DDBS: A Distributed Database System

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

Introduction

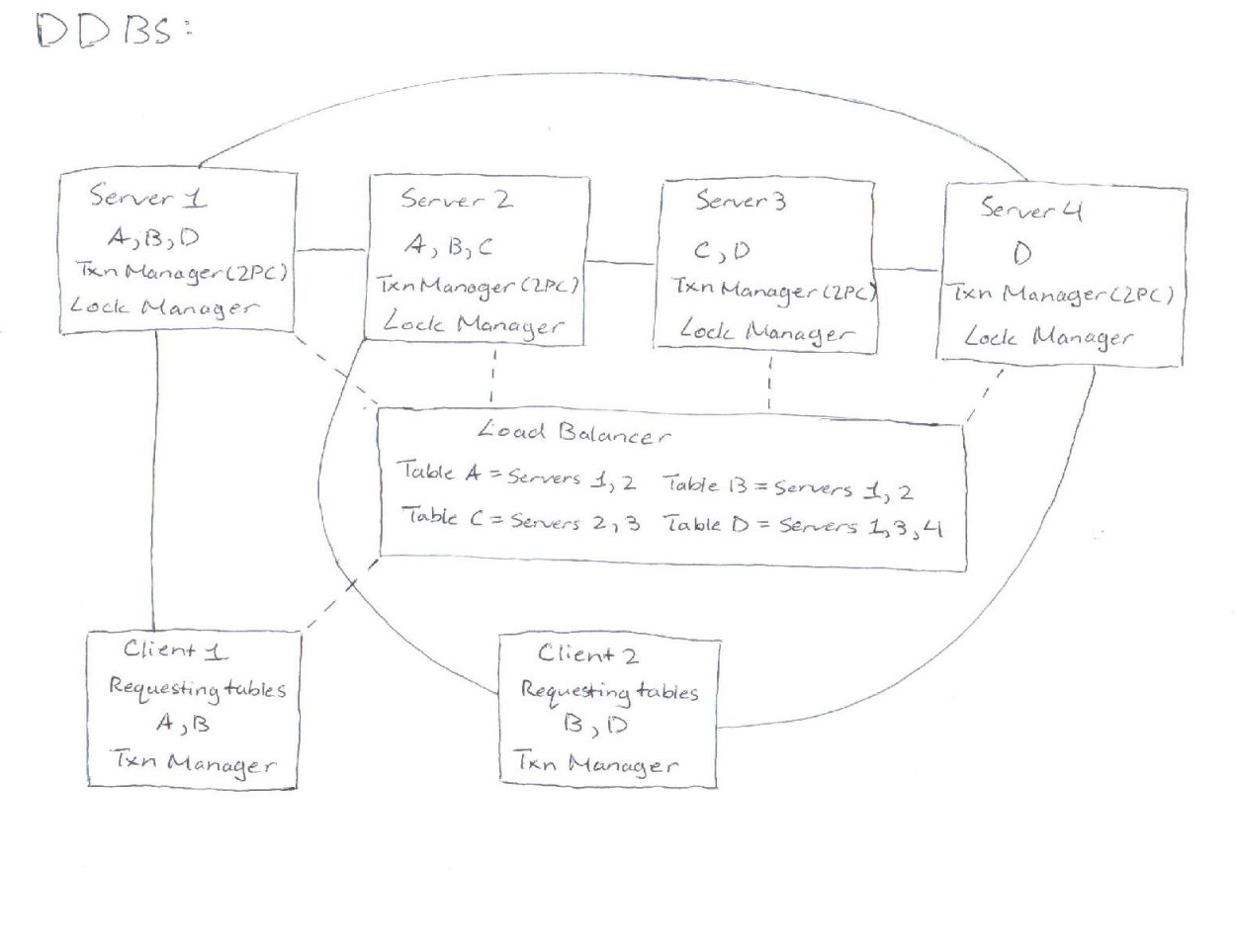
In this project, we would like to work on prototyping a *distributed database system* (DDBS). The system provides reliable service to the clients using the database, and should do so without high performance cost. The system will use replication to manage data loss and work distribution. We also attempt to achieve location transparency with regards to the client, as if the client is interacting with a single database. In reality, the database is distributed across different nodes in the network. At the same time, we must enforce consistency on a global scale. The unit of work performed on the DDBS is a *transaction* with ACI semantics. That is, the transactions are performed in the order that they are received by the DDBS, and a client that reads and writes to a database within one transaction must have an idempotent outcome.

