Distributed Software Transactional Memory

Abstract

The present paper describes the details of a distributed software transactional memory implementation. There are 2 scenarios presented for a Master/Server architecture:
a) perfect links and processes (Master and Servers)
b) perfect links and Master, but recoverable Servers

Transaction management, concurrency control, failure detector and replication mechanisms are described for both perfect and fault-tolerant architectures.

1. Introduction

Due to the revolution in the computing world and marketing strategies, todays world has moved to cloud based services rather than maintaining their own networks. The cloud service providers are well equipped with the hardware and software infrastructure to provide better services because of the aspect of Distributed computing.

In this model of computing one of the main problems that the providers has to face is the concurrent transactions. How the system behave when multiple users access the same object at the same time and manipulates them. How to maintain the proper order of commit and how to resolve conflicting manipulations. Here in this paper we are going to address this problem deeper and propose a solution which can be implemented and evaluated during next few weeks ahead.

In the following sections the implementation of a distributed software transactional memory library is presented for two scenarios:
a) perfect Master, Servers and links
b) perfect Master, links and recoverable, sequentially

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

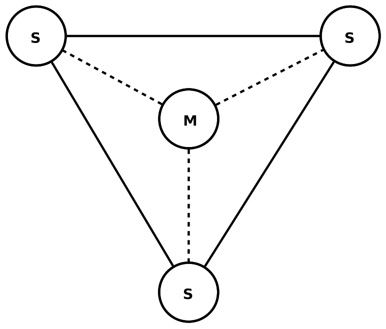

Figure 1: Master/Server entities in a perfect setup

2. Architecture

As illustrated in Figure **??**, the architecture consists of the following perfect entities:

- **M** - Master server
- **S** - Object server
- **C** - Client

We are proposing the system in two phases, where the first phase we assume all the servers and links never fails. The second phase will be an extension to the first phase so that we achieve the fault tolerant behavior in the system. Note: In the fault tolerant version we assume only one Object Server (S) to be failed at a time. This failure is not a byzantine failure but only a power crash. Once an object server is failed it will take some time to come back to a stable state and during that time period no new transactions are initiated. Below figure 1 illustrates a view of the system where three object servers (S) have connected to the Master Server (M).

2.1. Phase One, Perfect System

According to figure 1, master server (M) being a centralized server in the system, helps the object servers (S) to bootstrap the system. When bootstrap, the master server will keep information such as IP address, port of the bootstrapping server and issue valid references so that master can keep track of them. Once Object Servers (S) have joined the system the Clients (C) can connect and start 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.

Clients (C) are supposed to query the Master Server (M) if the local cache of the client is out dated or the Object Servers (S) in the cache do not respond. The Master Server (M) will provide the latest list of Object Servers(S) so that clients can continue operation.

Objects that are going to be stored in Object Servers (S) have unique identifiers (integer) so that each object can be uniquely identified throughout the system. Therefore in this solution we are using the modulo function against the number of Object Servers (S) available in order to find the actual location of the object to be stored or referred.

The clients (C) can connect multiple Object Servers inside a single transaction, hence it is required to have a coordinator in order to manage the transaction operations such as commit and abort. Therefore we consider the Master Server (M) as the coordinator. Once a Client (C) needs to manipulate an object it should ask the relevant object server to contact the coordinator (M) and join for the transaction. Then the manipulation operations (Read/Write) can be done by only contacting the Object Servers (S).

2.2. Phase two, fault tolerant system

This is an extension to the above proposed system, therefore in order to recover from failures, it is required to have an exact replication of the objects in a certain Object Server (S) in another Object Server (S). The Master Server (S) has to take care of the replica location of each server. Here we propose the replica location as a ring, where replica of S1 contains in S2, and replica of S2 contains in S3, etc. Furthermore each Object Server (S) is notified with its replica, by the Master Server (M). The transaction modal is similar to the phase one, the only extension required is to ask the replica server as well to join the coordinator for the transaction.

In case of a failure, the Master Server (M) will identify it by the drop of heart beat messages from the server and Master Server would issue a notification to all the Object servers (S) regarding the failure and it will block the issuing of TIDs for the clients until the failure recovered. Once the current transactions is completed, the master will rearrange the system in such a way that the system become stable with the correct replica locations as mentioned in the above paragraph.

At the fault recovery, the Master Server (M) will issue a notification to the Object Servers (S) asking to rearrange, so that the Object Servers (S) go through the list of objects which it is responsible and apply the modulo function again according to the new Object server count. All the Object Servers (S) process this action simultaneously and object movement among servers will rearrange the system. Once this is completed the Master Server (M) will issue another notification to the Object servers, so that they can invalidate the current replica object set they are holding and replicate the new object set. After this replication step the system becomes stable and the Master Server (M) starts to issue TIDs again. Furthermore, if a new server is added to the system, exactly the same steps mentioned above will be followed by the system.

2.3. Coordinator/Master responsiblities

The Coordinator's responsabilities and interface are different for the two scenarios (perfect and fault-tolerant) detailed in Section **??**. 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.

