Replicated Transactions*

Tony P. Ng
Shepherd S. B. Shi

Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

A scheme to replicate transactions is described. Our scheme allows a k-replicated transaction to survive k-1 failures. No coordination among the k replicas is needed until one of them reaches the end and proceeds to abort the others. Consequently, our scheme avoids the overhead and delay caused by failure detection, reconfiguration, and synchronization during the k replicas' execution. We describe a robust commit protocol to choose the transaction replica that should be committed, and a procedure to choose the nodes on which a transaction replica is executed. The goal of the procedure is to maximize reliability.

1 Introduction

In this paper we consider the problem of replicating the execution of a computation in several nodes in a distributed system. When a computation is initiated by a user, several independent replicas executing the same code are started concurrently. When the computation finishes, all but one of the surviving replicas are aborted. Such replication has the obvious benefit that as long as at least one replica survives, the computation can be considered successful. There are several advantages to this approach of achieving reliability:

1. No coordination is required among the multiple replicas except at the end of the computation. Each replica may execute differently because of non-determinism, but eventually only one replica is retained.

2. During the computation, no time is wasted to detect failures and to reconfigure. A replica is not aware of the fate of others. The fastest surviving replica gets to abort the others. This property is important for applications that have timing requirements.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 gives an overview of our system model and our basic algorithm. Section 4 presents an algorithm to choose a set of nodes to participate in a distributed replica. The sets of nodes belonging to different replicas should be as small as possible and have minimal overlap to maximize reliability. Section 5 describes how we make sure that only one replica is committed, despite failures that may occur during our commit protocol. Section 6 is the conclusion.

2 Related work

The replication techniques to achieve fault tolerance can be classified into two categories. The first category uses passive replicas. In those schemes, only one primary replica will perform the computation. The rest of the replicas monitor the current state of the computation, the primary will periodically checkpoint its state to the other replicas. Tandem [Bar81], and ISIS [Bir86] are examples of this approach. Many other algorithms are published in the literature [DS80, Had82, KT87, Mos84, Ng88, Svo84].

In contrast to the passive replicas approach, the second category uses active replicas. The duplicated components in the active replicas approach perform the computation concurrently. Circus [Coo85] is an example that uses active replicas to achieve higher reliability. Procedures are replicated as a troupe and the replicas are called the troupe members. When a server troupe is called by a client troupe, every member of the server troupe will receive calls from every member of the client troupe. Since calls are duplicated, the server has to make sure that the operation is performed exactly once. Each member in the client troupe will receive multiple replies too, so the client is responsible for gathering and selecting the appropriate response. Circus' approach requires the operation to be deterministic so that the server troupe members will remain consistent.

Clouds [DLA88, ADLW87] is another system that uses the active replicas approach. It avoids the problem of non-deterministic computation by committing only one replica of a replicated computation. Each replica visits different object replicas of an object. The object state in the object replica visited by the committed replica is copied to other object replicas in the troupe to maintain consistency. Duplicate invocations or replies are also avoided because each member of a client troupe only invokes one member of a server troupe.

Our approach is similar to Clouds'. However, there are several components in our design that distinguish our work from theirs. First, we address the problem of how to select a replica to commit. The commit protocol has to be robust enough so that it will survive failures during the protocol and preserve correctness. It also has to be efficient so that long delays due to reconfiguration are avoided. Second, we address the problem of how to select the nodes that a replica should visit. An intelligent selection algorithm can improve the reliability of a replicated transaction signif-
3 System Model and Basic Algorithm

3.1 System Model

We model our system as a collection of objects. Each object has its own internal state and supports a set of operations. Object invocations may be nested. An operation may invoke other operations serially or in parallel. A computation is modeled as an invocation on a root object.

A computation is also cast as an atomic transaction [Gra78]. In other words, the invocation of the root object is the beginning of a transaction, and the return is the end. Operations invoked in a transaction are either performed successfully all together or not performed at all. We assume there is a transaction manager responsible for starting and ending the transaction. At the end of a transaction, there is either a commit or abort command issued by the transaction manager. Only after the transaction commits can the object state modification made by that transaction be seen by other transactions. Transactions are also serialized by some concurrency control mechanism [Gra78, Ree78, BG81].

Internally, an object may be replicated in several nodes as object replicas. Each object replica has the following components:

1. Shared state: it refers to the state of an object that captures the modifications made by successfully committed transactions.
2. Code: this component is read-only.
3. Buffered updates and locks: each object may have several transactions executing within concurrently. For each of these transactions, we allocate an area to keep track of the locks acquired and the updates made by that transaction. The buffered updates are installed in the shared state component when the transaction is committed.