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 consistency is enforced across all nodes, but we do not tolerate network partition - a disconnected node storing interrelated data with respect to another node in the network cannot guarantee consistency. However, DDBS attempts to provide high availability to multiple clients concurrently. A centralized Database System cannot achieve high availability, because in the event of failure, none of the clients can access the database. On the other hand, if any node in a DDBS disconnects, the rest of the system should not be affected significantly and is able to continue to provide services to clients [1].

Overview

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, any tables that are interrelated can be queried together within a transaction by joining the tables. Different tables are stored in different Server nodes, and all tables are replicated in at least 2 nodes. The DDBS also contains multiple Client nodes that receive transaction requests from user application and interact with the Server nodes. The clients do not know where each table is stored, therefore the DDBS also has a single Load Balancer Service node (LBS). The LBS knows what tables are stored at each Server node, and is contacted by a Client every time a transaction is created in the Client.

The Topology of DDBS



The figure above illustrates that the distributed database contains 4 tables: A, B, C and D. Tables are replicated. The Load Balancer Service (LBS) stores which server has which tables. Clients are required to hold locks before initiating transactions with servers. Solid lines indicate bidirectional permanent connection (constantly sending heartbeats), and dashed lines indicate temporary connections (no heartbeats).

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. The contents of the two tables must be identical at all times when the servers are servicing requests from clients. Each server knows about the IP address of a 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.

Our model assumes that all nodes (the servers, the clients, and the LBS) can crash, and there are crash recovery mechanisms for each type of node. In addition, we do not have a threat

model since we assume that all nodes in the DDBS are trusted. Aside from the networking components that we discuss below, our application design is minimal.

ACI Semantics

To adhere to the ACI semantics, we use locks to enforce mutual exclusion. The locking states of tables are communicated between the servers, clients that try to lock the same table for a transaction must have only one client that succeeds. For each table that a transaction needs, a lock must be acquired for that table. Once the lock is acquired for a specific table, the server that the client is connected to (containing the table) must ensure that all the replicated table copies (stored in other servers) are made unavailable. That is, a table can be used by only one client globally.

To avoid deadlocks between the clients, we will use timeouts. If a client needs the lock for tables A and B but only holds the lock for table A, the lock for A is released after an extended period of time without acquiring lock for B. Table locking follows the *strict two-phase locking protocol*. The client cannot initiate the transaction until it has all the locks it needs, thus ensuring *independence* of transactions. If the client crashes before acquiring all the locks, the server can act in place of the client to release its locks so that other clients can proceed acquiring the lock.

We implement Locking Manager at each server node. That is, each server knows about which tables are currently locked, and which of its peers has the lock. However, the server only knows about the locking state of a table it stores, it does not know any states of tables that it does not have (if the server does not have table C, then it does not know if C is locked or not).

Both the client and the server needs a management system locally to manage transactions. For example, a client needs to keep track of each operation within a transaction, and needs to read data from the server and process the operations locally (such as joining the tables). The server needs to check for integrity constraints on the received data before adding it to its table (inserting a row to the table). Therefore we implement a Transaction Manager in both the Server node and the Client node. The roles of each Transaction Manager is described below:

Server Transaction Manager - Validates each operation, perform updates to the tables, and ensure that all replicated tables observe the changes. Thus, *consistency* is maintained across all copies of the table. Any server that holds the lock for the table becomes the Transaction Manager, thus there is no single fixed Transaction Manager that manages all tables.

Client Transaction Manager - Parses the transaction received from the user application, translates each operation into Server semantics and requests the Server to perform the operations. The Transaction Manager ensures that all operations have been successfully completed on the connected server before returning to the user application, thus ensuring *atomicity*.

The Server Transaction Manager and the Client Transaction Manager together implement the *two-phase commit protocol*. Two-phase commit protocol between the client and the server allows a transaction to succeed or fail atomically. When a client initiates a transaction, the contacted server must communicate with its neighbours (which also have a copy of the table) to propagate changes made during the transaction. The server talks to every other connected server to provide them with the updated table data, and the connected servers reply to inform them of success or failure. An operation cannot succeed until all servers have stored the new contents. The Client Transaction Manager then checks whether all of the operations succeeded before returning success or initiating abort.

