DDBS: A Distributed Database System

Ashley Lee (k7y8), Rock Luo (k0a9), and Haoran Yu (c4w9a)

Abstract

In this project, we worked on prototyping a distributed database system (DDBS). The system provides services to the applications accessing the database with high availability and strong consistency. Using the API we designed, we implemented well-known protocols such as the strict two-phase locking protocol, two-phase commit protocol, and the heartbeat protocol in order to achieve availability and consistency. However, our system topology dictates the implementations of the protocols, therefore, we modified the protocols to accommodate our topology. Our database system performs transactions with ACI semantics, and we simulated SQL queries using the database operations we defined in our system. We have tested our system with thorough test cases with scenarios where the different nodes crash at different time points.

Problem Statement

The world depends on information systems. Information is stored in databases and accessed by clients for different purposes. In traditional systems, all information inside a database is centralized in a single site, however this poses the following issues [1]:

1. The system cannot achieve **high availability**. In the event of failure, none of the clients can access the database.
2. The communication cost of serving multiple connected clients is high.

The solution proposed by [1] is to distribute parts of the database across many smaller sites, a system called a **Distributed Database System** (**DDBS**). All data is replicated, so if site X goes down, then the data stored there can still be accessed by going to a site(s) Y that has replicated all data on site X. The communication cost is reduced because each site now serves a smaller number of clients. However, a DDBS faces additional problems:

1. Database **transactions** performed on the data must follow a strong **consistency** model, because a traditional transaction has **ACID** (**atomicity, database consistency, independence, and durability**) semantics. Atomicity and independence require strong consistency: the transactions are performed in the order that they are received by the DDBS, and a client that reads and writes to a database cannot be influenced by another transaction.
2. DDBS must achieve **location transparency** with regards to a client, as if the client is interacting with a single database.

Design

In this project, we worked on prototyping a version of DDBS that attempts to solve the problems discussed above. The DDBS will consist of multiple **server nodes** that store the different parts of the database. Our database will consist of a collection of tables only. Different tables are stored in different server nodes, and each table is replicated in 2 or more server nodes. The DDBS also contains multiple **client nodes** that form transaction requests and interact with the server nodes to execute the transaction. The clients do not know where each table is stored, therefore the DDBS also has a single lookup service:

a **Load Balancer Service node** (**LBS**). The LBS knows what tables are stored at each server node, and is contacted by a client node every time a transaction is created in the client node.

Within one transaction, the **operations** supported by our DDBS are **READ**, **WRITE**, and **DELETE**. For example, a client can read rows from a table as well as insert and delete rows of a table. Most importantly, any tables that are interrelated can be queried together within a transaction by **JOIN**ing the tables. For example, given a Employee table and a Company table, we join the two tables in order to find out which employee works in which company.