In our model, nodes are fail-stop [Sch84] and crash independently. In addition, the network that connects nodes may lose or duplicate messages, deliver them out of order, but messages are not corrupted. We will first describe our algorithms assuming that there are no network partitions, then we will describe the changes necessary when partitioning is possible. We assume that a node can detect another node’s failure, either through periodic “I-am-Alive” messages, or through timeouts in their communication. By using a transaction model, we have also limited ourselves to computations that can be cast as transactions. Furthermore, since transactions are replicated and at most one replica is committed, any I/O performed outside the transaction boundary (e.g., terminal I/O) has to be buffered. Buffering can be avoided if the sender/receiver of such I/O activity is aware of the replication.

3.2 Outline of Proposed Scheme

In our proposed scheme, a transaction is executed as several concurrent and independent transaction replicas. We call the collection of replicas a replicated transaction. When a user submits the transaction, he has to specify the level of resiliency requested. A level of resiliency $k = 1$ means that the transaction will survive $k - 1$ hardware failures. A network partition between a group of communicating nodes can be regarded as “failures” of the nodes in the minority partition. The node that receives the transaction will try to locate $k$ transaction managers on behalf of the user. Each transaction manager is responsible for one transaction replica. The transaction managers are selected based on the locations of the root object replicas, nodes. The failure index is the day for a processor. The detailed selection algorithm will be described later.

Whenever a transaction manager is notified that it is selected, it can start the transaction immediately without waiting for other transaction managers. Eventually, the fastest transaction replica to be completed will have a higher chance of being committed.

When a transaction replica invokes a replicated object, the operation is executed on only one of the object replicas. If the transaction replicas of the same replicated transaction visit the same replicated object, they should coordinate among themselves so that each visits a different object replica. The question of how to map transaction replicas to object replicas is the subject of section 4. Choosing the locations of the transaction managers can be considered to be a special case of this problem.

After the invocation on the root object returns, the transaction manager will try to commit the transaction replica. A commit protocol is designed to ensure that only one transaction replica can commit. The proposed protocol is also able to handle any possible failures that might happen during the protocol.

4 Selecting an Object Replica

The basic idea of the object replica selection algorithm is to minimize the number of nodes visited by a transaction replica, visit the most reliable nodes, and minimize the overlap of nodes visited by two transaction replicas of the same replicated transaction. Intuitively, the fewer nodes that a transaction replica visits and the more reliable they are, the lower the probability that the transaction replica will fail. The less overlap between two transaction replicas, the lower the probability that more than one transaction replica will fail together.

In the calculations below, we will assume that a transaction replica can commit only when all (the nodes containing) the object replicas that it visited are alive. In addition, an object replica invoked must be able to collect a majority quorum among its replicas if network partitions are possible. Algorithms described in section 5 satisfy these assumptions.
In section 4.1 we will describe how a transaction replica selects an object replica when a replicated object is invoked, assuming that network partitions do not occur. In section 4.2 we will describe how network partitions will affect our decision procedure.

Finally, section 4.3 discusses the special case of choosing the root object replica locations, where transaction managers are located.

4.1 Selection of Object Replicas

For each object invocation, the caller object has to choose a replica of the callee object. For now we assume that each object visited by a k-replicated transaction has at least k replicas, so that each transaction replica can be matched up with a different object replica. An intuitive way is to choose an object replica randomly as in Clouds[DL85]. However, if the node containing the randomly chosen object replica is also visited by another transaction replica of the same transaction, both transaction replicas would fail if that node fails. To avoid this, we propose the following procedure to select a callee object replica:

1. Check whether there is a local (w.r.t. the caller) replica of the callee object. If there is one and it is not used by another transaction replica, the local object replica is chosen. Choosing a local copy reduces the number of nodes visited and communication costs.

2. If (1) fails to find a callee object replica, check the nodes that have been visited by this transaction replica. Choose an object replica on those nodes if there is one.

3. If (1) and (2) fail, choose a callee object replica whose node hasn’t been visited by any other transaction replicas of the same replicated transaction.

4. If none of the above finds an object replica, we can use any available object replica with the highest reliability (thus creating overlap between the nodes visited by this transaction replica and other replicas).

A small problem with the procedure above is that the decision made by step 3 is not always good. For example, if the object replica chosen in step 3 is on a relatively unreliable node, it may be better to choose some other object replica even though it creates overlap. The following attempts to correct this problem.

