Distributed Software Transactional Memory

Abstract

The present report describes the implementation details of a fault-tolerant, replicated, distributed software transactional memory. Performance of protocol is evaluated through latency and bandwidth required, overall throughput, load balancing of data storage and processing. Furthermore, a critical assessment of the quality of the protocol and future work is also provided.

1. Introduction

Due to the cloud computing revolution in computing, today's world has increasingly moved to cloud-based services rather than maintaining their own infrastructure. The cloud service providers are well equipped with the hardware and software infrastructure to provide better services by applying concepts of *Distributed Computing*.

In this model of computing, one of the main problems that providers face is concurrent transactions; how the system behaves when multiple users access/manipulate an object at the same time; how to maintain the proper order of commit and how to resolve conflicting operations. The paper describes the solution for handling transactions in a replicated, fault-tolernt, distributed system.

Section 2 introduces an overview of each scenario. Section 3 discusses the algorithms used for concurrency control, transaction management and Client-Server communication

2. Architecture

The assumption made for all the sections below is that the Master is a perfect process. The names Object Servers and Servers are used interchangeably and refer to any non-Master and non-Client process that participates in a transaction. The Master also acts as the*Coordinator* for handling transactions and *Replica Manager* for eagerly replicating objects among the Object Servers. The main assumption is that the Master does not fail, only Object Servers do.

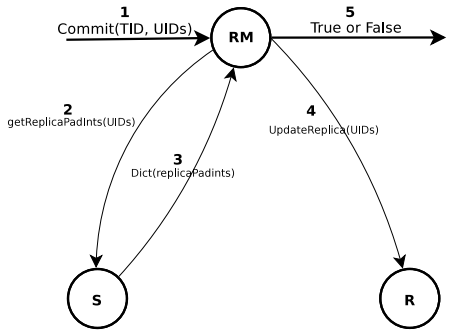


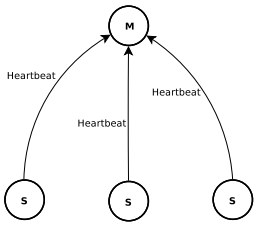
Figure 1: RM: Replication Manager; S: Object Server; R: Replica Server

Subsection 2.1 details the replication steps when a commit occurs, Subsection 2.2 lists the details of redistribution of objects among servers after a node fails or joins. Further details on the reponsibilities of the Coordinator/Master are in Subsection 2.3.

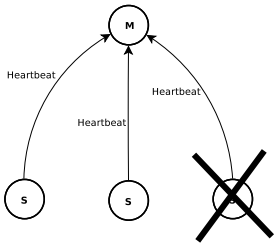
2.1. Replication

As illustrated in Figure **??**, the Master server *(M)* is used by the Object Servers *(S)* when bootstrapping. Information such as IP address and port number of the Server is stored by the Master which, then, issues unique identifiers for each Server. Once the Object Server (S) has been issued an identifier and notified the Master of joining the group of available servers, it can be connected by Clients *(C)* for handling transactions. Transaction identifiers (TID) should be unique to the whole system therefore the Master server (M) will keep track of the global sequence of TIDs.

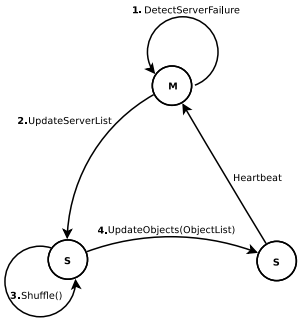
When the Coordinator (depicted as the Replica Manager in Figure 1) receives a `Commit` call along with the Transaction ID, and the list of UIDs, it is calling the workers' `GetReplicaPadInts` to retrieve a dictionary of PadInt objects which is used to call the replica's `UpdateReplica` method. Once the replication is completed, the Coordinator returns from the transaction and notifies the client of the commit's result (True or False).



(a) Periodic heartbeats sent to the Coordinator



(b) Failure of one of the Object Servers



(c) Reshuffling of objects

2.2. Group Membership Change

Since the Master server also assumes the role of a Failure Detector - refer to Figure 2a, upon observing a node failure, it updates its list of available workers and pushes it to the workers. The workers, upon receiving the list initiate the*redistribution* of objects procedure.

Once the server receive the new group membership list, it redistributes the objects among the other servers while new TIDs cannot be issued during this time. Figure 2c displays the steps occuring during such a redistribution of objects.

2.3. Coordinator/Master responsiblities

The Coordinator's responsabilities and interface are different for the two scenarios (perfect and fault-tolerant) detailed in Section 2. For instance, assuming perfect links and processes, the Coordinator can be any of the Servers, thus reducing the Master's load. The list of servers available to the Client is supplied by the Master and it's updated with every server group membership change.