System Topology

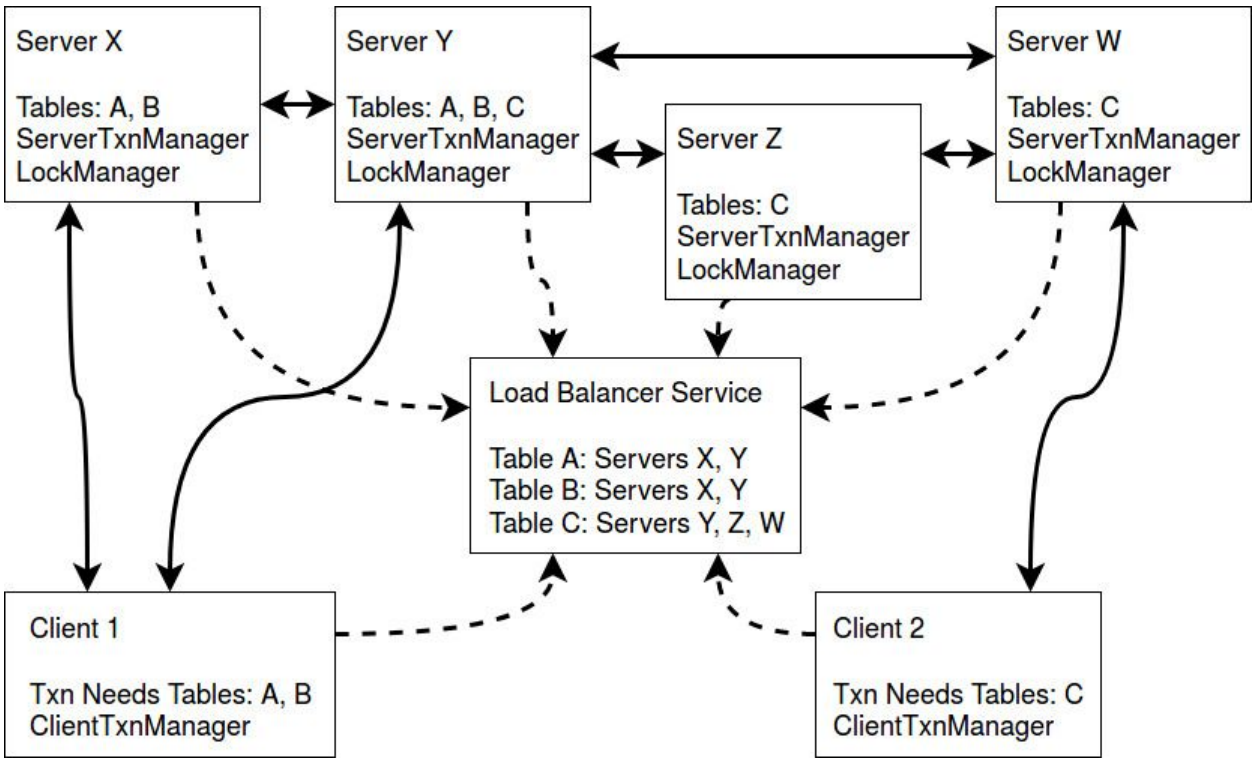

Figure 1: An instance of DDBS
In this particular instance, there are 4 server nodes, 2 client nodes, and a Load Balancer Service. The solid lines indicate a bidirectional connection between the nodes where heartbeats are sent periodically. The dashed lines indicate a connection without heartbeats. In this instance, there are 3 tables: A, B, C, replicated across different server nodes. The Load Balancer Service stores which server has which tables. Client nodes are required to hold locks before initiating transactions with servers. A lock manager is located in each server handling locking of its tables. The transaction manager is split between the client and its connected server. Note that in this instance, Client 1 is connected to Server X and Server Y to perform transaction with tables A and B. In this scenario, Server X might be servicing operations on table A and Server Y might be servicing operations on table B, however it is possible for Client 1 to only connect to one server to perform the transaction because both servers have A and B.

There are an arbitrary number of server nodes, a server X has birectional connection with a server Y if X and Y have at least one common table A. Each server knows about the IP address of an LBS node, the server connects to the LBS to inform what tables are stored on the server. A client establishes a

bidirectional connection with a server if the client is performing a transaction on the server's table(s), the client disconnects from the server when the transaction is completed or aborted. The clients do not talk to each other.

CAP Theorem

The CAP theorem states that distributed systems cannot achieve high consistency, availability, and network partition tolerance at the same time. This is particularly true for DDBS, where we attempt to enforce strong consistency across all nodes. We have attempted to achieve **read-your-write consistency** as well as **monotonic reads consistency** in our system. Consistency is explained in more detail in ACI Semantics section. We do not tolerate **network partition** - a partitioned node X storing replicated data of another node Y in the network cannot guarantee consistency if write operations are performed on X. DDBS attempts to provide high **availability** to multiple clients concurrently. DDBS guarantees high availability because in the event of failure, where any node in a DDBS disconnects, the rest of the system is not affected significantly and is able to continue to execute a transaction.