Suppose a transaction replica $t_i$ of a replicated transaction $rt$ has $m$ possible callee object replicas from which to choose: $a_n$, where $n = 1, 2, \ldots, m$. We use the following notation (in general, $P_n$ denotes the probability that $z$ fails, and $P_{ny}$ denotes the probability that $y$ fails given that $y$ has failed):

- $P_a$ = Probability that the replicated transaction $rt$ fails
- $P_{an}$ = Probability that $a_n$ fails
- $P_{ti}$ = Probability that transaction replicas $t_i$ through $t_j$ fail
- $P_{t(i_1,\ldots,i_n)}=P_{t(i_1)}\cdot P_{t(i_2)}\cdot \ldots \cdot P_{t(i_n)}$ = Probability that transaction replica $t_i$ fails given that transaction replicas $t_i$ through $t_j$ fail but $a_n$ has not failed $t(a_n)$ = the set of transaction replicas $i$ that have visited $a_n$
- $P_{t(i_1,\ldots,i_n)}-t(a_n)$ = Probability that all transaction replicas in the set $\{t_i, t_{i+1}, \ldots, t_j\}$ - $t(a_n)$ fail

If $a_n$ is chosen by $t_i$ as the callee object replica,

\[
P_{ti} = P(t_i) \cdot P_{t(a_n)} = 1 - P(t_i) \cdot P_{t(a_n)} + P(t_i) \cdot [P_{t(a_n)} - P_{t(a_n)}] = 1 - P(t_i) \cdot P_{t(a_n)} + P(t_i) \cdot [P_{t(a_n)} - P_{t(a_n)}]
\]

Since $P(t_i)$ and $P_{t(a_n)}$ are independent from the choice of $a_n$, $P(t_i)$ can be minimized by minimizing $P_{a_n} \cdot P(t_i) - n(a_n)$.

As $P(t_i) - n(a_n)$ (let it be $P_n$) is difficult to calculate in general, approximations can be used. For example, $P_n$ can be approximated by $(P_{a}^m)^n$, where $P_{a}$ is the probability that a "typical" transaction replica may fail, and $s(n)$ is the size of the set $\{t_1, \ldots, t_m\} - t(a_n)$. Alternatively, $P_n$ can be approximated with $c \cdot s(n)^{-1}$, where $c$ is some unspecified constant. To summarise, steps 3 and 4 of the object selection algorithm can be refined as follows: If (1) and (2) fail, choose an object replica $a_n$ with the minimum $P_n$.

In step 2 of our procedure, each transaction replica has to have remembered the list of nodes it visited. This list can be maintained either in a centralized or decentralized fashion. In the centralized approach, the transaction manager is responsible for keeping track of the list. However, for each object invocation, the caller object has to read and update the list with potentially remote operations on the transaction manager. The approach we propose is a distributed one. A visited-nodes-list is included in each object invocation and return. With this approach, only local information is needed in step 2. The list supplied with an invocation may only be a subset of the actual nodes visited if an object invokes several other objects concurrently (which may in turn invoke other objects). This inaccuracy is caused by each parallel invocation being unaware of the nodes visited by other invocations. We can either disregard these cases or require the transaction manager to provide coordination.

Our approximations of $P_{a}$ requires knowledge of $s(n)$. This can be obtained from $a_n$. We have also assumed in the procedure above that each node is aware of the locations of the object replicas of any object and the failure probability of each node. Such information can be obtained from a (replicated) location server and cached. An outdated cache may lead to a sub-optimal choice or a misdirected invocation. Outdated cache entries can be detected by the chosen object replica. An error message can be returned to the caller so that it can consult the location server again.

4.2 Network Partitions

In the algorithm presented above, we did not consider the possibility of the network partitions. Two changes are needed in the procedure above to accommodate network partitions. First, $P_{a}$ includes the probability that the node on which $a_n$ resides fails, as well as the probability that $a_n$ will be partitioned from the node where the transaction manager of $t_i$ resides. Second, since failure of $a_n$ can be caused by either a node failure or a network partition, $P(t_i)$ - $n(a_n)$ is not equal to $P(t_i) - n(a_n)$ in general.