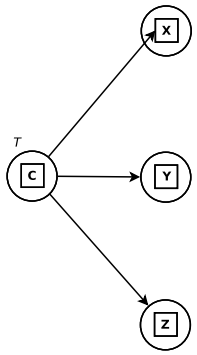


Figure 4: Flat Transaction model between a Client and 3 Servers

The Coordinator's responsibilities below are shared in both scenarios:

- assign a transaction identifier to the Client
- handle abort and commit commands from the Client or Object Servers
- interface with the participant servers in the transaction during the voting and commit phases

The Master also assumes the role of a failure detector through the use of heartbeat messages. In the fault-tolerant environment, the Master also assumes the Coordinator role.

3. Algorithms

3.1. Communication

The following sequence diagram depicts the communication between the client, coordinator and object servers, assuming no failures or joins (any group membership change).

3.2. Transaction Management and Concurrency Control

The *Flat Transaction* model allows for a client to manipulate objects on multiple servers in a single transaction. With regards to concurrency control, *Timestamp Ordering* will be used for this project. Figure 4 illustrates a flat transaction model setup with a transaction being executed from the client on 3 servers.

The advantages of choosing *Timestamp Ordering* over *Two-Phase Locking* or *Optimistic Concurrency* options are the following:

- Deadlock prevention - common with the use of locks
- Better performance for transactions with predominantly *read* operations [**?**]
- Faster conflict resolution when compared to locking - transactions are aborted immediately.

Figure 5 illustrates the steps between the Client, Coordinator, Master and Object Serves during a transaction. The

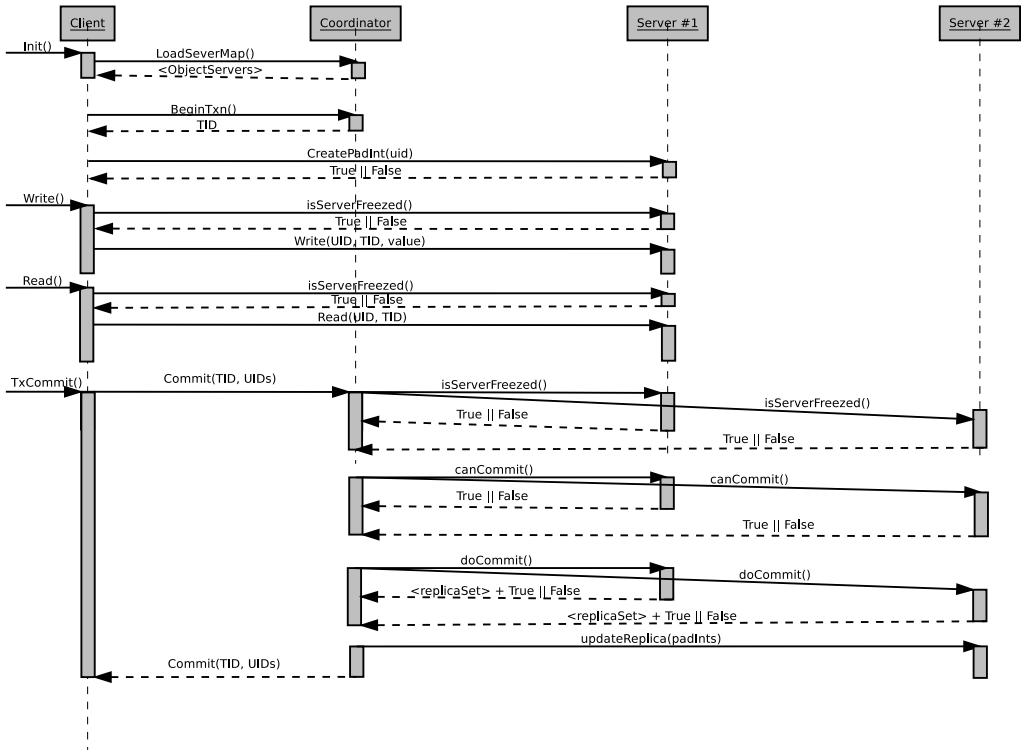


Figure 3: Communication between the client, coordinator and servers during a transaction

steps below offer more details on the communication that occurs in a perfect setup:

- **Steps 3-4:** Client acquires transaction ID from the Coordinator
- **Step 5:** Client passes TID to the Object Server
- **Step 6:** Object Server calls the Coordinator's *inteface*
- **Steps 7-8:** Client manipulates objects directly on the Object Server
- **Step 9:** Upon transaction end, Client asks the Coordinator to either *Abort* or *Commit*
- **Step 10:** Coordinator will request each participant server in the transaction to indicate whether it can commit a transaction or not.*(Voting Phase)*
- **Step 12:** If all participants answer/vote *positively*, the Coordinator issues *doCommit(TID)* for each parcipant to commit its part of the transaction
- **Steps 13-14:** Once completed, all servers acknowledge the commit and Coordinator notifies the Client that it's successful
- Should any of the participant servers be unable or disagree to commit and aborts, the Coordinator will request all the remaining participants to abort. Client will then be notified.