1. Although the LBS is a single point of failure, once the client knows about the IP address of the server to perform a transaction, the client does not need to talk to the LBS during the transaction.
2. We have implemented thorough server crash detection and recovery mechanisms, as well as client crash detection mechanisms. Once a server X crashes during a transaction, the client talks to the LBS to connect to a second server(s) Y that has the same tables as X. These will be explained in the Implementation section.

ACI Semantics

Due to the limited scope of the project, we decided to implement transactions with **ACI semantics** without durability. With ACI semantics, the contents of all copies of a table must be identical at the end of a transaction when the server nodes are servicing requests from client nodes.

To achieve **independence** between the transactions, we use locks to enforce mutual exclusion, and we have implemented the **lock manager**. For each table that a transaction needs, a lock must be acquired for that table first. The locking states of tables are communicated between the servers; clients that try to lock the same table (on different servers) for a transaction must have only one client that succeeds. Table locking follows the **strict two-phase locking** protocol.

To achieve **atomicity** and **database consistency**, we implemented **distributed transaction managers**: due to the topology of our DDBS system, both the client node and the server node needs a transaction manager. We need a **Client Transaction Manager** because a client may connect to multiple servers during a single transaction, therefore it needs to keep track of which operation is handled by which server, and in our system the client is also required to perform the JOIN operation locally (it does not talk to the servers to perform JOIN). A server needs a **Server Transaction Manager** to check for integrity constraints on the received operation before performing the operation to its table (such as inserting a row to the table), it also communicate with its neighbours (which also have a copy of the table) to propagate changes made during the transaction. The Server Transaction Manager and the Client Transaction Manager together implement the **two-phase commit protocol** (**2PC**). If a client is connected to 2 or more servers during a transaction (transaction needs tables A and B, and they are on 2 different servers), each Server Transaction Manager must ensure that all copies of the table have stored the new contents, and the Client Transaction Manager then checks whether all of the Server Transaction Manager returned success before completing the transaction.

Note that any server that holds the lock for the table becomes the Transaction Manager of that table, thus there is no fixed primary-backup relationships between the servers, that is, there is no single **primary server** that manages all the tables. For example, in Figure 1, Client 2 is performing a transaction on table C, and it is connected to Server W. Then Server W is the primary server for the transaction, and relative to W, Servers Y and Z are the backup/**secondary servers**.

Assumptions and Limitations

Our model assumes that all nodes (the servers, the clients, and the LBS) can crash, and we have partially implemented crash detection and recovery mechanisms for each type of node. The crash detection is done using the **heartbeat protocol**. The mechanisms are discussed in more details in the Implementation section. In addition, we have the following assumptions:

1. We do not have a threat model since we assume that all nodes in DDBS are trusted.
2. To prevent complexity in our system, we do not allow dynamic table creation and table deletions. We provide the table names as part of the command line arguments when we run the server code. We predetermine the location of the tables arbitrarily prior to running each server.
3. We do not support dynamic table replication. We assume that at least one copy of each table is available at all times (that is, server crashes will not cause a table to be unavailable globally). In addition, we assume that all tables are replicated in at least 2 nodes in the DDBS.
4. LBS does not perform optimizations. When the client requests a server to connect to, the LBS will randomly return a server IP address that has the requested table. The LBS does not attempt to minimize load of any single server because it has no knowledge of how many transaction requests each server is currently serving.
5. Causal consistency cannot be achieved due to transaction semantics. Therefore between the client and the server, only read-your-writes consistency is achieved. However, causal consistency can be achieved between the servers.
6. Network partition refers to a loss of connection between two or more nodes, however such a communication failure cannot be distinguished from a node crash, therefore our crash detection mechanism may be incorrect in the event of a partition.
7. We did not test the performance of our system, but we expect our system to be slower than other systems that do not use pessimistic concurrency control, but it still outperforms a centralized database system because our DDBS supports some degree of parallelism.

Implementation

We will discuss the protocols that we have implemented that comply to the design of our system.

