Managing Replicated Remote Procedure Call Transactions

WANLEI ZHOU AND ANDRZEJ GOSCINSKI

School of Computing and Mathematics, Deakin University, Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au

This paper addresses the problem of building reliable computing programs over remote procedure call (RPC) systems by using replication and transaction techniques. We first establish the computational model: the RPC transactions. Based on this RPC transaction model, we present the design of our system for managing RPC transactions in the replicated-server environment. Finally, we present some results of a correctness study on the system and two examples of the system.

Received 17 April, 1997; revised 30 August, 1999

1. INTRODUCTION

Two approaches are commonly used in supporting fault-tolerant computing. The first approach provides programming languages that are targeted at developing fault-tolerant systems. Typical examples of this approach are the Ada 95 [1], the fault-tolerant concurrent C [2], and the fault-tolerant version of the SR language [3]. The second approach provides a fault-tolerant computing toolkit or a model that can be used together with general programming languages and standard operating systems. Typical examples of this approach are the ISIS toolkit [4], the ARGUS system [5], the location-based replication paradigm [6], the fulfilment transactions approach [7], and the RPC transaction management system [8]. This paper follows the second approach.

Remote procedure call (RPC) is perhaps the most popular model used in today's distributed software development and has become a *de facto* standard for distributed computing. The use of RPC facilitates the building of distributed programs and removes concerns for the communication mechanisms from the programs that use remote procedures. Only fundamental difficulties of building distributed systems such as synchronisation and independent failure of components are left in RPC programming.

Many leading computer companies have agreed on a vendor-neutral distributed computing environment (DCE) architecture proposed by the Open Software Foundation [9]. This architecture is designed under the client/server model, and requires the interactions between its components to follow the RPC paradigm. Although the DCE architecture helps reduce the heterogeneity of server-access protocols and provides a limited fault-tolerant support in the service level, one important issue is still outstanding: the support for fault-tolerant computing from the RPC level.

Since fault tolerance is not provided in the RPC level, system services and user applications running on DCE have to employ their own mechanisms for dealing with reliability and availability of the system. This limitation has resulted in a number of problems such as (1) adding another dimension of difficulties (dealing with fault tolerance) in software development; (2) repeated development of fault-tolerant mechanisms in services and applications; and (3) less efficient fault-tolerant mechanisms since they are running on higher protocol levels.

For example, the Directory Service of the DCE uses a primary copy and a number of read-only copies to provide a distributed and replicated repository for information on various resources of a distributed system. This mechanism has the inconsistency and reconfiguration problems in the case of failures, as described in [10] and [11]. Reference [11] proposes an extension of the DCE Directory Service to provide a better fault-tolerant service. However, it can only solve the fault-tolerance problem on one service.

It has been suggested that the use of replication and transaction techniques can provide an environment for developing reliable programs [12]. Replication is the key to providing high availability, fault tolerance, and enhanced performance in a distributed system. However, although considerable research efforts have been directed towards the design of replication-control protocols, replication is still viewed as a 'necessary evil' [6]. Reference [13] gives a comprehensive overview of replication techniques and annotated bibliographies of selected literature on replication techniques and example systems.

Transaction management is a well-established concept in database system research. A transaction is defined as a sequence of operations over an *object system* (a system with an associated collection of objects, where an object can be a database file, an entry of a database file, or can model a real-world entity such as printers or actuators of a control system), and all operations must be performed in such a way that either all of them execute or none of them do [12]. Transactions are used to provide reliable computing systems and a mechanism that simplifies the understanding and reasoning about programs.

There have been some efforts to combine two of the three techniques—RPC, replication, and transaction—together to achieve reliable computing. However, none of the existing systems/proposals are completely satisfactory.

The ISIS toolkit [4] is a distributed programming environment, including a synchronous RPC system, based on virtually synchronous process groups and group communication. ISIS combines RPC and replication techniques to achieve the goal of developing reliable programs. A special process group, called a *fault-tolerant process group*, is established when a group of processes (replicated servers and clients) is cooperating to perform a distributed computation. Processes in this group can monitor one another and can then take actions based on failures, recoveries or changes in the status of group members. A collection of reliable multicast protocols is used in ISIS to provide failure atomicity and message ordering. The drawback of developing fault-tolerant programs using ISIS is the performance penalties incurred by using process groups.

The location-based paradigm for replication proposed by Triantafillou and Taylor [6] addresses the problem of combining reliability with performance issues. It uses replication and transaction techniques to develop reliable programs. The proposal provides reliability similar to quorum-based replication protocols but with transaction delay similar to a one-copy system. However, the proposal cannot deal with network partitions properly. Another paper by the same authors [14] addressed the issue of achieving high availability in a partitioned distributed system through the use of transaction and replication techniques.

The RPC transaction management system proposed by Zhou and Molinari [8] uses RPC and transaction techniques to develop reliable programs. RPCs are grouped into transactions that are guaranteed to be atomic. However, the proposed system does not work in a replicated environment.

The Coda file system [15] supports the replication of file volumes, managing the resulting multiple-update problem even in the presence of server failures. The replication of file volumes produces a fault-tolerant service. The most successful feature of the Coda system is its support for disconnected operations. However, the reintegration of files after a disconnection involves manual intervention in the case of conflicting updates.

The fulfilment transactions approach adopted by the Totem group [7] addresses the issue of reliable transaction management in a replicated environment, through the use of the Totem multicast group communication system [16]. Transactions normally commit even in the presence of system failures (e.g. during a network partition), and corresponding fulfilment transactions are generated to record the details of these affected transactions. These fulfilment transactions will then be processed by the system when the failure is recovered in order to bring all replicas to a consistent state. However, fulfilment transactions are application-specific and in some cases, human intervention is needed to reconcile the inconsistencies.

The purpose of this paper is to combine replication, transaction management and RPC together to form a reliable and efficient distributed computing environment. Fault-tolerant programs developed in our environment should be able to tolerate single failures such as a server failure, a site failure or even a network partition without involving manual intervention. These programs should also be efficient and should not incur too much overhead compared with a non-replicated system when there are no component failures.

The remainder of the paper is organized as follows. Section 2 introduces the replicas, the RPC model, the transaction model and the failure semantics. Section 3 presents the model and the algorithm for transaction management. Section 4 describes the replica management. Section 5 discusses the conflict-resolution algorithms. Section 6 presents some results of a correctness study of the proposed system. Section 7 describes an illustration example and an application example of the system. Section 8 concludes the paper.

2. SYSTEM MODELS

2.1. Replicas

A distributed system with replicated servers consists of many sites interconnected by a communication network. A service is provided by a group of replicated servers (called *replicas*) executing on some sites. These replicas manage some common data objects that can be shared by many clients. For simplicity, we assume that each replica knows the location of other replicas that store the same data objects. This assumption can be loosened if a replication directory service is used.

When requesting a service, a client specifies the service through, say, an attributed name [17]. The exact location of the service and the server that provides such a service will be determined by the system.

We model a service as a set of replicas $S = \{S_1, S_2, \ldots, S_i, \ldots, S_n\}$, where $i = 1, 2, \ldots, n$, are called the *sequence numbers* of these replicas. Each replica S_i manages a set of data objects $O^i = \{d_1^i, d_2^i, \ldots, d_m^i\}$. The consistency constraint requires that for $i, j = 1, 2, \ldots, n$ and for each service S, $O^i \equiv O^j$.

2.2. The RPC model

Each replica in our system provides a number of remote procedures that can be called by clients for processing the data objects managed by the replica. We use P to denote the set of all remote procedures provided by all replicas of the system:

$$P = \{p \mid p \text{ is a remote procedure of the system}\}.$$

Our RPC model has the exactly-once call semantics in the absence of failures and the at-most-once call semantics otherwise [17]. In particular, after we make an RPC the call may return successfully or fail. There may be several reasons for the failure of a call such as 'object not free', 'server error' or 'communication error.' With some error conditions it is clear to the client that the procedure was

not executed, while with other error conditions it is not clear if the procedure was executed or not. For example, on receiving an 'object not free' or a 'server error', we are sure that the RPC was not performed. But on receiving a 'communication error', the client will not be able to tell if the call was performed or not, because we do not know whether the error happened before or after the calling request arrived at the destination host.

We define the effects of an RPC as the processing of one data item of the object system. Hence we can abstract an RPC as a mapping of the following *type*:

$$c : P \times O \to \{OK, FL, US\},$$

where O is the union of all data objects managed by all services of the system. The values of the target set have the following meaning:

OK: This means that no failure occurred during the RPC's execution. By $c(p, d) = OK$, where $p \in P$ is a remote procedure and $d \in O$ is the data object processed by the call, we mean that the RPC call was successful (i.e. the remote procedure performed the job).

FL: This means *accessing failure*. By $c(p, d) = FL$ we mean that the destination server (the server that exports the remote procedure p) could not perform the job because, for example, the arguments between the client and server do not match, the versions are different, or the object managed by the server is not free. This means that the RPC has not executed.

US: This means *unknown state*. By $c(p, d) = US$ we mean that the client cannot tell if the RPC has or has not been executed because, for example, the destination host (the host that the server exporting procedure p resides on) is down, or the server is down, or the links between the client and server are down (that is, the client host and the server host belong to two partitioned sub-networks). In that case the RPC request may be lost, or the return message may be lost. So we do not know if the remote procedure has been executed or not.

Without loss of generality we assume that all RPCs are update-oriented operations. That is, if $c(p, d)$ is successful it transforms the data object d from the existing state to a new state.

