CS241 Coursework - Distributed Database System using Two Phase Commit

Jake Fairbrother
u1405424

November 21, 2015

1 Introduction

This report will be covering the design, implementation and improvements of a distributed database system using the Two Phase Commit (2PC) and Three Phase Commit (3PC) protocols. The reasons for using the chosen protocol will be discussed, along with its limitations.

2 Two Phase Commit

The 2PC protocol is an atomic commitment protocol, which allows multiple systems to reach a consensus about a transaction, and only complete the transaction if all agree on the result. The protocol can be used in distributed database systems. The protocol is as follows[5]:

2.1 Protocol

Coordinator

1. The coordinator server receives a request from the client. It then sends the query to the cohort and moves to the query waiting state.
2. If all cohort members return READY, send a COMMIT message to the cohort. If a member returns UNABLE, send an ABORT message to the cohort

3. Wait for all servers to ACK. If committed, return TRUE else return FALSE.

Cohort Member

1. A query message is received. Query the database and if there is enough stock, return READY. If there isn't, return UNABLE.

2. If COMMIT is received, then make the change to the database. If ABORT is received, then do not make the change. Then, return ACK.

2.2 The Problem

The given problem is that a client will send a request to remove stock from a database to a coordinator. The coordinator must check if all members of the database cohort agree that there is enough stock to remove the requested amount. Then if they all report their individual databases have enough, report ready so that the coordinator can commit the change. Having multiple databases in a global service is ideal as this means users can access local servers for queries, reducing latency and minimise the load on a single server[6].

2.3 Issues

One of the main issues with this protocol is that it is blocking[5]. Blocking is when a resource is held by one process, and other processes wishing to use the resource cannot and must wait. 2PC locks the database of a server between query and commit/abort for request R1. So if the server receives another request R2, R2 cannot alter the state of the database so R1 can commit the transaction knowing that the result of the initial query are still correct. This is important for atomicity. However, if the coordinator server were to crash between sending queries and the commit/abort message, R1 would be indefinitely waiting. As it holds the lock on the database, no other process on the server can access the database.

3 Three Phase Commit

In order to to solve the above problem associated with 2PC, the 3PC protocol can be used. The protocol is similar to the 2PC in regards to passing messages between a coordinator server and a cohort of database servers in order to reach a consensus, but differs in number of messages and content. The protocol is as follows[8]:

3.1 Protocol

Coordinator

1. Receive a request from the client and send a query message to all members of the cohort. Move to the query waiting state, and start a timer.

2. In the waiting state, if all members of the cohort return READY, send PRECOMMIT to the cohort and move to the pre-commit waiting state. If not all of the cohort return ready in a set time period, or a member returns UNABLE, send an ABORT message to all members of the cohort.

3. If all members of the cohort return ACK, send out a COMMIT message to the cohort, and return the result to the client. If a member does not ACK within a certain time period, assume it has failed and send an ABORT message to all members.

Cohort Member

1. Receive a request message from the coordinator, and query the database to see if able to perform the query. If yes, return READY, else return UNABLE. Move to the pre-commit waiting state, and lock the database.

2. Receive a PRECOMMIT message from the coordinator, return an ACK then move to the commit waiting state. If the coordinator does not respond in a certain time, assumee it's failed and treat it as an abort. Also, if an ABORT message is received, abort the transaction and release database locks.

3. Receive a COMMIT message from the coordinator, commit the request and release the database locks, then return ACK to the coordinator. If an ABORT message is received, abort the transaction. If the coordinator does not respond in a certain time, assume it has failed and commit the transaction, as it was known that all other members were able to commit. Release the lock.

3.2 Advantages

The advantage of this protocol is that it is non-blocking, unlike 2PC[8]. This can be seen when following possible failure cases of the servers.

- The coordinator fails after querying the cohort: The cohort times out and aborts the transaction, preserving integrity
- The coordinator fails after pre-committing: The cohort knows that all servers are ready, so they can commit the transaction
- A cohort member fails before pre-committing: The coordinator aborts as not all members respond to the pre-commit message.
- A cohort member fails before committing: The coordinator aborts as not all members ACK the pre-commit message
- A cohort member fails after commit message sent out: Does not matter as all servers agreed that there was enough resource.

From this it can be seen that in all failure cases, all locks will eventually be released. Also, In the event that the coordinator fails, the cohort may be able to continue with the transaction.

An obvious problem with the 3PC is that it requires more messages to be sent between servers. This will increase latency in the protocol and result in slower transactions than 2PC[8]. Also, the 3PC is not secure against multiple server failures, but this is less likely than 1 failure[7]. However, these issues can be considered a limitation of hardware, and having more physical resources in the forms of CPU and bandwidth capacity will reduce this latency.