Load Balancer Service Architecture

The Load Balancer Service stores table name to server IP mappings, and this map is actively updated as servers join and leave the system. The LBS does not interact with any nodes, it only acts as a centralized storage. When a client starts a transaction on a list of tables, it calls **LBS.GetServers()**, which returns a corresponding IP address for each table. When a server joins the system, it first registers the tables it has with the LBS by calling **LBS.AddMappings()**. The newly joined server then calls **LBS.GetPeers()** to request all the servers in the system that shares at least one table in common with this server, this server then connects to each of the returned servers. Additionally, when a server crashes, a connected peer

detects the crash and can call **LBS.RemoveMappings()**. Once this step is performed, a client will no longer receive the IP address of the crashed server by calling GetServers().

Load Balancer Service Crash and Recovery

To allow the LBS to survive a crash, the load balancer stores all registered server IP addresses on disk. A write occurs every time a server registers itself with the LBS. So when a crash occurs and its table mappings are lost, upon recovery, the load balance will retrieve all registered IP addresses from disk, and contact each address by calling **Server.GetTablesNames()** to retrieve its tables. Using this information, the load balancer will rebuild the table-to-IP address mappings table. The load balancer will not service any requests until recovery has finished.

Client Architecture

The client is responsible to create and execute transactions. Upon startup, an application will call **Client.StartClient()** which allows the client to connect to the LBS. The app will call **Client.NewTransaction()** to execute the transaction and commit it to the servers. When transaction has been successfully committed, Client.NewTransaction() will return true, else it will return false. The client stores the following: a load balancer RPC connection and a list of connected server RPC connections.

Table and Transaction Structure

Each table has fields Name and Rows. Rows is a map of key-value pairs, where the key is the primary key of a row and the value is a map of all attribute name to the actual data element.

Each transaction stores a list of operations. The following five types of operations (and the corresponding **server table API** that handles the operation) are supported:

1. SelectAll: this is a READ. Returns all contents in the table - **GetTableContents()**
2. Select: this is a READ. Returns the row corresponding to the given key - **GetRow()**
3. Set: this is a WRITE. Set key-value pair to the table, insert if key does not exist in the table - **SetRow()**
4. Delete: this is a WRITE. Delete key-row pair - **DeleteRow()**
5. Join: Join two tables based on the given attribute keys

We do not have a Server API for joins, because we can perform the join operation within the client. For example, within a transaction, we need to join two tables and perform update to the joined table. We first retrieve the entire two tables from the servers by calling GetTableContents(). Then for each table row in one table, use the foreign key to find the corresponding row in the other table. Then we use the joined table to perform updates. Since a row in the joined table is made up of two rows, the client calls SetRow() to the server that has that table row (for example, call SetRow(aRow) to server A and SetRow(bRow) to server B).

Below are high-level structure of the operations:

```
Operation{Type: “SelectAll”, TableName: name}
Operation{Type: “Select”, TableName: name, Key: key}
Operation{Type: “Set”, TableName: name, Key: key, Value: row}
Operation{Type: “Delete”, TableName: name, Key: key}
```

```
Operation{Type: "Join", TableName: name1, SecondTableName: name2,
FirstTableColumn: key1, SecondTableColumn: key2}
```

Distributed Locks

The distributed states in DDBS are the table locks. The client issues a **TableLock()** to the primary server containing the table, the primary server must then lock the table globally. LockTable() does not return until all servers containing the table are locked. A **TableUnavailable()** is issued to each of the secondary servers containing the table, they lock their own table and return. Once all copies of the table are locked, the primary server returns success to the client. The primary server servicing the client's request is the table lock owner, and it is responsible for unlocking all copies of the table when the transaction is about to complete. The clients issues a **TableUnlock()** once the transaction commits, the primary server receives the request and issues **TableAvailable()** to the secondary servers, they unlock their table locally and it is immediately available to service any other client's requests using the unlocked table. However, if the previous transaction is not finished with unlocking all copies of the table, then the new transaction cannot proceed until all copies of the table are unlocked. Once all secondary servers unlock the table, the primary server returns success to the client to complete the transaction. Each server containing a table knows who is the lock owner, this information is helpful when the lock owner crashes and the crash is detected by a secondary server.