2.3. RPC transaction model

We define a (parallel) *RPC transaction* as $T = \{c_1(p_1, d_1), c_2(p_2, d_2), \ldots, c_k(p_k, d_k)\}$, where $c_i(p_i, d_i)$ is an RPC and $p_i \in P$, $d_i \in O$. The semantics of an RPC transaction is that after issuing the transaction, all c_i of T will be executed if no error occurs (commit or OK), or if any one of them fails, all executed RPCs will be rolled back (abort or FL). An RPC $c_i(p_i, d_i) \in T$ returns OK if and only if all replicas of d_i have successfully performed the procedure p_i. Similarly, $c_i(p_i, d_i)$ returns FL if and only if some of its replicas of d_i failed to execute the procedure p_i on d_i.

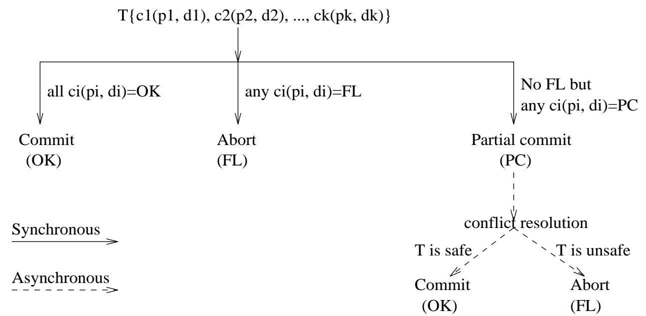

FIGURE 1. Semantics of the RPC transaction.

In addition to the two normal states (commit and abort), we define a *partial commit* (PC) state. The real meaning behind the PC state is that a replica is not accessible (e.g. the replica is down or the network is partitioned), but there are other replicas that can provide the same service. So the transaction has only performed RPCs on all replicas that are alive (can be reached now). For those replicas that are not accessible now, the transaction effect will be resolved when these replicas re-join the service (e.g. the failed replica recovers or the network is reunited). During the conflict-resolution phase, all the transactions that returned PC will be checked. If the work of such a transaction does not conflict with any other transactions (e.g. it does not use any common data objects with other transactions), then it is considered to be safe and will be committed. If a conflict is detected between two transactions, then one of them is chosen as a victim (unsafe) and will be aborted. The other will then be safe and committed. This is done asynchronously.

Most existing RPC systems are synchronous in nature, and hence fail to exploit fully the parallelism of distributed applications. Our RPC transaction model is synchronous when there is no failure and is asynchronous otherwise. The model can achieve high parallelism while retaining the simplicity of the RPC abstraction.

The execution of all c_i in T is in parallel. Some parallel primitives can be built for parallel execution of remote procedures [18]. Sequentially executed transactions can be easily established from the parallel model. However, if operations within a transaction are to be executed sequentially, the serialisability [19] must be considered. In this paper we only consider parallel RPC transactions. Figure 1 indicates the semantics of an RPC transaction.

Parallel transactions exist in many applications. For example, in a scheduling application a scheduler is responsible to schedule k ($k > 0$) tasks concurrently and the schedule is considered successful if and only if all the tasks are successfully scheduled. Schedulers for meetings, timetables, and even flight reservations (multi-hops) can be modelled as scheduling applications.

If we do not need to distinguish individual RPCs within an RPC transaction T, we then write $T = \{c_i(p_i, d_i)\}$ and use $|T|$ to represent the number of parallel RPCs within T.

2.4. Failures

There are four classes of failures in a replicated-server environment executing RPC transactions: transaction abort, replica-is-down, site-is-down and network partition.

An RPC transaction T aborts if any of its RPC returns an FL (e.g. the requested data object is not free, or if all replicas of a service that the RPC accesses are down), or T was in a PC state and then the conflict-resolution process identifies that T is unsafe. The usual treatment for a transaction failure is to retry it after a random period of delay. The random delay is necessary to avoid oscillation when two or more transactions need to use the same data object at the same time.

A replica being down means that the replica is not accessible. In this case, other replicas of the same service should still provide the service to clients (an RPC returns FL if all replicas of a service that it accesses are down). However, when the failed replica recovers and rejoins the service, it may have stale information because some data objects may have been updated when the replica was down. To solve this problem, we set all transactions on the live replicas of the service to the PC state. These partially committed transactions will be made to commit when the failed replica rejoins the service and starts the recovery process. At the same time, the failed replica will update all of its stale information.

A site being down means that the replica(s) that runs (run) on it is (are) down. We assume that each site has only one replica running on it. That reduces the problem into a replica-is-down failure.

A network partition failure means that replicas of a service may belong to two disconnected partitions. In this case, the two sets of replicas should still provide the service to clients. However, when the two partitions are reunited, the two sets of replicas may have performed some conflicting operations. To solve this problem, we also set all transactions performed during the network partition on these two parts of replicas to PC states. These partially committed transactions will be resolved when the two parts are reunited.

We use *system failures* to denote the latter three failures and we assume a single failure for system failures. That is, at any time only one of the replica-is-down failure, or the site-is-down failure, or the network-partition failure occurs. As discussed above, we can actually classify them into two types of failures: the *replica failure* and the *partition*. A replica failure means that a replica is down or the site that the replica is running on is down. A partition means the network is partitioned into two disconnected sets and both parts have some replicas running on them. In both cases, we can divide the replicas into two parts. For simplicity, we only assume crash failures in this paper. Byzantine failures can also be dealt with when the number of replicas of a service is more than three and a majority voting scheme is used.

3. RPC TRANSACTION MANAGEMENT

3.1. The RPC transaction processing model

We define a *primary replica* for a data object d as a replica that is the best (e.g. the nearest site to the *RPC transaction manager* (described below) that accepts the client request, although the measure is left for individual applications) in performing an RPC $c(p, d)$. Any replica can be chosen as the primary replica for a particular RPC. The management of primary replica information is itself a difficult issue. We will concentrate on the transaction management and therefore will ignore this problem here.

Three system components are involved in processing a transaction submitted by a client.

(i) An *RPC transaction manager* (RTM) accepts a transaction $T(c_1(p_1, d_1), c_2(p_2, d_2), \ldots, c_k(p_k, d_k))$ from the client. The RTM sends each RPC $c_i(p_i, d_i) \in T$ to a primary replica of d_i and asks the primary replica to check if the RPC can be performed or not. We denote this operation as $a(c_i(p_i, d_i))$. The RTM then acts as a coordinator for managing the atomicity of T through the help of the primary replicas. Section 3.2 describes the RTM algorithm.

(ii) A primary replica accepts, from the RTM, an RPC $c_i(p_i, d_i)$ and the request to check the executability of the RPC (the $a(c_i(p_i, d_i))$ operation). The primary replica sends the RPC to all replicas of data object d_i and asks all replicas (including itself) to check if they can execute the RPC (e.g. if d_i is free). We use $b(c_i(p_i, d_i))$ to represent this operation. The primary replica acts as a coordinator for managing the RPC $c_i(p_i, d_i)$ to be performed on all replicas (including itself). Section 4.1 describes the coordinating algorithm.

(iii) Each replica of the data object d_i accepts, from the primary replica of d_i, the RPC $c_i(p_i, d_i)$ and the request to check the executability of the RPC (the $b(c_i(p_i, d_i))$ operation). The replica then cooperates with the primary replica by returning the executability check and performing the RPC when requested. Section 4.2 describes the cooperating algorithm.

The request for checking the executability of an RPC is essentially a request for a lock on the data object to be updated by the RPC in most cases. However, in some cases other factors can also affect the executability of an RPC.

The actual effect of an $a(c_i(p_i, d_i))$ call depends on the associated $b(c_i(p_i, d_i))$ calls. That is, if all $b(c_i(p_i, d_i))$ calls return OK, $a(c_i(p_i, d_i))$ returns OK. If any $b(c_i(p_i, d_i))$ call returns FL, $a(c_i(p_i, d_i))$ returns FL. If no $b(c_i(p_i, d_i))$ call returns FL and there are PCs returned by $b(c_i(p_i, d_i))$ calls, or no FL return but there are US or OK returns (the primary replica must return OK or PC in this case), then $a(c_i(p_i, d_i))$ returns PC. An $a(c_i(p_i, d_i))$ may also return US if the primary replica for d_i is not accessible.

A client simply submits a transaction request to the RTM and waits for the transaction result and the resulting state. The resulting state returned at this stage can be one of OK, FL or PC. As soon as the client receives a resulting state of OK or FL, it can continue with its next operation. This means that other operations, such as operations required for consistency or for continuous committing, will be dealt with by the system without the participation of the client.

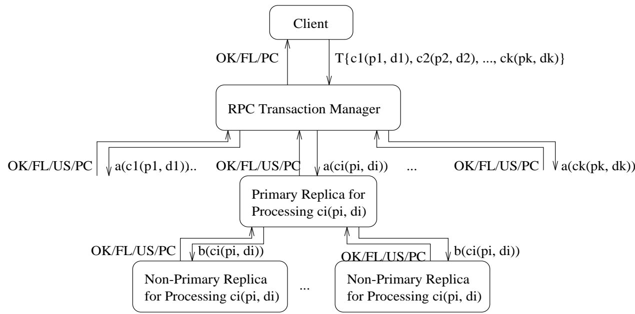

FIGURE 2. RPC transaction processing.

However, if no primary replica returns an FL but some primary replicas return US (i.e. these replicas are not accessible because of a site failure or a network partitioning), then the RTM will retry to find new primary replicas for those primary replicas that returned US. This process repeats until either there are no more available replicas that can act as primary replicas for some particular data objects, or the newly found primary replicas return OK/PC/FL.