Structure of a Transaction

Within a transaction, the client will be able to perform read, write, and delete operations, such as reading rows from tables, inserting and deleting rows of a table. A transaction that the client creates is a struct that stores all necessary fields for each operation in the transaction.

We support the following 4 types of operations:

1. SelectAll: Returns all content in the table
2. Select: Returns the row corresponding to the given key
3. Set: Set key to row, insert if key does not exist in the table
4. Delete: Delete key-row pair

Below are a few examples of 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}
```

The structure of a table in the database is a map data structure, where the key is the primary key of a row and the value is a collection of all fields in the row.

Failure Detection and Crash Recovery

Server

The connections between servers allow failure detection. When a server crashes, its neighbours detect the crash and inform the LBS. The LBS removes the crashed server's IPs from its table mappings. In order for the server to recover, each server stores a list of tables it has on disk. However, it does not store the contents of the tables, only the table names are stored. When the server restarts, it will read its tables from disk, and contact the LBS to receive the list of servers which it has at least one table in common with. The server proceeds to connect to them. The

crashed server will receive full copies of the tables from its connected neighbours. The restarted server then informs the LBS to register its tables in the mapping.

When the server crashes during a transaction, the client detects the crash and aborts. The server's peers detects the crash and rolls back all the changes made during the transaction.

Client
We assume that a single user application is interacting with a client, and the application is coupled with the client (the application is permanently associated with the client). When a client crashes, its connected servers detect the crash and release locks the client has. If a transaction has not been completed and committed, the Server Transaction Manager rolls back all the changes on all copies of the table.

LBS
The LBS maintains a map of database tables to the server IP addresses. At the same time, it stores a list of the server's IP on disk. When a new server joins, the server's IP address is stored on LBS's disk. When LBS crashes and restarts, the list of servers are contacted individually to check whether the server is still active, and it rebuilds the tables to server IP mappings. When a client requests server IP addresses for a list of table names, the LBS will return a minimum number of server IPs that contains all of the requested tables.

API

Server

Command	Parameters	Description	Response
ClientConnect	client Client	When a Client receives a new Transaction and has determined which Server to talk to, it will call this function to establish a bidirectional connection. This connection is maintained throughout the entire Transaction.	Returns a Server object. Can return connection Errors.
ConnectToServer	server Server	Server connects to a neighbouring Server (a peer). Servers communicate with each other via this bidirectional connection. This connection is	Returns a Server object. Can return connection Errors.