How Client Performs Transactions

The client transaction manager implements part of 2PC. Once the application calls NewTransaction(), the client computes the tables it needs to lock, calls the LBS to retrieve a list of primary server addresses that contain these tables, and connects to the primary servers. Once it connects to all the required servers, it will lock the tables. Then for each operation, the client will call the corresponding table API on the server side. After all operations are complete, the client will now send out a **Server.PrepareCommit()** message to all the primary servers. Each primary server will respond with a PrepareCommit ACK message when it and its secondary servers are ready to commit the transaction. Once the client receives all ACK messages, it will send out a **Server.CommitTransaction()** message to all the primary servers. Each primary server will respond with a CommitTransaction ACK message when it and its secondary servers have committed the transaction. Once the client has collected all CommitTransacton ACK messages, it will return success.

If a primary server crashes during any point of the process, the client will detect the crash using heartbeat protocol, and close all primary server connections. It will then refetch a list of primary server addresses from the load balancer and retry the transaction again. The client will keep retrying until the LBS respond with an empty list, then the client will abort the transaction and returns fail to the application.

Server Architecture

Each server connect to its peers, and handle transactions from clients. Upon system startup a server will register its IP address and tables with the LBS, then retrieve a map of table names-to-peers. A server only retrieves mappings for tables it owns. Next, the server will connect to each of its peers using **ConnectToPeer()**, and send heartbeats. The server then retrieves the contents of each table from peers by calling GetTableContents(), and copies the contents to its own tables. Finally, it begins listening for connections from both servers and clients.

A server stores the following: its local tables, a map of the table locks and their global state (whether they're currently locked or not), and the lock owner. It also stores a list of all connected peers and a list of connected clients.

How Server Handles Client Requests

Each server has tables as well as their backups. For example if a server has table A and table B, it will also store A_BACKUP and B_BACKUP. The architecture and the protocols are already discussed in the ACI Semantics section. When a new transaction is initiated and a table is locked on a primary server, the primary server copies its table contents to its BACKUP table, the secondary servers also backup the contents of the table by storing them in the BACKUP table. When the Client Transaction Manager issues a request (such as SetRow), the primary server receives the request and the Server Transaction Manager checks whether the request is valid, and locally updates its table (while leaving the BACKUP table unchanged). When the Client Transaction Manager calls PrepareCommit(), the primary server then talks to each of its neighbours/secondary servers by calling **PrepareTableForCommit()** with a copy of the updated table. When the secondary servers receive the message, they update their table. If a secondary server crashes at this stage, the primary server ignores it and return success to the client anyway, this is because the crashed server does not affect the consistency. When the client receives success, it then issues CommitTransaction to the primary server. The primary server then calls **CommitTable()** to each of the secondary servers, and the secondary servers respond success.

Server Crash Handling

Primary Server

When a primary server crashes, its locks must be released, its mappings removed from the load balancer, and any lost transactions handled. When a connection is established between servers, heartbeats are sent out periodically from one server and the other server constantly monitors the heartbeat. If a heartbeat is not received after 2 seconds, then it assumes that a crash has occurred. Whichever peer detects the crash first will release the locks, and handle lost transactions.

For lock handling, the crash detector unlocks any locked tables. Once finished, it disseminates lock availability to the rest of its peers by calling TableAvailable. Whenever a server is handling a transaction and crashes, all peers perform a rollback on the transaction tables. First, the crash detector rolls back its tables locally, then tells the remaining peers to do so by calling **RollbackPeer()**. If a server crashes while not handling a transaction, then the peer only needs to release its locks, and remove its mappings from the load balancer.