If the transaction returns a PC state, it should enable an asynchronous receive procedure (similar to an exception handler) to process the following asynchronous message returned for the transaction. This message will come when the partially committed transaction is resolved. In this case, either a commit or an abort message will be returned. The asynchronous receive procedure should then process this accordingly. Figure 2 depicts the model for RPC transaction processing.

The number of RTMs in a system depends on the requirements of specific applications. Our examples (Section 7) show two different configurations. The first example (Section 7.1) uses one RTM for the whole system, whereas the second example (Section 7.2) uses three RTMs. A number of strategies, such as the twin-server model [20], can be used to improve the reliability of the RTM.

3.2. The transaction manager

The transaction manager has three basic functions to perform when it receives a transaction T.

(i) It submits operations to their primary replicas. The manager is responsible for locating primary replicas for each RPC and then sending these RPCs to their corresponding primary replicas.
(ii) It manages the atomicity of the transaction T. It uses the 2-phase-commit (2PC) protocol to ensure that if all RPCs of T to the primary replicas return OK, T asks all primary replicas to commit and returns OK. If any of these RPCs return FL, it asks all primary replicas to abort and returns FL. In this case, any uncommitted update will eventually be rolled back. If no operation returns FL, but some operations return US, then the manager will find new primary replicas for those that returned US and try again. If all operations return OK or PC (and there is at least one PC return), then T asks all primary replicas to partially commit and returns PC.
(iii) It sets up an asynchronous receive procedure if a transaction returns a PC. The procedure awaits messages from all primary replicas that returned PC. If all of these primary replicas later return OK, then the procedure will upgrade the transaction to a commit state. If any of the primary replicas later returns an FL, then the procedure will downgrade the transaction to an abort.

We define a function `locatePrimaryReplica()` which takes as its input an RPC and returns the primary replica location of the RPC. If the function returns a NULL, this means that there are no more available replicas that can be a primary replica for the RPC concerned (e.g. all replicas for processing a particular object are cut off from the transaction manager). In this case the transaction has to be aborted. This function also guarantees that if $c_i(p_i, d_i), c_j(p_j, d_j) \in T$ and $d_i = d_j$, then `locatePrimaryReplica(`$c_i(p_i, d_i)$`)` = `locatePrimaryReplica(`$c_j(p_j, d_j)$`)`. This means that the two RPCs of the same transaction will use the same primary replica if they access the same data object.

The algorithm `manage_rpc_transaction()` of Listing 1 implements the RTM. The constant MAXRPCS defines the maximum number of RPCs allowed in an RPC transaction.

LISTING 1. Algorithm 1—the RTM algorithm.

```
manage_rpc_transactions()
{
 replica_address_handle primaryAddress[MAXRPCS];
 int InComplete = TRUE;
 while (TRUE) {
  /* receive RPC transactions */
  /* if more transactions come, queue up them until the
   processing of the current transaction is over */
  receive(client, T = {c_i(p_i, d_i)});
  T' <== T;
  while (InComplete) {
   COBEGIN
    primaryAddress[i] = locatePrimaryReplica(d_i);
     ∀c_i(p_i, d_i) ∈ T';
   COEND;
   if ∃primaryAddress[i] = NULL, ∀c_i(p_i, d_i) ∈ T' {
    /* no more available replicas to choose for c_i(p_i, d_i) */
    COBEGIN
     send(primaryAddress[i], "abort"), ∀c_i(p_i, d_i) ∈ T;
    COEND;
    tell the client the transaction returns FL;
    InComplete = FALSE;
    break;
   }
   COBEGIN
    ret_i = a(c_i(p_i, d_i)),
     execute on primaryAddress[i], ∀c_i(p_i, d_i) ∈ T';
   COEND;
```

```
    S_US = {a(c_i(p_i, d_i)) | ret_i ≡ US};
    S_FL = {a(c_i(p_i, d_i)) | ret_i ≡ FL};
    S_PC = {a(c_i(p_i, d_i)) | ret_i ≡ PC};
    S_OK = {a(c_i(p_i, d_i)) | ret_i ≡ OK};
    switch {
     case (S_OK ≠ ∅ ∧ (S_US ≡ S_FL ≡ S_PC ≡ ∅)):
      /* all a(c_i(p_i, d_i)) executed and returned OK */
      COBEGIN
       send(primaryAddress[i], "commit"), ∀c_i(p_i, d_i) ∈ T;
      COEND;
      tell the client that the transaction returns OK;
      InComplete = FALSE;
      break;
     case (S_FL ≠ ∅):
      COBEGIN
       send(primaryAddress[i], "abort"), ∀c_i(p_i, d_i) ∈ T;
      COEND;
      tell the client the transaction returns FL;
      InComplete = FALSE;
      break;
     case (S_FL ≡ ∅ ∧ S_US ≠ ∅):
      /* some of the primary replicas are not accessible */
      T' ⟸ S_US;
      break;
     case (S_FL ≡ S_US ≡ ∅ ∧ S_PC ≠ ∅):
      /* no failure, but there are PC returns */
      set asynchronous receive procedure handler;
      each primary replica is given the address of the handler
       (pigybacked by the following send);
      COBEGIN
       send(primaryAddress[i], "partial commit"),
        ∀c_i(p_i, d_i) ∈ T;
      COEND;
      tell the client that the transaction returns PC;
      InComplete = FALSE;
      break;
    }
   }
  }
}
```

4. REPLICA MANAGEMENT

4.1. The coordinating algorithm for the primary replica

When a primary replica receives an RPC request $c_i(p_i, d_i)$ from a transaction manager, it uses the coordinating algorithm to maintain the consistency of all replicas in terms of the RPC. This section describes the coordinating algorithm.

In the coordinating algorithm the primary replica uses the 2PC protocol to ensure replication consistency. In the first phase, the primary replica asks all replicas (including itself) to check the executability of the RPC (i.e. the $b(c_i(p_i, d_i))$ operation). If all replicas return OK for such an execution, the primary replica returns OK to the transaction manager. If the transaction manager requests commit, then in the second phase the primary replica asks all replicas to commit the RPC execution. If any replica returns FL, then the primary replica returns FL to the transaction manager and asks all replicas to abort the operation in the second phase. If all operations return either OK or PC or US (in this case, the primary replica must return OK or PC and other non-primary replicas may return OK, PC or US), the primary replica returns PC to the transaction manager. The primary replica also records the number of replicas that return OK and PC and the smallest sequence number among these replicas. These two numbers will be used in conflict resolution. If the transaction manager requests partial commit in the second phase, the primary replica asks all replicas to partially commit. If the transaction manager requests an abort, the primary replica then asks all replicas to abort, no matter what was returned by the primary replica in the first phase. The coordinating algorithm is listed in Listing 2. We assume that S_j is the primary replica.

LISTING 2. Algorithm 2—the coordinating algorithm.

```
primary_replica_process
{
 int k, NoOfPCs, smallestPC;
 while (TRUE) {
  /* receive RPCs from RTMs */
  /* if more RPCs come, queue up them */
  receive(RTM, a(c(p, d)));
  k = n; /* n is the number of replicas for d */
  Let S_j, j ≤ k be the primary replica;
  COBEGIN
   ret_i = b(c(p, d^i)),
    where d^i is the copy of d in replica S_i, i = 1, 2, ..., k;
  COEND;
  R_US = {b(c(p, d^i)) | ret_i ≡ US};
  R_FL = {b(c(p, d^i)) | ret_i ≡ FL};
  R_PC = {b(c(p, d^i)) | ret_i ≡ PC};
  R_OK = {b(c(p, d^i)) | ret_i ≡ OK};
  switch {
   case (R_OK ≠ ∅ ∧ (R_US ≡ R_FL ≡ R_PC ≡ ∅)):
    tell the RTM that the first phase returns OK;
    break;
   case (R_FL ≠ ∅):
    tell the RTM that the first phase returns FL;
    NoOfPCs = |R_OK|;
    smallestPC = min(i|b(c(p, d^i)) ∈ R_OK);
    break;
   case (R_FL ≡ ∅∧(ret_j ≡ OK∨ret_j ≡ PC)∧(R_PC ≠ ∅∨R_US ≠
∅)):
    tell the RTM that the first phase returns PC;
    NoOfPCs = |R_PC| + |R_OK|;
    smallestPC = min(i|b(c(p, d^i)) ∈ R_PC ∪ R_OK);
    break;
  }
  receive(RTM, command);
  switch {
   case (command is "commit"):
    COBEGIN
      send(S_i, "commit"), i = 1, 2, ..., k;
    COEND;
```

```
        break;
      case (command is "partial commit"):
        COBEGIN
          send(S_i, "partial commit", NoOfPCs, smallestPC),
            i = 1, 2, ..., k;
        COEND;
        break;
      case (command is "abort"):
        COBEGIN
          send(S_i, "abort"), i = 1, 2, ..., k;
        COEND;
        break;
    }
  }
}
```

In addition to the coordination work, the primary replica has to do all the work of a non-primary replica. The next section describes the role performed by a non-primary replica.

4.2. The cooperating algorithm for all replicas

When a non-primary replica receives a request from a primary replica, it checks whether the request can be proceeded or not and acts accordingly.

(i) If the request can be performed, the non-primary replica locks the required data object and returns OK. Later, if the primary replica asks to commit the operation, the non-primary replica performs the operation and releases the lock. If the primary replica asks to abort the operation, the non-primary replica releases the lock. If the primary replica asks to partially commit the operation, then the non-primary replica partially commits the operation and records this event.

