Implementing the *Two Phase Commit* protocol in `Java`

Zak Edwards*

November 24, 2015

Date: November, 2015
Module: `CS241` *Operating Systems & Computer Networks*

Abstract

The *two-phase commit* protocol is a transaction protocol intended to remediate the consistency complications that arise with distributed databases, principally *via* the utilisation of a *coordinator* employed to manage requests on individual databases. The protocol will here be employed to query the stock availability of a number of databases, and perform operations contingent on this availability, *via* the implementation of `Thread`s for concurrent execution.

1 Introduction

The *two-phase commit* protocol is a transaction protocol intended to remediate the consistency complications that arise with distributed databases, principally *via* the utilisation of a *coordinator* employed to manage requests on individual databases. The protocol can thus be separated into two algorithmic phases:

I Voting phase: The client issues a request to the coordinator, in turn querying some multitude of databases (`QUERY`). All databases then reply with either `READY` or `UNABLE`, depending on their capability (*i.e.,* the availability of stock), and the coordinator receives these responses.

II Commit phase*:* If all databases replied `READY`, the coordinator sends `COMMIT` to all databases – else, the coordinator sends `ABORT` to all databases. All databases reply with an `ACK`, and the coordinator receives these responses. The client is then informed of the success or failure of the request.

*`Z.Edwards@warwick.ac.uk`, Department of Computer Science, University of Warwick

Thus, the protocol will here be employed to query the stock availability of a number of databases, and perform operations contingent on said availability.

2 Requirements Analysis

In order to construct an effective and efficient implementation of the protocol that will allow the above detailed operations to be successfully performed, a number of requirements ought be specified.

Firstly, as a necessary minimum requirement, a connection between server and client, over which data can be successfully communicated, must be established.

The `Thread` interface will need to be considered for concurrent execution, in order to ensure scalability of the protocol.

To accomplish the intended objective with regarded to the querying of databases, there must exist, at some appropriate stage of execution, calls to the appropriate functions for identifying and updating a database's stock (`Server.queryDatabase` and `Server.writeDatabase` respectively). There must also exist custom operations to modify the quantity of stock remaining in a database, between client transactions.

3 Implementation

The value `port` is passed as the user-inputted `serverListenPort`, and a new `ServerThread` thread is created:

```
ServerThread server = new ServerThread(port);
new Thread(server).start();
```

The argument `server` is passed to `serverPort`, then:

```
openServerSocket();

private void openServerSocket() {
    try {
        this.serverSocket = new ServerSocket(this.serverPort);
    } catch (IOException e) {...}
}

while(!isStopped()) {

    Socket socket; // Client socket

    try {
        socket = this.serverSocket.accept();
    } catch (IOException e) {...}
```

```
    new Thread(
        new SocketThread(
            socket)
    ).start();
}
```

`socket` is passed to `SocketThread`, where:

```
try {
    if ((socket != null) && (socket.isBound())) {
         socket.close();
    }
} catch (IOException e) {...}
```

Then the appropriate operations are performed based on the success or failure of the subsequent queries.

4 Evaluation

A number of difficulties were encountered in attempting to implement the protocol. One such difficulty was precisely how to establish a coordinator to secure a number of connections. This was solved by cycling through an array of sockets, where the limit of the sockets' indices was defined by the length of the passed address `servers`.

```
public void connectServers(InetSocketAddress[] servers) throws IOException {

    System.out.println("Connecting...");
    Socket[] sockets = new Socket[servers.length];

    for (int i = 0; i < sockets.length; i++) {
         sockets[i] = new Socket();
         sockets[i].connect(servers[i]);
         System.out.println(servers[i] + "connected.");
    }

}
```

Considering how the function `handleClientRequest` was to be implemented resulted in another difficulty. The function was ultimately implemented to write to a database, as per the `writeDatabase()` method, based on the success or failure of a stock request on the respective database, passed from the `acceptServers` method. Thus, the function carries out the requirement of issuing a `READY` or `UNABLE` response to the coordinator.

```
public boolean handleClientRequest(StockList stock) throws IOException {
```

```
    if (success == true) {
        System.out.println("COMMIT");
        writeDatabase(stock);
        System.out.println("Request succeeded.");
        return true;
    } else {
        System.out.println("ABORT");
        System.out.println("Request failed.");
        return false;
    }

}
```

5 Conclusion

The detailed implementation of the two-phase commit protocol, intended to satisfy the requirements detailed in 2, relies on the instantiation of a thread for each client that has connected; whilst this approach may prove effective for a insubstantial quantity of clients, it may prove problematic in efficiently dealing a larger number of clients, largely due to computational inefficiency of creating multiple threads. Thus, in further refinements to the current implementation of the protocol, it may be desirable to circumvent the costly creation of many threads when a greater quantity of clients is necessary.

An intrinsic limitation of the implementation lies in the status of the two-phase commit being a *blocking* protocol – that is, if the coordinator is for any reason halted or disturbed, and thus cannot provide a response to the databases, participants will block resources whilst indefinitely waiting for such a response. The participant may therefore never resolve its transaction.

Similarly, as a further limitation of the implementation, it is conceivable that the coordinator itself should block resources in the event that a participant database requires an excessive period of time to provide a `READY` or `UNABLE` reponse. In the event that no acknowledgement is provided by the participant database, the coordinator can also block resources indefinitely.