Finally, a peer must remove the crashed server's mappings from the LBS. There is no way for each peer to know whether another has already removed the mappings, so every peer calls RemoveMappings. This is alright, as the server mappings are stored within a map, and delete does nothing if an entry is nonexistent. Thus, we can call RemoveMappings multiple times without adverse effect.

There are four time points during a transaction where the primary server could crash. These are explained in detail in Evaluations.

Secondary Server

Relative to the primary server, a secondary server is not performing a transaction. When a secondary server crashes, its mappings must be removed from the LBS. As in the case of the primary server, whichever peer first detects the crash calls RemoveMappings on the load balancer.

Server Recovery

After a server has crashed, then there is the possibility it may rejoin the network. Upon doing so, the server must register again with the LBS, retrieve its peers, connect to them, and then retrieve the table contents from its peers and copied the table contents to its local tables.

How Client Crash Affects the System

During a transaction, the client could crash. In this case, the primary server(s) that handle the client's transaction will detect the crash via the heartbeat protocol. The primary server then rolls back its table contents using its BACKUP table and it tells the secondary servers to roll back their tables as well (if new changes were already made in their table), it removes the client's IP from its list of connected clients map, and unlocks the table locked by the client and talks to the secondary servers to unlock their tables as well.

There are four time points during a transaction where the client could crash, explained in Evaluation.

How Deadlocks are Prevented

One issue that cannot be solved using strict two-phase locking is that deadlocks can still occur. If two clients lock for tables A and B but one holds the lock for table A, while the other holds lock for B, then a deadlock has occured. To prevent such scenario from happening, we force each client to lock tables by the alphabetical order of the table names. However, we note that in the real world, such an ordering might not be easy to determine.

Evaluation

We have implemented all of the above except for the test cases for the demo. We have evaluated our system by running multiple servers nodes, client nodes, and a LBS node on Azure.

Shiviz Sample Diagrams:

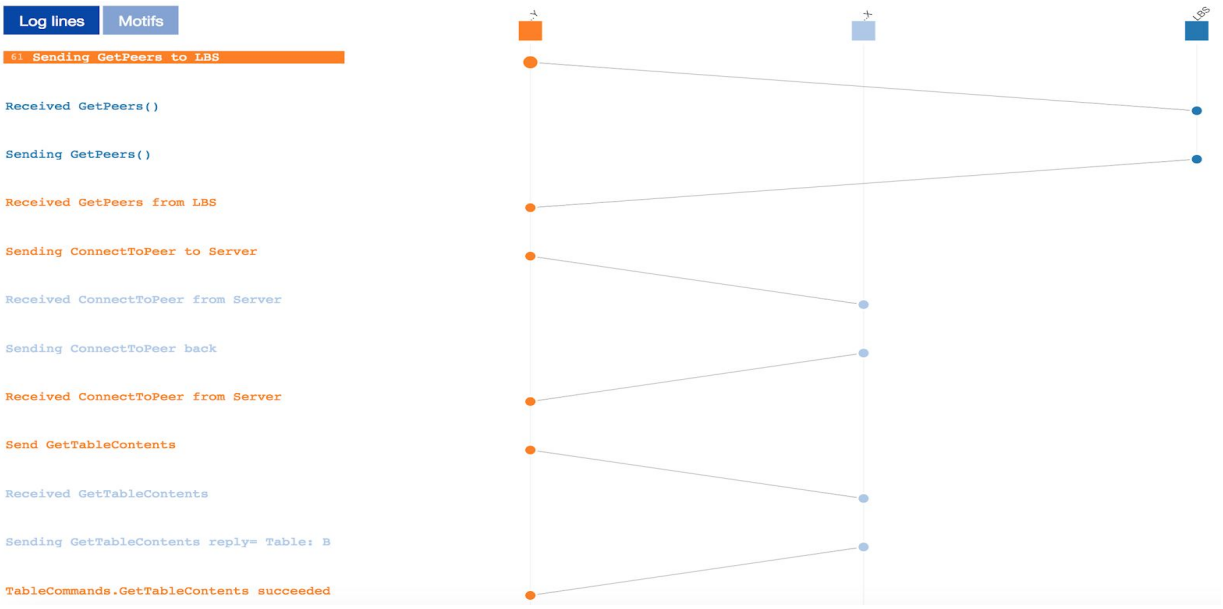

Figure 2: After server gets its peers, it connects to them and retrieves table contents