(ii) If the non-primary replica finds that the operation cannot be executed (e.g. the required data object is not free), it then returns an FL.

(iii) If the non-primary replica finds that the data item is already in a partially committed state, then it returns PC to the primary replica. Later, if the primary replica asks to partially commit the operation, the non-primary replica then partially commits the operation and records the event. If the primary replica asks to abort the operation, then the non-primary replica aborts the operation.

The 2-phase-commit protocol used by the RPC transaction manager guarantees that the replicas will stay in a consistent state if a transaction returns an OK or an FL. However, to guarantee that all replicas do the same thing to the transactions with PC pending, a consensus should be reached among all replicas. This is a classic consensus problem in fault-tolerant computing [21]. We use a *need-to-do* (NTD) table, which essentially is a checkpointing log, to record the events of partially committed RPCs. Then, during a recovery we let replicas exchange their NTD table entries in order to reach a consensus for actions on the transactions with PC returns. The NTD table of each replica is kept in *stable storage* [22]. Therefore, information stored in the NTD table will not be affected by system failures. The NTD table structure is listed in Listing 3.

LISTING 3. The need-to-do table.

```
typedef struct ntd {
 char *rpc; /* name of the partially committed RPC */
 char *data; /* data object name used in this RPC */
 int pc; /* No. of replicas partially committed this RPC */
 int sm; /* smallest sequence # among all PC replicas */
 void *ori; /* before image of the data object */
 void *handler; /* asynchronous receive handler address */
 NTD *pre; /* the previous PC RPC for this data object */
 NTD *nxt; /* the next PC RPC for this data object */
} NTD;
```

When the primary replica asks for a partial commit for an RPC, all replicas (including the primary replica) will record this event into their own NTD tables as a new entry. If $t \in$NTD is such an entry, then *t.rpc* contains the name of the RPC, *t.data* contains the data object used in the RPC, *t.pc* stores the number of replicas that have partially committed this RPC, and *t.sm* stores the smallest sequence number among all the replicas that have partially committed the RPC. These two numbers (*t.pc* and *t.sm*) are sent by the primary replica to each replica when it asks for partial commit (see Listing 2). They are used later by the conflict resolution algorithms to determine which partial commit should be upgraded to a commit, and which partial commit should be downgraded to an abort if a conflict occurs.

In order to downgrade a PC to an abort, a *before image* of the data object is kept in the NTD table. *t.ori* is used to record the address of the before image.

The order of each partially committed RPC over a data object is also very important when a recovering replica carries out these RPCs. We use a pair of pointers to record this order. The *t.pre* stores the previous partially committed RPC and the *t.nxt* stores the next partially committed RPC for the same data object. The *t.handler* stores the address of the asynchronous receive handler if the replica is the primary replica.

We associate with each data object d a lock *d.lock* and a partial commit flag *d.pc*. The actual effect of $b(c(p, d))$ on d is to check or change the values of *d.lock* and *d.pc*. That is, if $d.lock \equiv LOCKED\ (-1)$, $b(c(p, d))$ returns FL. If $d.lock \equiv d.pc \equiv FREE$ (0, free and not partially committed), $b(c(p, d))$ returns OK and sets $d.lock = LOCKED$. If $(d.lock \equiv FREE) \wedge (d.pc \equiv n > 0)$ (free but partially committed), $b(c(p, d))$ returns PC and sets $d.lock = LOCKED$ and $d.pc = n + 1$. If the replica is down or the network is partitioned during the operation (i.e. the replica is unreachable), a US is returned by the RPC system.

All RPCs of a transaction are actually performed in the second phase of the algorithm. During this phase, each replica has to release the lock *d.lock* (set to FREE) if the $c(p, d)$ has a normal commit or a partial commit. An entry about this partial commit is also inserted into the NTD table. The conflict resolution algorithms are then responsible for

upgrading it to a normal commit or for downgrading it to an abort. The cooperating algorithm is given in Listing 4.

LISTING 4. Algorithm 3—the cooperating algorithm.

```
all_replica_process()
{
 NTD t;
 int myState;
 while (TRUE) {
  /* receive RPCs from primary replicas */
  /* if more RPCs come, queue up them */
  receive(PR, b(c(p, d)));
  switch {
   case (d.lock ≡ FREE ∧ d.pc ≡ 0):
    d.lock = LOCKED; myState = OK;
    tell the primary replica that the first phase returns OK;
    break;
   case (d.lock ≡ LOCKED):
    myState = FL;
    tell the primary replica that the first phase returns FL;
    break;
   case (d.lock ≡ FREE ∧ d.pc > 0):
    d.lock = LOCKED; d.pc + = 1; myState = PC;
    tell the primary replica that the first phase returns PC;
    break;
   /* US is returned if a replica is unreachable */
  }
  receive(PR, command, NoOfPCs, smallestPC);
  switch {
   case (command is "commit"):
    do (ret = c(p, d)) until ret ≡ OK;
    d.lock = FREE;
    break;
   case (command is "partial commit"):
    d.lock = FREE;
    if myState ≡ OK then d.pc = 1; endif;
    t.rpc = c(p, d); t.pc = NoOfPCs;
    t.sm = smallestPC; t.data = d;
    t.ori = beforeImage(d); t.pre = previous(d); t.nxt = ∅;
    if the replica is the primary replica of this RPC then
     t.handler = the asynchronous receive procedure handler
                 for this RPC;
    endif;
    do (ret = c(p, d)) until ret ≡ OK;
    break;
   case (command is "abort"):
    if myState ≡ OK then d.lock = 0;
     else if myState ≡ PC then d.lock = 0; d.pc − = 1;
    endif;
    break;
  }
 }
}
```

The algorithm will not cause deadlocks since there is no waiting in the algorithm: a failure (FL) is returned as long as a data object has been locked.

5. CONFLICT RESOLUTION

An RPC returns US only when the server is down or the server is unreachable (e.g. the network is partitioned). If the server is down, it will eventually be repaired and return to service. In that case, the server missed all updates to its data objects while it is down. Fortunately, we have set these updates in 'partial commit' state and recorded them in NTD tables. Therefore, the conflict resolution algorithms can use this information and make the data objects managed by this failed replica in line with all other replicas. If the network is partitioned into two disconnecting parts, the two parts will eventually be reunited again. In this case, replicas in both partitions may have some partially committed updates. The conflict resolution algorithms are also responsible for making replicas in these two parts consistent.

Since the recovery process is highly critical, we assume that during the recovery process (1) no system failures will happen, and (2) no RPC transactions are allowed. These assumptions can be loosened if more communication and stable storage are used in conflict resolution. For instance, if system failures do happen during the recovery process, then we require that the recovery process logs all its work in a stable storage where the next recovery process can continue based on the stored information. If RPC transactions are allowed to be submitted to a replica during the recovery process, then these transactions will be stored in a stable storage on the replica. The clients will be notified of possible delays for the transactions. When the recovery process completes, these stored transactions will be processed immediately.

When recovering from a replica failure, we assume that there is a process that deletes all entries of the NTD table of the recovering replica, except those entries with $t.handler \neq \emptyset$. The reasons for doing this are: (i) those entries with $t.handler = \emptyset$ are no longer useful—they will be dealt with by other alive replicas; and (ii) those entries with $t.handler \neq \emptyset$ have to be dealt with by the recovering replica because it was the primary replica for the particular RPC. This NTD table is sent to all alive replicas by the recovering process.

Two algorithms are needed during the recovery of a system failure. When recovering from a replica failure, the recovering replica has to send a 'reuniting' message (it includes the sequence number of the recovering replica) to other alive replicas. This enables the alive replicas to send outstanding NTD table entries to the recovering replicas by using the `send_out_ntd()` algorithm.

When recovering from a network partition, replicas of each part of the partition have to send a 'reuniting' message (it includes the sequence numbers of all replicas of the partition) to replicas of the other part. This enables the exchange of NTD tables among replicas in both parts by using the `send_out_ntd()` algorithm.

Once all replicas have received the NTD table from the other part, the `conflict_resolution()` algorithm is used by all replicas to resolve possible conflicts and to finalize the outstanding partially committed transactions.

The `send_out_ntd()` algorithm is very simple. It detects the reuniting messages. If such a message comes, it then sends all entries of the NTD table to all replicas included in the reuniting message. Listing 5 describes the algorithm.

LISTING 5. Algorithm 4—the algorithm for sending out an NTD table.

```
send_out_ntd()
{
 while (TRUE) {
  /* detect the reuniting messages */
  /* the process suspends until such a message comes */
  reUnitSet = {sequence numbers of reuniting replicas};
  while the NTD table is locked
   wait;
  COBEGIN
   send(S_i, NTD), i ∈ reUnitSet;
  COEND;
 }
}
```

The `conflict_resolution()` algorithm is used by all replicas to resolve potential conflicts for a data object. The algorithm receives NTD entries from replicas of the other part. Only the first arriving NTD table is accepted by the algorithm and others are ignored. The algorithm then checks the received NTD entries against its own NTD entries to resolve potential conflicts.

We define a *leader* of an NTD table as the first partial commit entry for a data object d. That is, if $t \in T$ is a leader in an NTD table T, then $t.previous = \emptyset$. The algorithm uses three functions for processing NTD table entries led by a leader:

(i) The `abort_all(`t_{me}, q`)` function aborts all RPCs led by the leader q of the NTD table t_{me}.
(ii) The `commit_all_own(`t_{me}, q`)` function commits all RPCs led by the leader q of the NTD table t_{me}. The NTD table t_{me} is the replica's own NTD table.
(iii) The `commit_all_received(`t_{ot}, t`)` function commits all RPCs led by the leader t of the NTD table t_{ot}. The NTD table t_{ot} is the received NTD table.

