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

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 consistent Servers

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 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 can be any server, not necessarily the Master
-

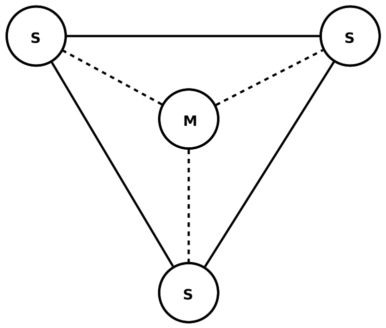

Figure 1. Master/Server entities in a perfect setup

2.1. Perfect links and processes

As illustrated in Figure 1, the architecture consists of the following entities:

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

Note that the number of object servers can be varied.

2.2. Recoverable, replicated Servers with perfect Master

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:

- **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:

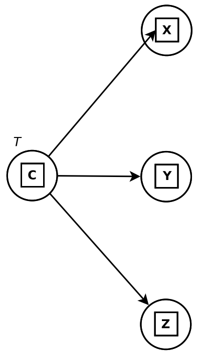

Figure 2. Flat Transaction model between a Client and 3 Servers

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.

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.

The following steps are made when performing a transaction, as illustrated in Figure 3 :

1. Client acquires transaction ID from the Coordinator
2. Client passes TID to the Object Server
3. Object Server calls the Coordinator's*join(TID)*
4. Client manipulates objects directly on the Object Server
5. Upon transaction end, Client asks the Coordinator to either *Abort* or *Commit*

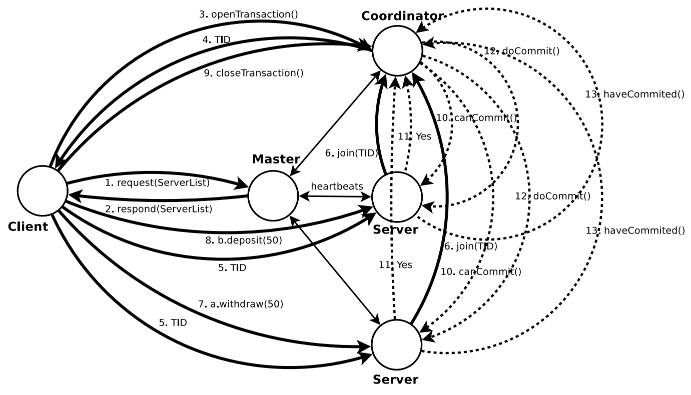

Figure 3. Transaction steps between the Client, Master, Coordinator and Servers

6. Coordinator will request each participant server in the transaction to indicate whether it can commit a transaction or not.*(Voting Phase)*
7. If all participants answer/vote *positively*, the Coordinator issues the **doCommit(TID)** command such that each parcipant commits its part of the transaction
8. 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 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.* [Reference: Distributed Systems]. 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 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

Please use footnotes sparingly[1] and place them at the bottom of the column on the page on which they are referenced. Use Times 8-point type, single-spaced.

[1]Or, better still, try to avoid footnotes altogether. To help your readers, avoid using footnotes altogether and include necessary peripheral observations in the text (within parentheses, if you prefer, as in this sentence).

4.1. References

4.2. Conclusions

References

[1] I. M. Author. Some related article I wrote. *Some Fine Journal*, 99(7):1–100, January 1999.

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