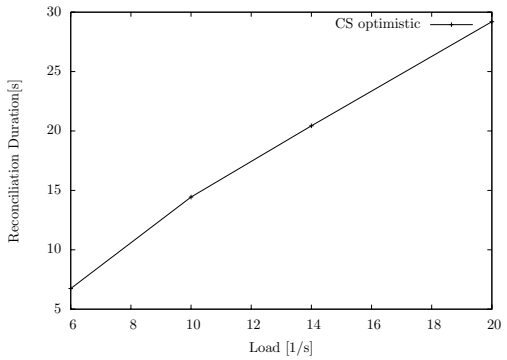


Figure 11. Reconciliation Duration vs. Load

4.4.4. Effect of Load

We have also investigated the effect of load on the reconciliation protocol. Earlier work shows that termination of the reconciliation protocol is [2] conditional on the minimum invocation inter-arrival time being greater than the average replay time during reconciliation. However, even if we achieve termination, the reconciliation time might still be very long. Therefore, we are interested in measuring the actual reconciliation duration as a function of system load.

Figure 11 shows that the duration increases as the load increases. The same parameters were used as in the availability measurements, but with a varying load. The 95% confidence intervals are within 9% from the mean. The experiment was run for higher loads than 20 as well, but in those cases the reconciliation did not finish during the length of the experiment. The rate at which operations were replayed was in the range 10-15 per second. So it is consistent with the termination proof [2] that with higher loads, termination may not happen. Again it is important to realize that the sandbox replay service has not been optimised with regards to performance.

4.5. Scalability

Due to the nature of our experiment setup we have not been able to perform extensive scalability tests. Thus, we do not claim to have verified that our extensions work in a large-scale setting. However, we have already discussed the complexity of the algorithms in sections 3.1.1.1 and 3.2.2 (which also discussed the scalability of Spread). These results indicate that it should be possible to scale up provided that the rate of reconciliation is higher than the load of incoming operations (see the previous subsection). Naturally, the underlying group communication layer (in our case Spread) is a limiting factor since reliable broadcasts are themselves expensive. We rely on such messages in replication, reconciliation and transaction management.

5. RELATED WORK

Coan et al. [9] define an availability metric for partitioned networks. The basic idea is to measure the proportion of operations that succeed over the total number of submitted operations. This can then be divided in two parts, availability given by performing update transactions, and availability due to performing read transactions. There is a major problem with this metric when combined with integrity constraints. If an operation or transaction is rejected due to an integrity constraint, then this is part of the normal behaviour of the system and should not be seen as reduction of availability. On the other hand, if we consider rejected operations as successfully completed then we have another problem. If we optimistically accept an operation without being able to check for consistency and later revoke it, should it still be counted as successfully completed?

In 1978, Gray[15] described the design of a two-phase commit protocol that enables transactional properties in a distributed environment. The two-phase commit protocol is frequently found in commercial implementations of distributed transaction systems, e.g., *JBoss Transaction Service*, *Microsoft Distributed Transaction Coordinator* and *Tuxedo*. Recently, Gray and Lamport [17] have proposed a distributed algorithm based on Paxos consensus which avoids the need for a single transaction coordinator.

Group communication and transactions are both well known means for achieving fault tolerance in distributed systems. In [8] Birman showed how group communication can be used for replication of state. Schiper and Raynal [29] then proposed using group communication as the framework for implementing transactions, with a practical implementation of an optimistic atomic commit described by Kemme et al. [20].

The CORBA platform has been a popular platform for research on fault-tolerant middleware. Felber and Narasimhan [13] survey some different approaches to making CORBA fault-tolerant. There are basically three ways to enable fault tolerance in CORBA, (1) by integrating fault tolerance in the Object Request Broker (ORB), examples include the Electra [22] and Maestro [35] systems (2) by introducing a separate service as done in DOORS [25] and AqUA [10] (3) by using interceptors and redirecting calls. The interceptor approach is adopted in the Eternal system [24] which is also designed to be partition-tolerant (discussed below). An implementation of standard FT-CORBA that combined separate services and interceptors to provide crash tolerance was presented in [33].

There are a number of systems that have been designed to provide partition tolerance. Some are best suited for systems in which communication fails only for short periods. None of the systems have proper support for system-wide integrity constraints.