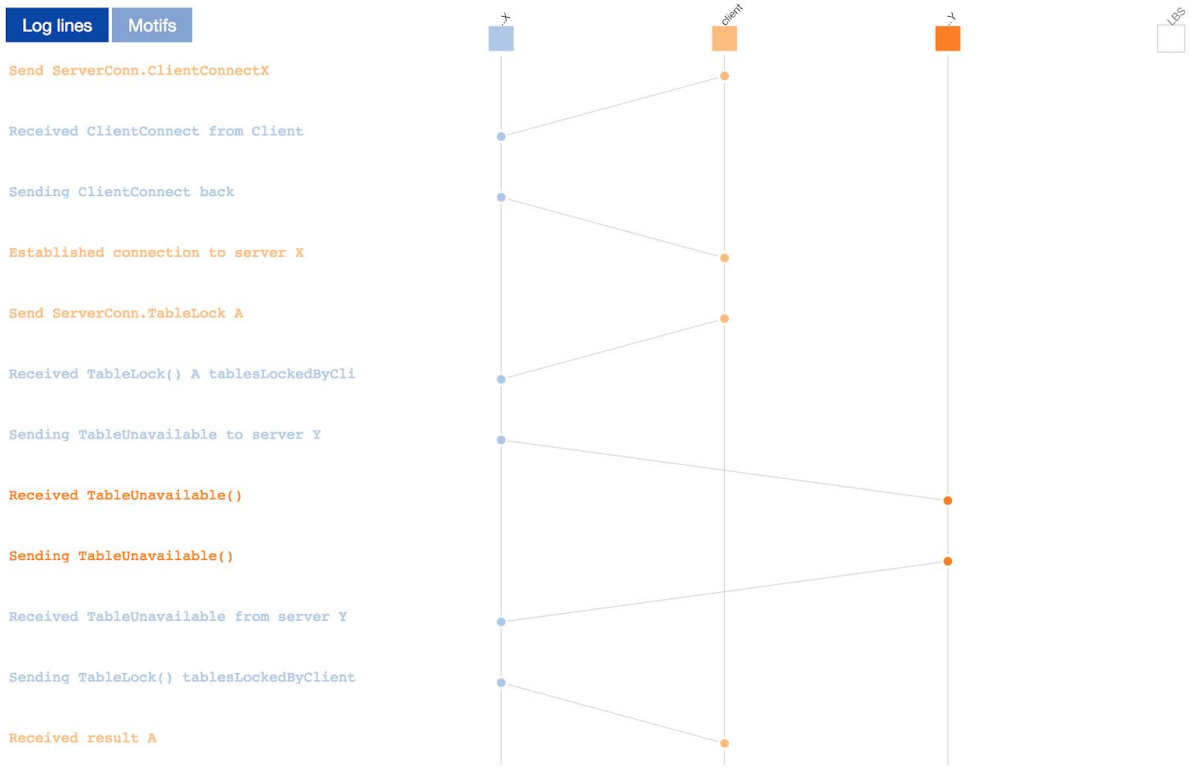

Figure 3: An example of a client starting transaction. The client connects to a server, and locks table A