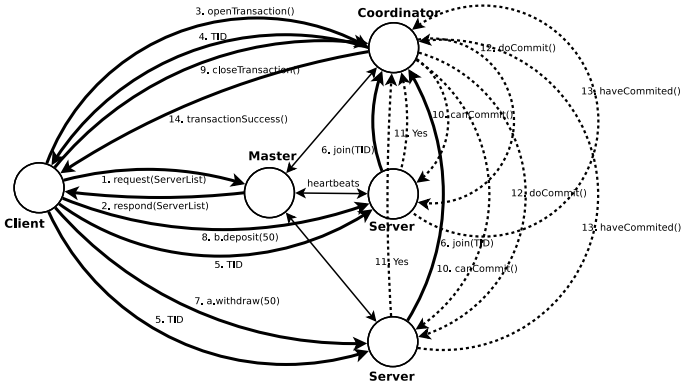


Figure 5: Transaction steps between the Client, Master, Coordinator and Servers. The dotted lines represent the 2-phase distributed commit

3.3. Timestamp Ordering and Deadlock Detection

Due to the use of timestamp ordering for concurrency control, there is limited probability of deadlock occurance. Each transaction is assigned a unique timestamp on start. Each operation in a transaction is validated when it is carried out. Should a transaction fail the validation it is immediately aborted, but can be restarted by the Client. The Client is issued with a globally unique transaction timestamp by the Coordinator. The servers are jointly responsible for ensuring serial equivalence. For instance, server S1 access an object before S2, thus server S1 appears before S2 for all objects. The Coordinators must agree on timestamp ordering to maintain all servers synchronised with the same ordering. The timestamp consists of a *(local timestamp, server-id)* pair and is kept synchronized by the use of local physical clocks coordinated by the Master.

Conflict resolution is performed at each operation. The basic timestamp ordering rule is based on operation conflicts and is very simple: *A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transaction.*[**?**]. Each transaction has its own tentative version of each object it accesses, such that multiple concurrent transactions can access the same object. The tentative versions of each object are committed in the order determined by the timestamps of their transactions by transactions waiting, if necessary, for earlier transactions to complete their writes. Since transactions only wait for earlier ones (thus, no cycle occurs in the wait-for graph), no deadlocks occur.

Conflict resolution is performed at each operation. If the resolution is to abort the transaction the Coordinator is informed and it will abort the transaction for all the participants.

3.4. Replication

The fault tolerant design supports failure of a single Object Server. It extends the perfect architecture by assuring the replication of objects in the adjacent server. For instance, the object stored in Object Server S1 keeps a copy in S2 as a replica. The Master server is responsible for keeping track of the replica servers.

When an Object Server **fails**:

1. Master detects the failure from the absence of heartbeat messages from the server
2. Master server issues a *"failure warning notification"* to the other servers, so that the current transactions related to the absent server be aborted. Others will continue but new transactions will not be accepted. (Master no longer issuing TIDs)
3. Master coordinates the object replication so that the currently available servers share the objects held by the absent server.
4. Once the Master notifies all the servers about the new, stable status, transactions can be take place again.

When the failed server **recovers**:

1. Master detects the arrival of a new server
2. Master server issues a *"New server notification"* for the servers. Once the current pending transactions are completed, servers are stabilized
3. Master server coordinates object replication such that a portion of objects in the current system will be transferred to the new server
4. Once the Master notifies all servers of the stabilized status, transactions can be resumed

4. Conclusion

Across the sections we discussed the solutions proposed in order to implement Distributed Software Transactional Memory System. Our solution addresses two scenarios: perfect links and processes and a fault tolerant system. A very important goal has been to provide a simple solution which would eliminate conflicts in transactions without encountering the inefficiency of a distributed environment. The sections and subsections address each of the two architectures individually in order to provide a clearer distinction of responsibilities and complexity of the system. Furthermore,*Timestamp Ordering* based concurrency control aids in eliminating local and global deadlocks which represents one of the major issues that need to be addressed in DSTMs.

Finally, we believe that the concepts we came across in the discussion of this paper builds a firm foundation for implementing this solution in an efficient manner.