We first describe these three functions.

Listing 6 is the `abort_all(`t_{me}, q`)` function. It finds the tail of the entries led by q and rolls back all RPCs from the tail to the leader. If the replica is also the primary replica of the RPC, an FL message is also sent to the asynchronous receive procedure.

Listing 7 is the `commit_all_own(`t_{me}, q`)` function. It finds the tail of the entries led by q and commits all RPCs from the tail to the leader. If the replica is also the primary replica of the RPC, an OK message is also sent to the asynchronous receive procedure.

Listing 8 is the `commit_all_received(`t_{ot}, t`)` function. It finds the tail of the entries led by t and commits all RPCs from the tail to the leader.

Listing 9 describes the `conflict_resolution()` algorithm.

LISTING 6. The `abort_all()` function.

```
abort_all(t_me, q) /* abort all RPCs led by q */
{
 NTD h, t;
 h = q; t = the tail of the NTD entries led by h;
 while (TRUE) {
  d = t.ori; d.pc − = 1;
   /* other activities for rolling back an RPC omitted */
  if t.handler ≠ ∅ then
   send an FL message to t.handler;
  endif;
  t = t.pre; /* the previous entry */
  if (t.pre ≡ ∅) then break; endif;
 }
 delete all entries led by q;
}
```

LISTING 7. The `commit_all_own()` function.

```
commit_all_own(t_me, q)
/* commit all RPCs led by q, own NTD table */
{
 NTD h, t;
 h = q; t = the tail of the NTD entries led by h;
 while (TRUE) {
  d.pc − = 1; t = t.pre;
  if t.handler ≠ ∅ then
   send an OK message to t.handler;
  endif;
  if (t.pre ≡ ∅) then break; endif;
 }
 delete all entries led by q;
}
```

LISTING 8. The `commit_all_received()` function.

```
commit_all_received(t_ot, t)
/* commit all RPCs led by t, received NTD table */
{
 NTD h, a;
 h = q; a = the tail of the NTD entries led by h;
 while (TRUE) {
  do a.rpc until an OK returns;
  a = a.pre;
  if (a.pre ≡ ∅) then break; endif;
 }
}
```

LISTING 9. Algorithm 5—the conflict resolution algorithm.

```
conflict_resolution()
{
 NTD t, q;
 while (TRUE) {
  /* receive NTD entries from alive/re-uniting replicas */
  /* the process suspends until such a message comes */
  /* if more such messages come, only accepts the first one
   others ignored. Received NTD entries are stored in t_ot */
  receive(t_ot);
  Let t_me be the own NTD table of this replica;
  for each leader t of t_ot do
   if (∃ a leader q ∈ t_me such that t.data ≡ q.data) then
    /* conflict updates */
```

```
      switch {
        case (t.pc > q.pc) ∨ (t.pc ≡ q.pc ∧ (t.smallestPC >
q.smallestPC)):
          /* abort all RPCs led by q in this replica */
          abort_all(t_me, q);
          /* commit all RPCs led by t for other replicas */
          commit_all_received(t_ot, t);
          break:
        case (t.pc < q.pc) ∨ (t.pc ≡ q.pc ∧ (t.smallestPC <
q.smallestPC)):
          /* commit all RPCs led by q in this replica */
          commit_all_own(t_me, q);
          break:
      }
      else commit_all_received(t_ot, t); /* no conflict */
    endif;
  endfor;
  for each leader q of t_me do
    commit_all_own(t_me, q); /* no more conflicts */
  endfor;
 }
 delete all entries of t_me (own NTD table);
}
```

6. CORRECTNESS

In this section we outline the informal analysis of the correctness. We first assume the life-cycle of the system entities (sites, links and replicas) is

$$\text{work} \rightarrow \text{crash} \rightarrow \text{repair and restart} \rightarrow \text{work}.$$

Without loss of generality, we assume that the maximum down time (including crash, repair and restart time) is finite, and it is denoted as T_d.

ASSERTION 1. *If a transaction returns* OK*, all its RPCs have been executed successfully.*

Proof. The only way that a transaction returns OK is that in Algorithm 1 we have $(S_{OK} \neq \emptyset \wedge (S_{US} \equiv S_{FL} \equiv S_{PC} \equiv \emptyset))$. This means all RPCs to the primary replicas have returned OK. A primary replica returns OK if and only if in Algorithm 2 we again have $(R_{OK} \neq \emptyset \wedge (R_{US} \equiv R_{FL} \equiv R_{PC} \equiv \emptyset))$. This means all RPCs to all replicas return OK. A replica returns OK if and only if in Algorithm 3 the data object involved is free and is not partially committed. If the transaction returns OK, in the second phase of Algorithm 1 we order all primary replicas to commit the transaction. In this case, all primary replicas will order their replicas to commit in the second phase of Algorithm 2, and therefore all replicas will successfully perform the real RPC in their second phase of Algorithm 3.

ASSERTION 2. *If a transaction returns* FL*, no RPCs of the transaction have been executed.*

Proof. There are two ways that a transaction could return an FL. The first way that a transaction returns FL is that in Algorithm 1 we have $(R_{FL} \neq \emptyset)$. This means some of the primary replicas returned FL. A primary replica returns FL if and only if in Algorithm 2 some of its replicas returned FL. Furthermore, a replica return FL if and only if in Algorithm 3 it finds that the data object is not free. If the transaction returns FL, in the second phase of Algorithm 1 we order all primary replicas to abort the transaction. In this case, all primary replicas will order their replicas to abort in the second phase of Algorithm 2, and therefore in Algorithm 3 those replicas that returned OK will release their locks to the data object. The replicas that returned PC will set the associated integer of the data object to the free state. The replicas that returned FL will do nothing. In any case, none of the RPCs is executed.

The second way that a transaction returns an FL is that in Algorithm 1 we cannot find any more replicas to be the primary replica for a particular RPC. In this case all primary replicas are ordered to abort the transaction. The rest of the work done by Algorithms 2 and 3 is the same as described above. Therefore, none of the RPCs is executed in any way.

ASSERTION 3. *The NTD table will not grow indefinitely and any entry of the table will be deleted eventually.*

Proof. An NTD table increases if and only if there is a system failure, such as a replica or a site is down, or the network is partitioned into two parts. From our life-cycle assumption for system entities, we know that the maximum time of a system failure is T_d. After that, the replica or the site will rejoin the service and the partitioned network will be reunited. During the recovery, all replicas have to execute Algorithm 5 and the algorithm will delete all NTD table entries at the end of conflict resolution (it is easy to show that Algorithm 5 terminates).

ASSERTION 4. *After the conflict resolution, all outstanding PCs will be either committed or aborted.*

Proof. The conflict resolution is carried out by Algorithm 5 after the exchange of NTD tables between replicas of the two parts (failed or alive replicas, or two parts of the partition). The algorithm checks the replica's NTD table against the received NTD table to see if there are any update conflicts. If no conflict is detected, all PC updates in both NTD tables are committed. If there is a conflict for an update, the algorithm determines the part that has more replicas, or has the smallest sequence number if both parts have the same number of replicas. Then the NTD update from this part is committed, while the NTD update from the other part is aborted. This process continues until all entries of both tables (i.e. all outstanding PCs) are exhausted.

ASSERTION 5. *If a transaction returns PC, the transaction will be notified of an* OK *or an* FL *return in a finite time.*

Proof. The only way that a transaction returns PC is that in Algorithm 1 we have $(S_{FL} \equiv S_{US} \equiv \emptyset \wedge S_{PC} \neq \emptyset)$. This means there is no FL nor US return but some primary replicas return PC. A primary replica returns PC if and only if in Algorithm 2 we have $(R_{FL} \equiv \emptyset \wedge (ret_j \equiv OK \vee ret_j \equiv PC) \wedge (R_{PC} \neq \emptyset \vee R_{US} \neq \emptyset))$. This means there is no FL return, and the primary replica returns OK or PC (this

guarantees that the primary replica is accessible), and there may be PC or US returns from the replicas. A replica returns PC if and only if in Algorithm 3 the replica finds that the data object is already in a PC state. A replica returns US if it is not accessible. If a transaction returns PC, in the second phase of Algorithm 1 we set up an asynchronous receive procedure and notify each primary replica of the address of this procedure. Then we order all primary replicas to partially commit the RPC. The real effect of a partial commit is in Algorithm 3, where an entry is stored in the NTD table of each involved replica. As we have shown in Assertions 3 and 4, after a finite time all outstanding PCs in all NTD tables will be either committed or aborted. At the same time, these commit/abort messages will be sent by relevant primary replicas to the relevant asynchronous receive procedures (in functions `commit_all_own()` and `abort_all()`). These asynchronous receive procedures will then notify the relevant transactions of the final result.

ASSERTION 6. *After the recovery of a system failure, all data objects managed by the replicas will be in consistent states.*

Proof. There are two types of system failure: replica failure and network partition. During the recovery of a replica failure, the recovering replica will receive the NTD entries of an alive replica and all alive replicas will receive an NTD table from the recovering replica. During the recovery of a network partition, replicas of both parts of the partition will exchange their NTD tables. During the conflict resolution, all replicas will perform similar work because they all have the same information (NTD tables of both parts). Entries in these two tables are compared to check conflicts. If no conflicts are found, RPCs in both tables will be committed by all replicas. If any conflict of an RPC occurs, only the RPC from one table will be committed; the other will be aborted. In any case, the result will be consistent.