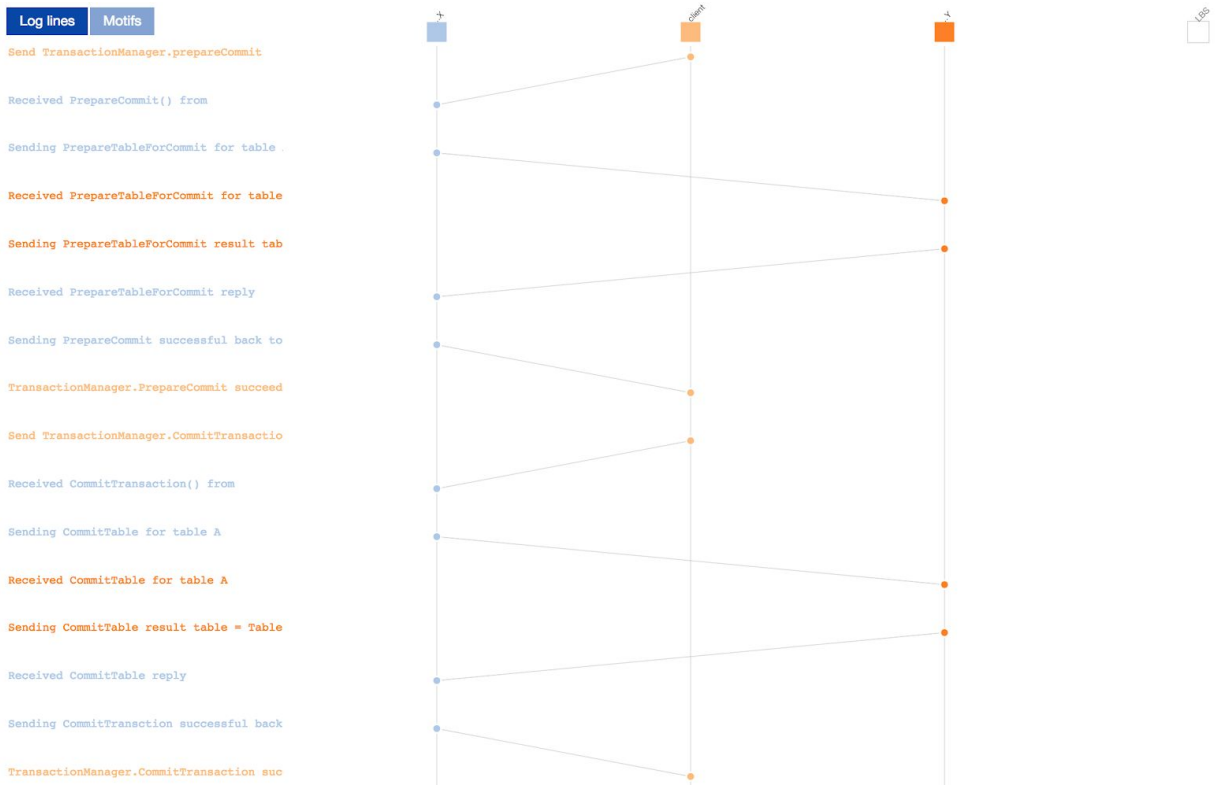

Figure 4: An example of the 2-phase commit protocol. After committed, the client will unlock table A:

Additional Complete Shiviz Diagrams

`govectorLog-Join.txt` - Transactions succeed. No server/client crashes.

`govectorLog_crashedDuringTxn.txt` - Primary server crashes during a transaction.
`govectorLog_crashedServerCommit.txt` - Primary server crashes after client has sent commit.
`govectorLog_crashedServerPrepare.txt` - Primary server crashes after client has told it to prepare to commit.
The fourth case is when commit has succeeded.

`govectorLog_crashedClient_case2.txt` - Client crashes during a transaction.
`govectorLog_crashedClient_case1.txt` - Client crashes after client sends out PrepareCommit.
`govectorLog_crashedClient_case3.txt` - Client crashes after client sends CommitTransaction.
`govectorLog_crashedClient_case4.txt` - Client crashes after client receives commit ACK from all of the primary servers. Servers persist the commit.
`govectorLog_lbsCrash.txt` - The LBS crashes and recovers from the crash
`govectorLog_deadlock.txt` - A deadlock scenario that will not occur in our system

References

[1] Hakimzadeh, H. "Distributed databases fundamentals and research." *Haroun Rababaah Advanced Database–B561* (2005).