Bayou [34, 26] is a distributed storage system that is adapted for mobile environments. It allows provisionally (tentatively) accepting updates in a partitioned system and to finally accept (commit) them at a later stage. There is support for integrity constraints through preconditions that are checked when performing a write. If the precondition fails, a special merge procedure that needs to be associated with every write is performed. Bayou will always operate with tentative writes. Even if the connectivity is restored, there is no guarantee that the set of tentative updates and committed updates will converge under continued load.

The Eternal system by Moser et al. [23] is a partition-aware CORBA middleware. Eternal relies on Totem for totally ordered multicast and has support both for active and passive

replication schemes. The reconciliation scheme is described in [24]. The idea is to keep a sort of primary for each object that is located in only one partition. 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. The problem with this approach is that it cannot be combined with constraints. The reason is that one cannot assume that the state which is achieved by combining the primaries for each object will result in a consistent state on which operations can be applied.

Jgroup supports programming partition-aware systems using a method that the authors call enriched view synchrony. Helvik et al. [19] have performed thorough experimental study of Jgroup/ARM which is a system that reacts on faults and reconfigures the system so that there is always the correct configuration of replicas in every part of the system. Unfortunately as it is only a simulation it is not possible to determine overheads that the system causes.

Singh et al. [31] describe an integration of load balancing and fault tolerance for CORBA. Specific implementation issues are discussed. They use Eternal for fault tolerance and TAO for load balancing. It seems that the replication and load balancing more or less cancel each other out in terms of effect on throughput. Only crash failures are considered.

Several replicated file systems exist that deal with reconciliation in some way [28, 21]. Balasubramaniam and Pierce [6] specify a set of formal requirements on a file synchroniser. These are used to construct a simple state-based synchronisation algorithm. Ramsey and Csirmaz [27] 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.

6. Conclusions

Despite increased levels of connectivity and extensibility it is still a challenge to uphold totally connected networks. The above factors also bring with them more complexity and higher frequency of misconfigurations and local overloads, that can result in network partitions. Also, higher connectivity is a fertile ground for subversive activities in which overloads can be created through attacks. This paper has addressed the challenge of upholding availability despite partitions, albeit for a subset of services in distributed object systems.

The main thesis has been that middleware services can provide for continuous service of incoming operations in an optimistic mode, at the cost of some operations being revoked later. An implementation in a CORBA environment is used to demonstrate the feasibility of the approach and the ensuing costs in terms of performance overheads. Since the extensions depend little on CORBA-specific services, we believe that these extensions can easily be adapted to other middlewares as well.

Using variations in a synthetic application in which the ratio of critical and non-critical operations can be changed the variations in applicability of the method were illustrated.

An excellent example in which networked applications may naturally suffer from frequent partitions is communication in disruption-tolerant networks (DTN) [12]. An interesting direction of work is to apply concepts analogous to consistency in DTN.

ACKNOWLEDGEMENTS

This work has been supported by European Community under the FP6 IST project DeDiSys (Dependable Distributed Systems, contract 004152). The development of the CORBA architecture was a team effort and included work by the members of the team at ITI the support of which is gratefully acknowledged. The second author was partially supported by University of Luxembourg during preparation of this manuscript.