ASSERTION 7. *The proposed system will not impose too much overhead on the system compared with a non-replicated system.*

Proof. In our system, the transaction returns to the client as soon as it has made the second phase order. The condition for a transaction to make the second phase order is that all primary replicas have returned their feasibility checks (i.e. if the RPCs can be performed). Furthermore, the condition for a primary replica to return the feasibility check is that all replicas of the primary replica have returned the feasibility checks. This means the transaction can return the resulting state to the client without waiting for the RPCs of the transaction to complete. The real RPCs of the transaction are done after these feasibility checks have been returned.

It is anticipated that a feasibility check is faster than a real RPC because the former only checks if the data object is free. In this case, our system has a comparable response time to a non-replicated system, where all real RPCs to all non-replicated servers have to be returned before the transaction can return to the client.

The system response time can even be improved by using

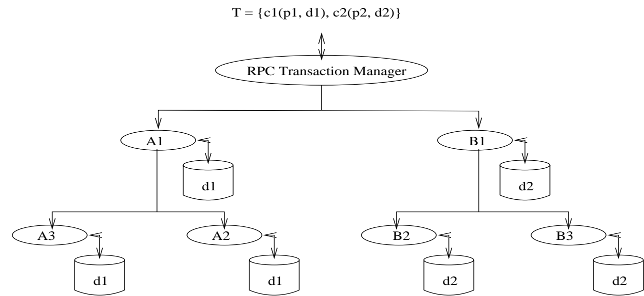

FIGURE 3. An example of RPC transaction processing.

the feasibility check results of all primary replicas instead of waiting for the feasibility check result of all replicas of each primary replica. A tentative decision can be made according to these results. An asynchronous procedure is then set up by the transaction manager to process the results returned from all other replicas, and the final transaction result will be decided according to these returns.

7. EXAMPLES

In this section we present two examples of the system. The purpose of the first example is to illustrate the algorithms described earlier, and the aim of the second example is to (1) show that our system can be used in practical applications, and (2) experiment with the system in a real implementation to obtain some indications on performance issues.

7.1. An illustrating example

In this example we assume that there are two data objects d_1 and d_2. Data object d_1 is replicated in three replicas A_1, A_2 and A_3, and data object d_2 is replicated in B_1, B_2 and B_3. An RPC transaction $T = \{c_1(p_1, d_1), c_2(p_2, d_2)\}$ is to be executed through the help of our RPC transaction manager. Without loss of generality, we also assume that the transaction manager has chosen A_1 and B_1 as the primary replicas for the execution of the transaction. Figure 3 shows the RPC transaction processing for this example.

Let us discuss the following three typical scenarios. Other situations can be analysed in the same manner.

Case 1: No failures. The RPC transaction manager uses Algorithm 1 to ask primary replica A_1 to check the feasibility of $c_1(p_1, d_1)$ and B_1 to check the feasibility of $c_2(p_2, d_2)$ (the first phase). The two primary replicas then use Algorithm 2 to check the feasibility of the work. Each replica uses Algorithm 3 to do the job. Since there are no failures, all replicas will return OK and the data objects are locked by individual replicas. In the second phase, the two primary replicas are asked to commit the transaction. In turn, these primary replicas ask their replicas to commit the transaction. In that case, all replicas will commit (the second phase of Algorithm 3) the transaction and release the locks. The transaction is then successfully executed. Figure 4 shows the actions of the RPC transaction manager and the replicas when there are no failures.

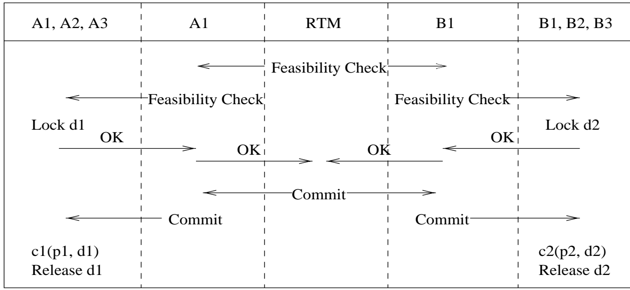

FIGURE 4. No failures.

t:

t.rpc=c1(p1,d1)	t.pc=2	t.sm=1	t.data=d1	t.ori=beforeImage(d1)	t.handler=asyn. procedure addr	t.pre=ϕ	t.nxt=ϕ

(a) The NTD entry for A1 when process T={c1(p1, d1), c2(p2, d2)}

t:	t.rpc=c1(p1,d1)	t.pc=2	t.sm=1	t.data=d1	t.ori=beforeImage(d1)	t.handler=asyn. procedure addr	t.pre=ϕ	t.nxt=t1
t1:	t1.rpc=c3(p3,d1)	t1.pc=2	t1.sm=1	t1.data=d1	t1.ori=beforeImage(d1)	t1.handler=asyn. procedure addr	t1.pre=t	t1.nxt=ϕ

(b) The NTD entry for A1 when process T'={c3(p3, d1), c4(p4, d2)}

FIGURE 5. The NTD entries during the processing of T and T'.

Case 2: A_2 is down and then recovers. Let us first look at the situation that A_2 is down. In this case the feasibility check by B_1 will return an OK, but the feasibility check by A_1 will return a PC. All data objects (except A_2's data object) are locked. In the second phase, the RPC transaction manager will decide to partially commit the transaction and each primary replica will be given the address of an asynchronous handler (but only A_1 stores the address). All replicas (except A_2) will then execute the RPC and release the locks. Also, replicas A_1 will set the partial commit number for d_1 ($d_1.pc$) to 1 and will record the information depicted in Figure 5a into their NTD tables.

A_3, B_1, B_2 and B_3 will also have a similar entry added into their NTD tables.

When A_2 recovers and is reunited into the system, A_2's recovery process will clean up its NTD table and send a reuniting message and the empty NTD table to all other replicas of d_1 (i.e. A_1 and A_3). Upon receiving the reuniting message, A_1 and A_3 will use Algorithm 4 to send their NTD table entries to A_2 (the first arrived NTD table will be accepted, others discarded). Then, all three replicas will use Algorithm 5 to resolve conflicts based on the same view of the NTD tables. In that case, A_1 and A_3 will commit the RPC that returned a PC previously by decreasing the *d.pc* number by 1, and delete the entry from their NTD table. Since A_1 was the primary replica for this RPC, it also has to use its *t.handler* field to send an OK message to the RTM. The RTM then can tell the client program that the transaction is upgraded to a commit state. Figure 6 shows the actions of this scenario.

If a second transaction $T' = \{c_3(p_3, d_1), c_4(p_4, d_2)\}$ is submitted to the RTM before A_2 is recovered, then all

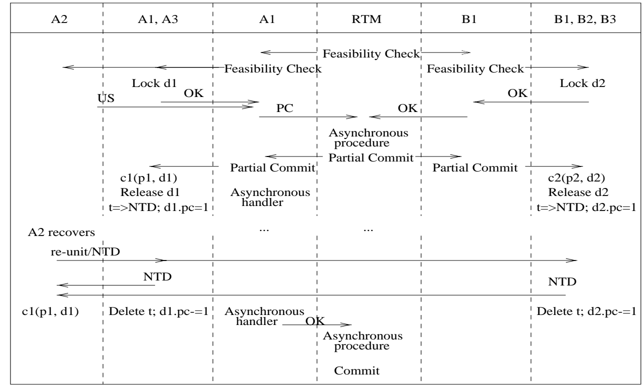

FIGURE 6. A_2 is down and then recovers.

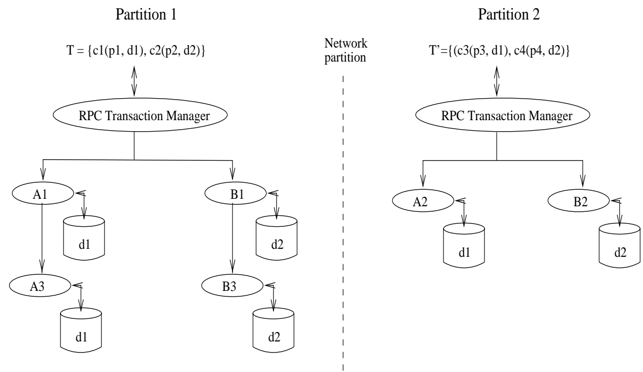

FIGURE 7. The network is partitioned into two parts.

feasibility checks will return PC states and the RTM will decide to do a partial commit. In that case, an entry t_1 will be added to all replicas' NTD tables, with the new entry linked to the previous entry, as shown in Figure 5b. The $d_i.pc, i = 1, 2$, fields are increased by 1. Both transactions will be upgraded into the commit state when A_2 recovers.

Case 3: The network is partitioned into two parts. Suppose that we have the situation depicted in Figure 7, where the network is partitioned into two parts and a transaction is submitted to each part of the partition.

The feasibility checks of both partitions will return PC states and therefore both transactions will be partially committed by their RTMs. When the two partitions are reunited, both partitions will exchange information of their NTD tables and use Algorithm 5 to resolve conflicts. Since partition 1 has more partially committed replicas, T will be committed in both partitions and T' will be aborted in partition 2.

7.2. An application example

The system described in this paper has been used in our implementation to improve the reliability of a loosely integrated heterogeneous database system [23]. This section describes the architecture of the system and some results of our experiments.

