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

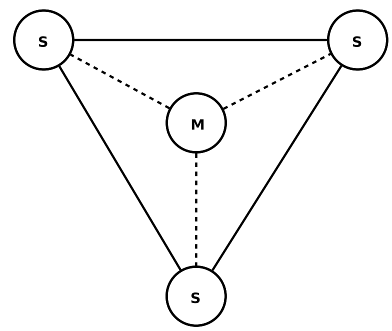

Figure 1: Master/Server entities in a perfect setup

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 across the sections below and refer to the any non-Master process participant in a transaction. Main differences between the two architectures, perfect and fault-tolerant, are the following:

- for perfect, the Coordinator of a transaction can be any Server, not necessarily the Master
-

2.1. Perfect links and processes

As illustrated in Figure 1, the Master server (M) is used by the Object Servers (S) when bootstrapping. Information such as IP address, the port of the Server is collected 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, they 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.

Note: the number of Object Servers can vary.

Clients (C) will query the Master (M) if the their cache is outdated or trying to reach the Object Servers (S) times out. The Master (M) provides the Client the updated list of available Object Servers (S) to be used for establishing transactions.

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 to multiple Object Servers inside a single transaction, hence it is required to have a Coordinator that manages the transaction operations such as commit and abort. The Coordinator can be any Server from the list provided by the Master in the current, perfect, scenario. On the other hand, in the fault-tolerant scenario, we consider the Master (M) as the coordinator. When a Client (C) needs to manipulate an object it connect to the relevant Object Server, which will contact the Coordinator (M) and join the transaction. Then the manipulation operations (Read/Write) are handled only by the Client and the Object Servers (S).

2.2. Fault Tolerant system

For recoverability in case of failure, an extension to the above system is necessary. Having a replication of the transaction's objects in a particular Object Server (S) across other Servers (S) is necessary to assure recoverability. The Master (S) handles the replica location of each Server. Our proposal for the replica locations is a conceptual ring, where S1's replica is stored in S2, and that of S2 in S3, etc. The Master notifies each Ssever of its replica location. 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.

The Master server (M) also assumes the role of a *failure detector*, by using the heartbeat messages regularly exchanged between the Master and the Server. In case of an Object Server(S) failure, the Master server(M) will identify it through heartbeats and issue a notification to all the other Object Servers (S) about the failure. No further TIDs are issued for the clients until the failure recovered. Once the current transactions complete, the Master will stabilise the system with the correct replica locations.

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 the object movement among servers will stabilize the system. Once the system is stable, the Master (M) will issue another notification to the Object Servers to invalidate the current replica object set 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 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.

The responsibilities below are shared for 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

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

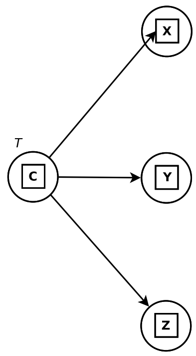

Figure 2: Flat Transaction model between a Client and 3 Servers

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 2 illustrates a flat transaction model setup with a transaction being executed from the client on 3 servers.

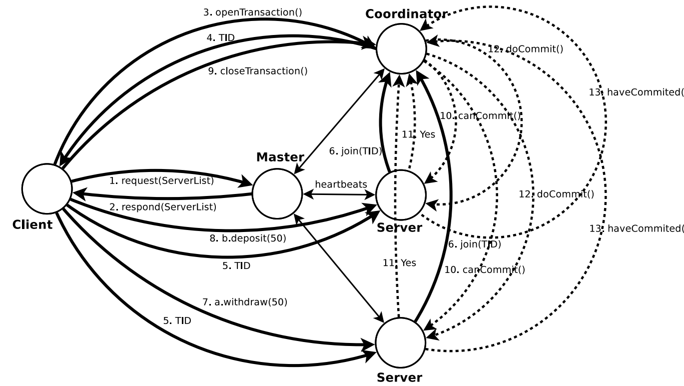

Figure 3: Transaction steps between the Client, Master, Coordinator and Servers

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 [2]
- Faster conflict resolution when compared to locking - transactions are aborted immediately.

Figure 3 illustrates the steps between the Client, Coordinator, Master and Object Serves during a transaction. The 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
- **Step 13:** 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.

3.3. Timestamp Ordering and Deadlock Detection

Since timestamp ordering will be used for concurrency control the chances of deadlock occurring are limited. Each transaction is assigned a unique timestamp upon start. Each operation in a transaction is validated when it is carried out. Should a transaction fail the validation, it is aborted immediately, 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 i.e. if server S1 access an object before 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.

Conflict resolution will be performed at each operation. 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 scenario

In this phase we design to provide fault tolerance for single object server failure. This is an extension for the previous design. The idea is to keep an exact copy of the objects in adjacent server. For instance the object stored in Object server S1, should keep the exact copy in the S2 server 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 message 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 proceeded. (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, stabilizing the servers occurs.
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. Conclusion

5. References

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.