Suppose node $a_n$ is the node on which $a_n$ resides, we can divide $P_{a_n}$ into three parts:

\[
P_{a_n} = P_{a_1} + P_{a_2} + P_{a_3}
\]
where
- \( a_{n1} \) is the event that \( node(a_n) \) fails and \( node(a_n) \) is partitioned from the transaction manager of \( t \);
- \( a_{n2} \) is the event that \( node(a_n) \) fails and \( node(a_n) \) is not partitioned from the transaction manager of \( t \);
- \( a_{n3} \) is the event that \( node(a_n) \) does not fail and \( node(a_n) \) is partitioned from the transaction manager of \( t \);
- \( P_{a_{n1}} \) = Probability that \( a_{n1} \) happens
- \( P_{a_{n2}} \) = Probability that \( a_{n2} \) happens
- \( P_{a_{n3}} \) = Probability that \( a_{n3} \) happens

\[
P_{t(a_{n1})|a_n} = r_1 \cdot P_{t(a_{n1})|a_n} + r_2 \cdot P_{t(a_{n1})|a_n} + r_1 \cdot P_{t(a_{n1})|a_n}
\]

\[
\approx r_1 \cdot P_{t(a_{n1})|p(a_n)} + r_2 \cdot P_{t(a_{n1})|p(a_n)} +
\]

\[
r_3 \cdot P_{t(a_{n1})|p(a_n)}
\]

where
- \( r_1 = P_{a_{n1}} / P_{a_n} \)
- \( p(a_n) \) is the set of transaction replicas in \( t \) that are likely to fail given that \( a_n \) is partitioned from the transaction manager of \( t \).

4.3 Selection of Transaction Manager Locations

For a \( k \)-replicated transaction, \( k \) replicas of the root object must be chosen to run the transaction managers, which in turn start the transaction replicas by invoking the local root object replicas. Since the transaction replicas have not visited any other nodes yet, the procedure described above is reduced to choosing the \( k \) most reliable nodes.

5 Commit Protocol

When a transaction replica finishes execution, control is returned to the transaction manager at the root object. The transaction manager asks each of the object replicas it visited to prepare. In the first phase, a commit protocol is initiated to commit this transaction replica. The commit protocol consists of three phases. In the first phase, the transaction manager asks each of the object replicas it visited to prepare. In the second phase, a mutual exclusion protocol is executed to guarantee that only one transaction replica is committed. In the third phase, the object replicas are told that their transaction replica has been committed or aborted.

From sections 5.1 to 5.4 we describe our commit protocol assuming that there are no network partitions. In section 5.5 we describe the changes needed to handle network partitions. Finally, section 5.6 discusses the performance of our protocol.

We assume that shared state in an object replica is guarded by 2-phase read/write locks\cite{Graham78}. Since a transaction replica executes on only one of the object replicas, concurrent transaction replicas can acquire the same lock but on different object replicas. This is acceptable if the concurrent transaction replicas belong to the same replicated transaction, as only one of them will be committed. However, if they belong to different replicated transactions and both are committed, inconsistencies may result.

There are two solutions to this problem. First, locking can be made into distributed operations. A transaction replica executing at a particular object replica must acquire locks at all the other (surviving) object replicas. A lock can be acquired at the same time by multiple transaction replicas only if they belong to the same replicated transaction. This solution has the disadvantage that it requires remote operations whenever a lock is needed. The second solution requires the commit protocol to detect such lock conflicts. Assuming lock conflicts are rare, we will adopt such a solution and describe how the conflicts can be detected in phase 1 of the commit protocol below.

5.1 Phase 1

During the first phase, prepare messages are sent by the transaction manager to each of the object replicas visited. On receiving a prepare message, an object replica should send its buffered updates to the rest of the replicas of this object. Only after receiving buffered messages from all the other surviving object replicas should this replica return a prepared message to the transaction manager. By storing the buffered updates at all the object replicas, we are guaranteed that a prepared \( k \)-replicated object can survive \( k-1 \) node failures. The replication of buffered updates can be considered as a form of stable memory\cite{Lam80}. Note that multiple transaction replicas of the same replicated transaction may be preparing concurrently, thus an object replica may have both sent out and received buffered updates. Eventually only one transaction replica will be committed.

Conflicting transaction replicas belonging to different replicated transactions are checked during phase 1. When an object replica receives buffered updates from another replica or when a local transaction replica tries to prepare, conflicting locks exist and the conflicting transaction replicas belong to different replicated transactions, the currently preparing transaction should be blocked (but not aborted yet). If no conflicts exist, arriving buffered updates are stored locally and tagged with the identifier of the originator. A buffered message is returned. Occasionally, two concurrently preparing transaction replicas (belonging to two different replicated transactions) may be dead-locked because they prepare at two replicas in different orders. This deadlock can be detected and resolved using any deadlock detection algorithm\cite{CMS82, Obel82}.

