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

2.1. Perfect links and processes

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

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

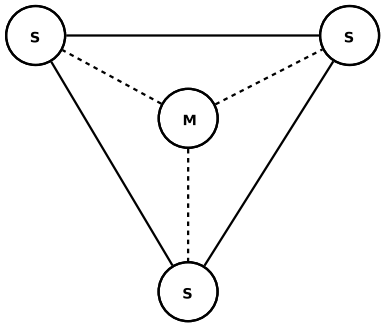

Figure 1. Perfect nodes and links for Master/Server entities

2.2. Recoverable, replicated Servers with perfect Master

2.3. Coordinator/Master responsiblities

The Coordinator

3. Algorithms

3.1. Communication

The following communication occurs **before** the client attempts connection to the Master node.

1. Master starts first followed by the Object Server bootstrap
2. During Object Server bootstrap:
   (a) Master identifies the Object Servers trying to connect
   (b) Master assigns a unique name to the Object Servers for later use as a reference
3. Object Servers's heartbeat messages periodically sent to Master

Upon **attempting a transaction** (object storage) the following communication exists between the Client and Master and Object Servers:

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.

When the Client attempts to **manipulate an existing object** in the *perfect* Object Servers, 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* is being chosen. 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 in the **perfect** environment - assuming no link or process failures:

1. Client acquires transaction ID from the Master

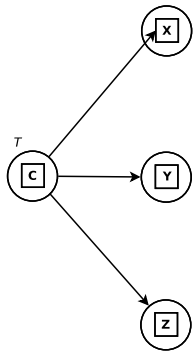

Figure 2. Flat Transaction model between a Client and 3 Servers

2. Client passes TID to the Object Server

3. Object Server calls the Coordinator's(Master's) *join()* method along with TID as parameter

4. Client manipulates objects directly on the Object Server

5. Upon transaction end, Client asks the Coordinator to either *Abort* or *Commit*

6. Coordinator will request a each participant server in the transaction to indicate whther it can commit a transaction or not.

7. If all participants answer *yes*, Coordinator issues **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.

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.