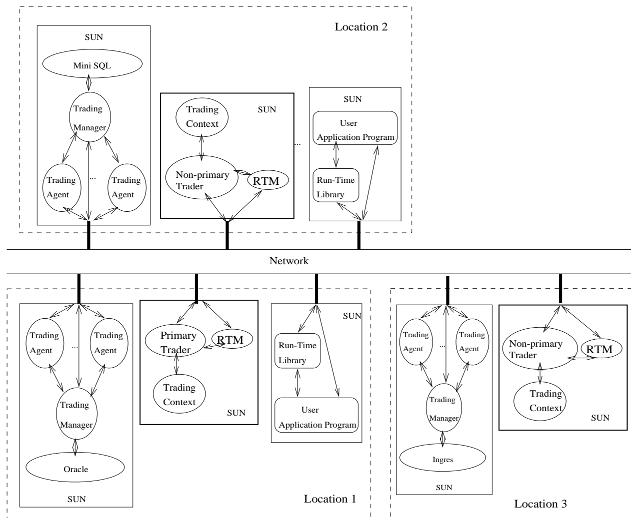

FIGURE 8. Architecture of the loosely integrated system.

7.2.1. The architecture

Figure 8 depicts the architecture of our loosely integrated heterogeneous database system.

The system consists of the following components:

(i) Database servers. Each individual database server maintains database operations directed to it. Currently three database servers (Oracle, MiniSQL and Ingres) are running on three separate Sun workstations located about 100 km apart.

(ii) The trader. A *trader* is a third-party object that links clients and servers in a distributed system [24]. We use a trader to manage the shared information among participating databases of the loosely integrated database system. If a database system is willing to share part of its information with other database systems or outside applications, it *export*s (an update operation) that information as a *service offer* to the trader. These service offers are managed by the trader as *trading context*. Application programs (clients) wishing to make use of the shared information have to *import* (a read-only operation) such service offers from the trader and then access the database(s) concerned. Trading operations, including the export and import operations, are implemented as RPCs.

(iii) The trading manager. A *trading manager* is built on every participating database system for performing all common tasks of trading preparation and management. It is responsible for such tasks as checking the validity of trading requests, forming local offers, executing the service, and returning request results.

(iv) Trading agents. *Trading agents* are appointed by database servers to manage some special service offers. For example, if some schema translation is needed, or if a service offer involves accessing multiple database systems, then a trading agent can be appointed to manage the offer. A trading agent exports the service offer it manages to the trader. The client imports the service offer from the trader and is given the agent's address (and the description of the service, of course). It then calls the agent and obtains the service.

Individual databases in the heterogeneous database system are allowed to maintain their autonomy, yet through the use of traders, they provide a substantial degree of information sharing. The trading manager executes on the same host as the database server. The trader, trading agents and user application programs can be executed on any Sun workstations.

The trader is implemented as a server that manages the trading context through RPCs. The trader is a key component of the system—the system cannot function once the trader fails and therefore it is replicated. That is, each location has a trader (and the replicated trading context) running on it. The trader on Location 1 (with an Oracle database running in that location) is assigned as the primary replica, whereas the other two traders are assigned as non-primary replicas. An RPC transaction manager (RTM) also runs on each location for managing RPC transactions initialized from the location. These RPC transaction managers have well-known addresses.

Our previous fault-tolerant RPC system [25] is used to implement the reliable communications required by various components of the RPC transaction management system. The RPC transaction features are now manually coded (using threads) and an extension of the interface definition file is under way to include the expression of RPC transactions.

In order to reduce the communication time, a communication channel (TCP stream) is established between the primary replica and non-primary replicas (through the local RTMs) during the lifetime of an RPC transaction.

7.2.2. Experiments

We have carried out a number of experiments using the system. The first class of experiments is on the fault tolerance of the system, especially the fault tolerance of the replicated traders. These experiments include the situations of (1) no failures, (2) a non-primary replica of the trader server fails and then recovers, (3) the primary replica of the trader fails and then recovers, (4) the combination of failures of a trader and an RPC transaction manager, and (5) the network is partitioned into two subnets (through software simulation) and then recovers. These experiments have shown that the system, when using algorithms described in this paper, can tolerate these failures.

The second class of experiments is on the performance of the replicated trader servers when there are no failures. As a comparison, we measured the performance of a null RPC, a single trader, two traders and three traders. Figure 9 depicts the architecture of these experiments.

All clients and servers were executed on Sun SPARC 5 workstations running the Solaris operating system. For Figures 9a and 9b, both the clients and server/trader were executed on separate workstations within the same subnet. For Figures 9c and 9d, the client and the primary

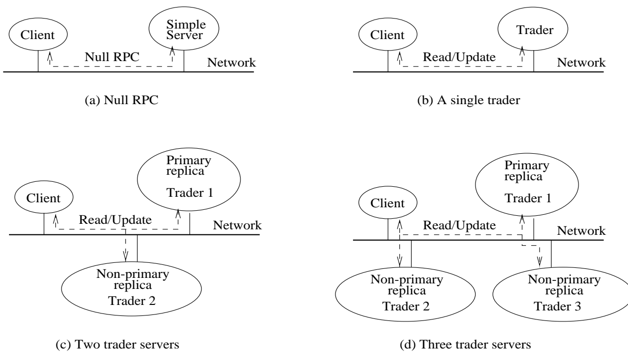

FIGURE 9. Performance tests.

TABLE 1. Performance comparison (no failure).

Service type	Mean (ms)	SSD (ms)
Null RPC	3.07	0.51
Single trader (Read-only)	5.24	0.84
Single trader (Update)	5.31	0.87
Two traders (Read-only)	5.23	0.73
Two traders (Update)	6.14	1.06
Three traders (Read-only)	5.25	0.88
Three traders (Update)	6.27	1.04

trader were executed on separate workstations of the same subnet, whereas each non-primary trader was executed on a workstation of another subnet. Workstations of each subnet are linked through a 10-Mbit Ethernet network. The three subnets (about 100 km apart) are linked through a 34-Mbit ATM Backbone (the implication is that normally the ATM Backbone will not be the bottleneck of the communication during our tests).

All tests were carried out during the night when the traffic was low. Each service type was tested 1000 times and then the mean and the sample standard deviation (SSD) were calculated. Table 1 lists the test results.

We can draw the following observations and explanations from Table 1:

(i) The overhead of a single trader is not light compared to a null RPC, even if it does not involve any fault-tolerant issues. The table shows that for a simple trader the overhead is more than 2 ms. Most of this overhead was due to the implementation of the trading algorithms and the access time for the trading context (stored as a disk file). It is also interesting to see that the read-only operations and update operations do not have much difference in a single-trader case.

(ii) There is virtually no difference in read-only operations when the number of traders increases from one to three. This is simply because we implemented our traders in such a way that all read-only operations are dealt with by the local traders, no matter how many traders are used in the system.

TABLE 2. Performance comparison (with failure).

	Read-only (ms)		Update (ms)	
Service type	Mean	SSD	Mean	SSD
No failure	5.23	0.98	6.14	1.24
Non-primary fails	5.24	1.10	7.90	1.27

(iii) The mean response time for an update operation has increased by 0.83 ms when the number of traders increases from one to two. This overhead is mainly due to the implementation of the algorithms for keeping the two replicas consistent. The increase is not significant since the communication between the two traders (a TCP channel, implemented through our own RPC system [25]) has been established before any update operation can be accepted by the replicas.

(iv) The mean response time for an update operation only increases by 0.13 ms as the number of traders increases from two to three. The main reason for such a small increase is due to the way we handle operations involving multiple replicas (as shown in Algorithms 1, 2 and 3), i.e. all these operations are sent to involved replicas concurrently (using threads). Since the traffic was low, the synchronization of these operations did not take too long.

Obviously, a different configuration will lead to a different result. For example, if each subnet is changed to a 100-Mbit Ethernet network, then the cost of communications among each subnet will become a significant part of the total cost since the speed of the ATM backbone is only 34 Mbits. However, it is also worth noting that a configuration with two replicas will perform similarly to a configuration with N replicas ($N > 2$ and being small) if the networks are equally fast. The main reason is that all replicas process and reply to requests from the primary replica in parallel.

The third class of experiments is on the impact of failures on the performance of the replicated traders. We carried out our experiments on the system of two traders, as depicted in Figure 9c, and measured the performance of export (update) and import (read-only) operations when the non-primary replica is faulty. In the case of the failure of the non-primary replica, RPC transactions can be treated as partial commit, and the primary replica has to record these partially committed transactions in its NTD table. Table 2 shows the result of the experiment.

It can be seen from Table 2 that the performance of read-only operations virtually does not change when the non-primary replica fails, since read-only operations are not affected in our system as long as there is one alive replica. However, the mean time of update operations has increased by 1.76 ms. This increase is mainly due to the disk I/O used to access the NTD table (which is stored as a disk file).

The last class of experiments is on the issues of recovery. We used the same system architecture (Figure 9c) as we did

TABLE 3. Performance comparison (recovery).

	1 update (ms)		10 updates (ms)	
Recovery type	Mean	SSD	Mean	SSD
Non-primary failure	8.83	1.41	24.67	2.81
Partition (no conflict)	8.98	1.39	24.79	2.92
Partition (conflict)	12.35	1.56	45.91	5.30

in the third class of experiments. Here we consider two types of recoveries: (1) a failed non-primary replica rejoins the system, and (2) a recovery from a network partition.

In the first case, during the time the non-primary replica is faulty, the primary replica may have carried out some update operations on the trading context. These update operations have been recorded in the NTD table of the primary replica and should be executed by the non-primary replica when it recovers. When the non-primary replica recovers, it cleans up its NTD table and sends a reuniting message to the primary replica. The primary replica will then send the relevant NTD table entries to the recovering replica and commit the outstanding partially committed transactions. In the meantime, the non-primary replica will carry out the missed operations. Here, we measured the time from the point that the recovering replica sends out the reuniting message to the time that it has completed all missed operations.