5.2 Phase 2

When a transaction manager has received prepared messages from all the object replicas visited, it should enter into a mutual exclusion protocol. Transaction managers working for the same replicated transaction use the protocol to ensure that only one transaction replica is committed.

Our mutual exclusion protocol imposes a static ordering among the transaction managers. To obtain the permission to commit, a transaction manager has to send enter_mutex messages to transaction managers with higher priorities. When a transaction manager receives the enter_mutex message, it returns an okay message if it is not in phase 2 itself. After sending the okay message, a transaction manager is blocked from entering phase 2 itself since it has already given the permission to a manager with lower priority. A transaction manager possesses the privilege to commit when it receives okay messages from all
managers with lower priorities will not receive an okay message to commit. It will commit when a higher-priority surviving manager is not committing, and all managers with higher priorities than it must be blocked from entering phase 2, and all managers with lower priorities will not receive an okay message from t.

Our algorithm also guarantees that at least one manager will commit as long as not all managers fail. The correctness of the argument below depends on the assumption that a manager cannot be blocked by a failed manager. We will show how this can be achieved in section 5.4. Consider the highest-priority surviving manager t. If t is not committing, it must be blocked from entering phase 2 by a surviving manager t2 with a lower priority. Since t2 has obtained an okay message from t1, it must have already entered phase 2. If t2 does not commit, it must have failed to obtain an okay message from a higher-priority surviving manager t3. Since t3 refuses to send an okay message to t1, it must have already entered phase 2. Consequently, t3 cannot be t1. Similarly, t4 either will commit or is blocked by a higher-priority surviving manager t5 which is not t1. Since there are a finite number of surviving managers, eventually one manager will not be blocked by any surviving managers.

5.3 Phase 3

After acquiring permission to commit, a transaction manager t sends out entered messages to other transaction managers. The latter acknowledge by returning congratulations messages. This exchange of entered and congratulations messages serves as a form of stable memory to remember t's decision to commit. It allows other transaction managers to finish phase 3 on behalf of the committing transaction manager, in case the latter crashes.

After receiving all the congratulations messages, commit messages are sent to all the participating object replicas. At the object replicas, all conflicting transaction replicas (regardless of whether they belong to the same replicated transaction or not) are aborted. Buffed updates from the committing transaction are installed into the shared state. The commit message is also propagated to other replicas of these objects and the same actions are taken there. Ack messages are returned by the object replicas to the transaction manager after the local actions and the propagation are performed.

At other transaction managers, their transaction replicas are aborted when their root object replicas receive the entered messages. However, they still retain a record of the committing transaction replica, in case the latter is interrupted by a failure before sending all the commit messages. These records can be deleted when the committing transaction manager sends its peers done messages, which are sent only after ack messages are received from all the object replicas visited.

5.4 Failures during Commit Protocol

Failures can happen before and during any of the three phases of our commit protocol. If one of the object replicas fails before sending its buffered updates to any other object replicas, the transaction replica in which it participates will fail to prepare and abort. If the object replica fails afterwards, another object replica that has received the buffered updates has all the necessary information to finish the commit protocol on behalf of the failed replica.

We assume that the failure of a transaction manager t is detected by another transaction manager(s). The latter would ask the highest-priority surviving transaction manager t' to finish on behalf of t. t' guarantees that if t has sent any commit messages, t' will finish sending those messages on behalf of t.

When t' is asked to continue on behalf of t, t' should follow the algorithm below:

1. If t' has received the done message from t, t' would have
deemed all records of this replicated transaction. Done messages can be sent to all surviving transaction managers.

2. If the root object replica at t' has already received a commit message, then t must have already received all the congratulations messages and t' can finish on behalf of t by resending the commit and done messages. The set of object replicas visited by t can be determined if a list is included in the entered messages sent by t originally.

3. Determine the set of surviving managers that have received an entered message from t. If this set is not empty, t' can finish on behalf of t by resending the entered, commit and done messages.

4. Otherwise, t can be aborted because it must not have entered phase 3. It may have acquired okay messages from some managers of higher priorities; those managers can be informed to assume they have never sent any okay messages. Some managers of lower priorities may be blocked waiting for an okay message from t; those managers can be informed to ignore t and proceed. Finally, t may have visited some object replicas; these object replicas can find out the fate of t from t'.