REFERENCES

1. Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Jonathan Stanton. The spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Johns Hopkins University, Center for Networking and Distributed Systems (CNDS), 2004.
2. Mikael Asplund and Simin Nadjm-Tehrani. Formalising reconciliation in partitionable networks with distributed services. In M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems*, volume 4157 of *Lecture Notes in Computer Science*, pages 37–58. Springer-Verlag, 2006.
3. Mikael Asplund and Simin Nadjm-Tehrani. Post-partition reconciliation protocols for maintaining consistency. In *Proceedings of the 21st ACM/SIGAPP symposium on Applied computing (SAC'06)*, New York, NY, USA, April 2006. ACM Press.
4. Mikael Asplund, Simin Nadjm-Tehrani, Stefan Beyer, and Pablo Galdamez. Measuring availability in optimistic partition-tolerant systems with data constraints. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks (DSN'07)*, pages 656–665, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
5. A Avizienis, J.-C Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Dependable and Secure Computing*, 1(1):11–33, 2004.
6. S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking (MobiCom'98)*, pages 98–108, New York, NY, USA, 1998. ACM Press.
7. Stefan Beyer, Francesc D. Muñoz-Escoí, and P. Galdámez. Implementing network partition-aware fault-tolerant CORBA systems. In *Proceedings of the 2nd International Conference. on Availability, Reliability, and Security (ARES'07)*, Los Alamitos, CA, USA, 2007. IEEE Computer Society Press.
8. Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.
9. Brian A Coan, Brian M Oki, and Elliot K Kolodner. Limitations on database availability when networks partition. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing (PODC'86)*, pages 187–194, New York, NY, USA, 1986. ACM Press.
10. M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W.H. Sanders, D.E. Bakken, M.E. Berman, D.A. Karr, and R.E. Schantz. AQuA: an adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, pages 245–253, Los Alamitos, CA, USA, October 1998. IEEE Computer Society.
11. DeDiSys. European IST FP6 DeDiSys Project. http://www.dedisys.org, 2005–2007.
12. S. Farrell and V. Cahill. *Delay-and Disruption-Tolerant Networking.* Artech House, Inc. Norwood, MA, USA, 2006.
13. Pascal Felber and Priya Narasimhan. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers*, 53(5):497–511, 2004.
14. L. Froihofer, K.M. Goeschka, and J. Osrael. Middleware support for adaptive dependability. In *Proceedings of the ACM/IFIP/USENIX 8th Int. Middleware Conference.* Springer, 2007.
15. Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, Heidelberg, Germany, 1978. Springer-Verlag.
16. Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proccedings of the 1996 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 173–182, New York, NY, USA, 1996. ACM Press.
17. Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31(1):133–160, 2006.

18. Abdelsalam A. Helal, Bharat K. Bhargava, and Abdelsalam A. Heddaya. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.
19. Bjarne E. Helvik, Hein Meling, and Alberto Montresor. An approach to experimentally obtain service dependability characteristics of the Jgroup/ARM system. In *Proceedings of the Fifth European Dependable Computing Conference (EDCC'05)*, Lecture Notes in Computer Science, pages 179–198, Heidelberg, Germany, April 2005. Springer-Verlag.
20. Bettina Kemme, Gustavo Alonso, Fernando Pedone, and Andre Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the nineteenth IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, page 0424, Los Alamitos, CA, USA, 1999. IEEE Computer Society.
21. James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.
22. Silvano Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, Berkeley, CA, USA, June 1995. USENIX Association.
23. L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the eternal system. *Theory and Prectice of Object Systems*, 4(2):81–92, 1998.
24. P Narasimhan, L E Moser, and P M Melliar-Smith. Replica consistency of CORBA objects in partitionable distributed systems. *Distributed Systems Engineering*, 4(3):139–150, 1997.
25. B. Natarajan, A. Gokhale, S. Yajnik, and D.C. Schmidt. Doors: towards high-performance fault tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'00)*, pages 39–48, Los Alamitos, CA, USA, September 2000. IEEE Computer Society.
26. Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: replicated database services for world-wide applications. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop (EW 7)*, pages 275–280, New York, NY, USA, 1996. ACM Press.
27. Norman Ramsey and Elöd Csirmaz. An algebraic approach to file synchronization. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-9)*, pages 175–185, New York, NY, USA, 2001. ACM Press.
28. Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Conference Proceedings*, pages 183–195, Boston, MA, June 1994. USENIX.
29. Andre Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, 1996.
30. Daniel P. Siewiorek and Robert S. Swarz. *Reliable computer systems (3rd ed.): design and evaluation*. A. K. Peters, Ltd., Natick, MA, USA, 1998.
31. A. V. Singh, Louise E. Moser, and P. M. Melliar-Smith. Integrating fault tolerance and load balancing in distributed systems based on CORBA. In *Proceedings of the 5th European Dependable Computing Conference (EDCC'05)*, volume 3463 of *Lecture Notes in Computer Science*, pages 154–166, Heidelberg, Germany, 2005. Springer.
32. Spread. The spread toolkit. http://www.spread.org.
33. Diana Szentivanyi and Simin Nadjm-Tehrani. Middleware support for fault tolerance. In Qusay Mahmoud, editor, *Middleware for Communications*. John Wiley & Sons, 2004.
34. D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP'95)*, pages 172–182, New York, NY, USA, 1995. ACM Press.
35. Alexey Vaysburd and Ken Birman. The maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Prectice of Object Systems*, 4(2):71–80, 1998.
36. Haifeng Yu and Amin Vahdat. Minimal replication cost for availability. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing (PODC'02)*, pages 98–107, New York, NY, USA, 2002. ACM Press.