The responsibilities below are shared for both scenarios:

- **M** - Master server
- **S** - Object server
- **C** - Client

3. Algorithms

In the subsections below, algorithms for communication, concurrency, deadlock detection and transaction management are enlisted.

3.1. Communication

The following communication occurs **before** the Client attempts connection to the Master node - between the Master and the Servers.

1. Master boots, followed by the Object Servers bootstrapping
2. During Object Servers bootstrap:
    - Master identifies the Object Servers trying to connect
    - Master assigns a unique name to the Object Servers to later be used as a reference
3. Object Servers's heartbeat messages periodically being sent to Master. Heartbeats are also used for replica management in the faul-tolerant scenario. Should one of the replicas fail, the objects are replicated in another server in order to maintain a certain level of availability in the system.

Upon *attempting a transaction* the following communication exists between the Client, Master and Object Servers in a **fault-tolerant** environment:

1. Client connects to the Master and requests a transaction identifier
2. Master maintains a global transaction sequence and returns a unique transaction identifier (TID) to the Client. Furthermore, a list of healthy Object Servers is also returned to the Client.
3. Client generates a unique ID (integer) when generating the object and also be used in order to select an available server from the list (used in a modulo function)
4. Client uses the modulo result to connect to the Object Server and store the object representing the tentative versions of the transaction.

The main difference between the above steps and the communication in a **perfect** environment is that the TID is not necessarily assigned by the Master, but any of the Servers.

When the Client attempts to **manipulate an existing object** in the *perfect* environment, the following communication takes place:

1. Client requests and retrieves TID from Master
2. Client retrieves object's previously generated unique ID
3. Client applies modulo on the retrieved ID to locate the relevant Object Server to request for reference
4. With the retrieved object reference, client can manipulate the object value

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 **??** 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 **??** illustrates the steps between the Client, Coordinator, Master and Object Serves during a transaction. The steps below further details the communication that occurs between processes in a perfect setup:

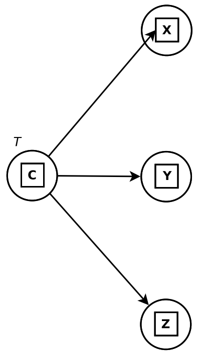

Figure 2: Flat Transaction model between a Client and 3 Servers

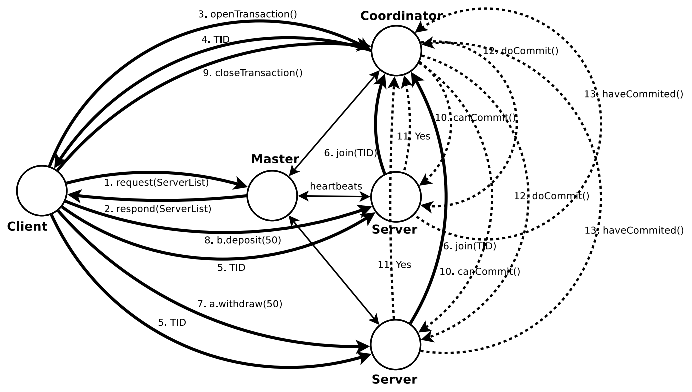

Figure 3: Transaction steps between the Client, Master, Coordinator and Servers

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

3.3. Timestamp Ordering and Deadlock Detection

Since timestamp ordering will be used for concurrency control the chances of deadlock appearing are eliminated. Each transaction is assigned a unique timestamp value when it starts. Each operation in a transaction is validated when it is carried out. If a transaction fails the validation, it is aborted immediately and can then be started by the client.

The client is issued with a globally unique transaction timestamp by the first coordinator accessed by a transaction. The servers are jointly responsible for ensuring serial equivalence i.e. if server S1 access an object before server S2, then server S1 is before S2 at all objects. The coordinators must agree on timestamp ordering so as to achieve the same ordering at all the servers. The timestamp will consist of a *¡local timestamp, server-id¿* pair. Timestamps can be kept roughly synchronized by the use of synchronized local physical clocks coordinated by the master.

The basic timestamp ordering rule is based on operation conflicts and is very simple: *A transactions request to write an object is valid only if that object was last read and written by earlier transactions. A transactions request to read an object is valid only if that object was last written by an earlier transaction.*[1]. 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, when necessary, for earlier transactions to complete their writes. Since transactions only wait for earlier ones (and no cycle could occur in the wait-for graph), no deadlocks occur.

Conflict resolution will be performed at each operation. If the resolution is to abort the transaction then the coordinator will be informed and it will abort the transaction at all the participants.

3.4. Fault Tolerant Algorithm

When the object server **fails**,

1. Master detects the server failure from the absence of the heartbeat message from the object server.
2. Master server issues a *failure warning notification* for the servers, so that the current transactions related to the absent server will be aborted. Others will continue but new transactions will not be proceeded. (Master block providing TIDs)
3. Master server coordinates 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 stabilized status, transactions can be continued 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, stabilizing the servers occur.
3. Master server coordinates object replication so that the currently available servers share the objects held by the absent server.
4. Once the master notifies all the servers the stabilized status, transactions can be continued again.

4. Footnotes

4.1. References

1. George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, USA, 5th edition, 2011.

4.2. Conclusions