If t' fails while finishing on behalf of t, the next highest-priority surviving transaction manager can take over. Our algorithm above guarantees that either all or none of the commit messages will be sent. As long as not all transaction managers fail, at least one will finish on behalf of t.

5.5 Handling Network Partitions

To handle network partitions, voting techniques[GI79] can be used to guarantee that only one partition would be able to proceed.

Phase 1 Previously in section 5.1 we have assumed that an unreachable object replica must have failed. Thus only the surviving replicas need to be prepared. With network partitions, each object replica must propagate the updates to a write quorum of the replicas, and the read locks to a read quorum. When a read lock is acquired, the receiving object replica has to verify that the version read by the preparing transaction replica is at least as recent as the local version. Network partitions make the phase 1 of our commit protocol less likely to succeed. A transaction replica can be aborted because it read an outdated version. Remotely locking a read quorum before proceeding to read alleviates this problem.

Phase 2 Instead of favoring higher-priority managers, our mutual exclusion protocol should favor managers that are able to finish phase 1. Choosing such managers increases the chances of committing a transaction replica, since a high-priority manager may be partitioned from the object replicas visited in phase 1 and from other managers. To avoid the majority of managers blocked by an isolated high-priority manager, we require each manager to send enter_mutex messages to all other managers. A manager possesses the privilege to commit when a majority of the managers return okay messages. A manager can send at most one okay message. (A manager entering into phase 2 can be regarded as sending itself an okay message.)

The algorithm above obviously guarantees mutual exclusion because there are at most k okay messages and a majority requires more than k/2 messages. Unfortunately no managers will be able to acquire a majority if they finish phase 1 closely in time, even when no network partitions have occurred. To alleviate this problem, we re-introduce the concept of priority: a manager is allowed to “transfer” the okay messages it acquired to the highest-priority manager with which it can communicate. Our algorithm usually commits the highest-priority manager in a group of managers that enter phase 2 concurrently and is in the majority partition.

Phase 3 Entered messages serve as a form of stable memory. The more nodes receive the entered messages, the less likely will a transaction manager t be left partially committed. There is no reason why all k managers have to remember t’s decision, nor why the decision cannot be remembered in some other set of nodes (chosen a priori). Since phase 2 requires t to obtain only more than k/2 okay messages to achieve mutual exclusion, we can either:

1. reduce the number of congratulations messages needed by t to be at least k/2 (as long as more than k/2 failures are highly unlikely), or
2. remember the entered messages in some other set of nodes.

We will assume the first option in our discussion below.

Handling Failures Failures can be handled as described in section 5.4 except that a “failure” of t may be caused by network partitions. t' can finish the commitment on behalf of t if the number of managers that have received entered messages from t is non-zero. However, t' would still have to make sure that number is more than k/2 before it can start sending commit messages. On the other hand, if the number of managers that have not received entered messages is more than k/2, t' can abort t. This prevents a majority of managers from being blocked by a minority. If neither conditions are met, t' has to wait until it is reconnected with more managers. If less than k/2 managers have failed, the surviving managers would be reconnected eventually to decide on the fate of t.

5.6 Performance

The cost of a replicated transaction is comparable to k times the cost of an unreplicated transaction. Before the commit protocol, each transaction replica accesses only one replica of an object. The cost of sending the buffered updates in phase 1 and the cost of sending entered/congratulations messages in phase 3 are really costs of using a special form of stable memory. These phases correspond to phases 1 and 2 in a typical two-phase commit protocol[Gra78], in which a non-replicated transaction would need to perform preparation and remember its commitment using stable memory. Although our stable memory access costs may not be the same as those using a traditional implementation of stable memory, we believe that they are of the same order of magnitude. Compared to a non-replicated transaction, phase 2 is the extra work that needs to be performed by a transaction replica. However, it should be noted that a high-priority manager sends
relatively few messages in phase 2. Furthermore, if it finishes relatively quickly, it will stop other managers from incurring extra message costs. In the worst case, the total number of message sent by all $k$ managers is $O(k^2)$. Since $k$ is expected to be small, $O(k^2)$ messages are not prohibitively expensive.

6 Conclusion

We have described an algorithm to provide reliable transactions in this paper. Our approach of executing independent transaction replicas consumes more resources than a passive replica approach (checkpointing). However, no failure detection, reconfiguration, or synchronization are necessary until the transaction replicas commit. Consequently, a transaction can meet deadlines despite failures. We have worked out a commit protocol that works correctly despite failures during the protocol. Furthermore, we have provided a procedure to determine how a transaction replica chooses an object replica to invoke.

References