In the second case, when the network is partitioned, the two replicas are isolated from each other (we simulated this failure by deliberately coding wrong addresses into the two replicas). According to our algorithms, both replicas will become the primary replica of their own subnet and will be able to accept transactions. We have considered two situations for the update operations carried out during the network partition: (1) all of these update operations are not conflict, and (2) all of them are conflict.

When the two replicas recover from the network partition failure, they have to exchange their NTD tables and use Algorithm 5 to resolve conflicts. For situation (1), since there is no conflict, both replicas will execute the missed operations and commit their own outstanding partially committed operations. For situation (2), since the original primary replica (Trader 1) has a smaller sequence number, the original non-primary replica (Trader 2) has to abort all its partially committed operations and execute the operations carried out by Trader 1 according to Algorithm 5. Here, we measured the time from the point that Trader 2 issues the reuniting message to the time that it has completed all missed operations.

In all cases of the experiments related to recovery issues, the numbers of update operations during the failure time were set to 1 and 10. Table 3 lists the test results.

We make the following observations from Table 3:

(i) The difference between the time used to recover from a replica failure and the time for a network partition failure is small when there is no conflict. This is mainly because the same algorithms were used and the recovering replica Trader 2 (the one where we measured the time) had to perform the same operations in both cases.

(ii) The mean recovery time increases significantly as the number of update operations increases. This is mainly due to the way that we process update operations: the update operations were processed sequentially on the trading context file. We have proposed an algorithm to parallelize recovery operations [26], and it is planned to embed the algorithm into the RPC transaction system.

(iii) The mean recovery time increases dramatically as the number of conflict operations increases. This is mainly due to the way we process conflict operations. For each conflict operation, a replica has to perform two disk I/O operations (one on the NTD table and one on the trading context file) sequentially. Once again, our proposed algorithm for parallelizing recovery operations [26] can play an important role here in improving the performance.

8. REMARKS

A system for building reliable computing over an RPC system is described in this paper. The system combines the replication and transaction techniques together and embeds these techniques into the RPC system. The paper describes the models for replicas, RPCs, transactions, and the algorithms for managing transactions, replicas, and resolving conflicts during system recovery. Finally, an informal correctness analysis is carried out and an illustration example and an application example are described.

Although many ideas used in this paper are well known, and some of them have been implemented in commercial products, the paper does provide the following novel contributions. The first major contribution of the paper is the combination of replication, transaction management and RPC techniques to form a system that supports the development of reliable services in the RPC level. The main advantages of supporting fault tolerance in the (lower) RPC level instead of the (higher) application/service level are: (1) efficiency, since failures can be dealt with in a lower protocol level, (2) failure containment, since failures can be dealt with quickly and can reduce the danger of failure propagation, and (3) lower development cost, since the lower-level support means that many higher-level applications/services will be less likely to repeatedly develop their own fault-tolerant protocols.

The second major contribution of the paper is the introduction of a partial commit (PC) concept to facilitate the processing of transactions in case of failures. The motivation of using the PC state for a transaction is to let the transaction proceed even if a replica is down or the network is partitioned. Partially committed transactions will be upgraded to commit states or downgraded to abort states

when the system recovers from the failure that affected the transactions.

A similar work of combining replication, transaction management and RPC techniques to support fault-tolerant computing is the Encina toolkit [27, 28]. Encina extends the basic DCE services to include facilities such as the transactional RPC and the Encina Monitor (a transaction processing monitor). The major difference between our work and the Encina toolkit is that Encina only has two states for a transaction: commit or abort. Our proposal uses a partial commit concept to let transactions proceed even during a replica failure or a network partition.

The fulfilment transaction approach [7] also allows transactions to proceed during a network partition, by generating some new transactions (fulfilment transactions) that will be processed when the network partition failure is recovered. The main difference between our work and the fulfilment transaction approach is that our model does not require the application-specific knowledge to build those fulfilment transactions. That gives us a freedom of implementing our model in a lower level of the system hierarchy which can be shared by many different applications. With enough business policies embedded into the generation of fulfilment transactions, the chance of manual intervention in these fulfilment transactions can be greatly reduced. However, our method avoids the use of any manual intervention.

The algorithms described in the paper have been used in improving the reliability of traders used in an experimental loosely integrated database system. A series of experiments has been conducted on the implementation to test the reliability of the system, the overhead for maintaining replicas, the impact of failures on the system performance, and the overhead during recovery. From these experiments we have shown that the proposed algorithms can provide reliability without incurring too much negative impact on the system performance.

ACKNOWLEDGEMENTS

The authors would like to thank the referees for the helpful comments.

REFERENCES

[1] Wellings, A. J. and Burns, A. (1996) Programming replicated systems in Ada 95. *Comp. J.*, **39**, 361–373.

[2] Cmelik, R. F., Gehani, N. H. and Roome, W. D. (1988) Fault tolerant C: A tool for writing fault tolerant distributed programs. In *Digest of Papers: The 18th Ann. Int. Symp. on Fault-Tolerant Computing*, Tokyo, Japan, pp. 56–61.

[3] Schlichting, R. D. and Thomas, V. T. (1995) Programming language support for writing fault-tolerant distributed software. *IEEE Trans. Comput.*, **44**, 203–212.

[4] van Renesse, R. and Birman, K. (1994) Fault-tolerant programming using process groups. In Brazier, F. M. T. and Johansen, D. (eds), *Distributed Open Systems*, pp. 96–112. IEEE Computer Society Press, Los Alamitos, CA.

[5] Liskov, B. (1988) Distributed programing in ARGUS. *Commun. ACM*, **31**, 300–312.

[6] Triantafillou, P. and Taylor, D. J. (1995) The location-based paradigm for replication: Achieving efficiency and availability in distributed systems. *IEEE Trans. Softw. Eng.*, **21**, 1–18.

[7] Melliar-Smith, P. M. and Moser, L. E. (1998) Surviving network partitioning. *Computer*, **31**, 62–68.

[8] Zhou, W. and Molinari, B. (1996) A system for managing remote procedure call transactions. *J. Syst. Softw.*, **34**, 133–149.

[9] Open Software Foundation (OSF) (1990) *OSF Distributed Computing Environment Rationale*. Open Software Foundation, Cambridge, MA.

[10] Acevedo, B., Bahler, L., Elnozahy, E. N., Ratan, V. and Segal, M. E. (1995) Highly available directory services in DCE. In *Proc. Symp. on Principles of Distributed Computing (PODC'95)*. ACM Press, New York.

[11] Elnozahy, E. N., Ratan, V. and Segal, M. E. (1995) Challenges in building highly-available software using DCE. In *Workshop on Parallel and Distributed Platforms in Industrial Products, The 7th IEEE Symp. on Parallel and Distributed Processing*. IEEE Computer Society Press, Los Alamitos, CA.

[12] Gray, J. and Reuter, A. (1993) *Transaction Processing*. Morgan Kaufmann Publishers, San Mateo, CA.

[13] Helal, A. A., Heddaya, A. A. and Bhargrava, B. B. (1996) *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, Boston, MA.

[14] Triantafillou, P. and Taylor, D. J. (1996) VELOS: A new approach for efficiently achieving high availability in partitioned distributed systems. *IEEE Trans. Knowl. Data Eng.*, **8**, 305–321.

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

[16] Moser, L. E., Milliar-Smith, P. M., Agarwal, D. A., Budhia, R. K. and Lingley-Papadopoulus, C. A. (1996) Totem: A fault-tolerant multicast group communication system. *Commun. ACM*, **39**, 54–63.

[17] Goscinski, A. (1991) *Distributed Operating Systems: The Logical Design*. Addison-Wesley Publishing Company, Reading, MA.

[18] Zhou, W. and Molinari, B. (1990) A model of execution time estimating for RPC-oriented programs. In *Lecture Notes in Computer Science*, vol. 468, pp. 376–384. Springer-Verlag, Berlin.

[19] Bernstein, P. A. (1990) Transaction processing monitors. *Commun. ACM*, **33**, 73–86.

[20] Zhou, W. and Goscinski, A. (1997) Fault-tolerant servers for RHODOS system. *J. Syst. Softw.*, **37**, 201–214.

[21] Barborak, M., Malek, M. and Dahbura, A. (1993) The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, **25**, 171–220.

[22] Lampson, B. W. (1981) Atomic transactions. In *Lecture Notes in Computer Science*, vol. 105, pp. 246–265. Springer-Verlag, Berlin.

[23] Zhou, W., Hepner, P. and Wang, X. (1996) Using traders for loosely integrating heterogeneous database systems. In *Proc. 1996 Int. Computer Symp.: Int. Conf. on Distributed Systems, Software Engineering, and Database Systems, Kaohsiung, Taiwan, December 18–21*, pp. 25–32.

[24] ISO/IEC (1994) *Working Document–ODP Trading Function*. ISO/IEC JTC1/SC21 N8409.

[25] Zhou, W. (1996) Supporting fault-tolerant and open distributed processing using RPC. *Comput. Commun.*, **19**, 528–538.

[26] Zhou, W. and Wang, L. (1996) Parallel recovery in a replicated object environment. In *Proc. 3rd Australasian Conf. on Parallel and Real-Time Systems*, Brisbane, Australia, September–October 1996, pp. 172–177

[27] Umar, A. (1997) *Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies*. Prentice-Hall PTR, Upper Saddle River, NJ.

[28] Umar, A. (1997) *Object-Oriented Client/Server Internet Environments*. Prentice-Hall PTR, Upper Saddle River, NJ.