Therefore it can be concluded that a 3PC is more desirable than 2PC, as the system is more reliable in the event of crashes.

4 Implementation

A client connects to a ServerSocket, so this will define if the coordinator treats the cohort as clients or servers. By treating as them clients, and creating a ServerSocket on the coordinator, the issue of assigning ports is easily handled. For each connecting member of the cohort, the ServerSocket will give the connection a free Socket[9]. This solution requires connections to be attempted multiple times, as two cohort members may try to connect concurrently. Thus one will have to retry when the ServerSocket is free. This approach also allows an arbitrary number of servers to be created. In the other model, each cohort member needs to be given a free port manually - this is difficult with many servers.

Each Socket can only have one input and one output stream[9]. The Socket along with an ObjectInputStream and ObjectOutputStream will be encapsulated in ThreadedCoordinatorWorker, which extends Callable.

Communication between the coordinator and the cohort will be done in threads. This is because if a cohort member is slow, a non-threaded coordinator will freeze while waiting for a response. A thread avoids this. A ThreadPool will be used to run threads, so that the system is not overloaded in the case of a large cohort. When communication is needed, the ThreadedCoordinatorWorker is run in the ThreadPool. To get the return values, the ThreadPool is wrapped in a CompletionService[1]. The ThreadedCoordinatorWorker is reusable, having a method to set the output message. This makes it efficient as thread objects can be reused.

Throughout the protocol, the coordinator creates messages, passes them to the ThreadedCoordinatorWorker and submits to the ThreadPool. It reads the results via the CompletionService and acts according to the protocol. The cohort members create a socket, streams in the ConnectServers method, and have a case statement for handling the different messages during the protocol.

5 Difficulties in the Implementation

I had difficulties in communicating over sockets while multithreading. At first, the implementation created new Input and Output streams for each socket on each run of the thread. This however did not work. Upon reading stack overflow[4], it was found that only 1 pair of input/output streams can be created per socket, confirmed in the JavaDoc[10]. Closing the stream releases any system resources, which meant that no further streams could be created. I solved this issue by encapsulating the input and output streams inside a thread object, instead of passing the streams to the thread each time. Reusing the threads is also more efficient, as creating threads is an expensive process in terms of memory[2].

6 Possible Improvements

The improvement of using log files on the coordinator and cohort servers would be implemented if recreating a solution[7]. These log files would record every action of the server; such as open connections and their port numbers, the state of the protocol and the query send by the client. This can then be used by backup servers in the event of a failure to restore to a recent state. This would mean that the system would be more reliable. Instead of returning FALSE in the event of a failure, after a slight delay, the actual result of the query would be returned.

7 Limitations of the Implementation

A limitation of the system is recovery. If the coordinator fails during the transaction, the cohort will abort. However ideally the coordinator should log actions and, on failure, a secondary coordinator should be able to continue where the first left off[7]. This will save resources and time, as transactions will not have to be repeated. Another limitation is what requests the system can take. At the moment, the sent message only contains a StockList, not a command of what to do with it[3]. This means that the system assumes it needs to remove the list, and cannot perform actions such as "add stock".

References

[1] Bhaskar. "when should i use a completionservice over an executorservice?". "http://stackoverflow.com/questions/4912228/when-should-i-use-a-completionservice-over-an-executorservice".

[2] S. C. Why is creating a thread said to be expensive? "http://stackoverflow.com/questions/5483047/why-is-creating-a-thread-said-to-be-expensive".

[3] DCS. Javadoc - cs241. "https://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs241/
16/javadoc.zip".

[4] fortran. System.out closed? can i reopen it? "http://stackoverflow.com/questions/8941298/system-out-closed-can-i-reopen-it".

[5] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Data- base Systems*, pages 5–7, 9 2005.

[6] J. Gray and S. Metz. Solving the problems of distributed databses. *Tandem Computer Inc*, 10 1982.

[7] S. K. S. Kushwaha. Extended two phase commit protocol in real time distributed database system. *International Journal Of Engineering And Computer Science*, 4:12188–12193, 5 2015.

[8] P. Lannhult and B. Lindblom. Review of non-blocking two-phase commit protocols. 11 2009.

[9] Oracle. Javadoc. "http://docs.oracle.com/javase/7/docs/api/"".

[10] Oracle. Outputstream documentation. "http://docs.oracle.com/javase/7/docs/api/java/io/OutputStream.htmlclose()".