		maintained throughout the entire Transaction.	
SetRow	tableName string, key string, tableRow Row	Can be called by a Client or a Server. Called by a Client to add a new Row or update an existing Row in the table tableName. The Server validates the Row (whether the field types in the Row matches the schema), and then calls SetRow to propagate the change to its peers. The peers then update their table.	Can return DisconnectedError, RowInvalidError.
GetRow	tableName string, key string	Called by a Client to get the Row from table identified by the key.	Returns the row from the specified table. Can return DisconnectedError, RowDoesNotExistError.
DeleteRow	tableName string, key string, tableRow Row	Can be called by a Client or a Server. Called by a Client to delete the Row. The Server calls DeleteRow to propagate the change to its peers. The peers then update their table.	Returns boolean value representing whether the table was deleted. Can return DisconnectedError, RowDoesNotExistError.
GetTableContents	tableName string	Returns all contents in the table. This function is called by a Server. When a Server is initialized, the Server does not have any data, so it talks to each connected peer by calling this function to retrieve the contents of each table. When the Server crashes and restarts,	Returns all contents of the specified table. Can return DisconnectedError, TableUnavailableError.

		it calls this function as part of the recovery mechanism. This function can also be called by a Client. The Transaction Manager in the Client may decide to retrieve the entire table to process a Transaction.	
TableLock	tableName string	Locks a table to provide exclusive access to the Client. This function is called by a Client that requires access to a table during a transaction. The Server will call SetTableUnavailable for each of its peers to prevent lock the replicated tables.	Returns boolean value representing whether the table was locked. Can return DisconnectedError, TableUnavailableError
TableUnlock	tableName string	Unlocks a table. This function can be called by a Client. Once the Client finishes its transaction, it unlocks the Server's table. However, if the client crashes, then the Server will abort the transaction and unlocks all the tables the client possesses lock for. Within the function, the Server will contact its peers by calling SetTableAvailable to "unlock" the replicated tables.	Can return DisconnectedError
SetTableUnavailable	tableName string	Called by a Server to inform its peers that the table is currently locked and in use.	Can return DisconnectedError
SetTableAvailable	tableName string	Called by a Server to inform its peers that the table is no	Can return DisconnectedError

		longer locked and is available for use.	

Client

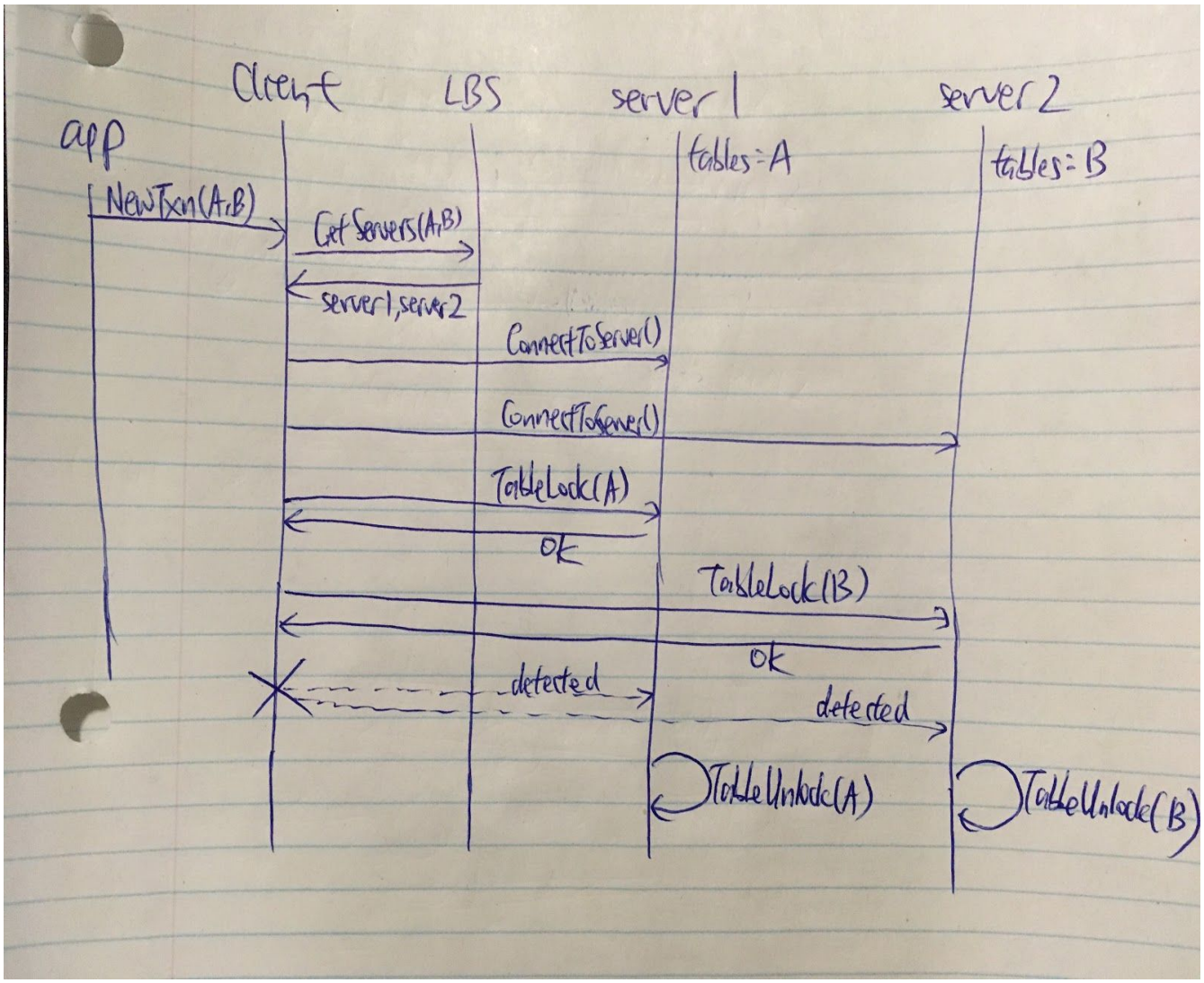
Command	Parameters	Description	Response
NewTransaction	txn Transaction	The Client makes appropriate connections with the Servers and follows the Server semantics to execute the transaction (e.g. GetRow, SetRow, etc.).	Return True if the Transaction has been completed successfully, return False if the Transaction aborted. Can return DisconnectedError

LBS

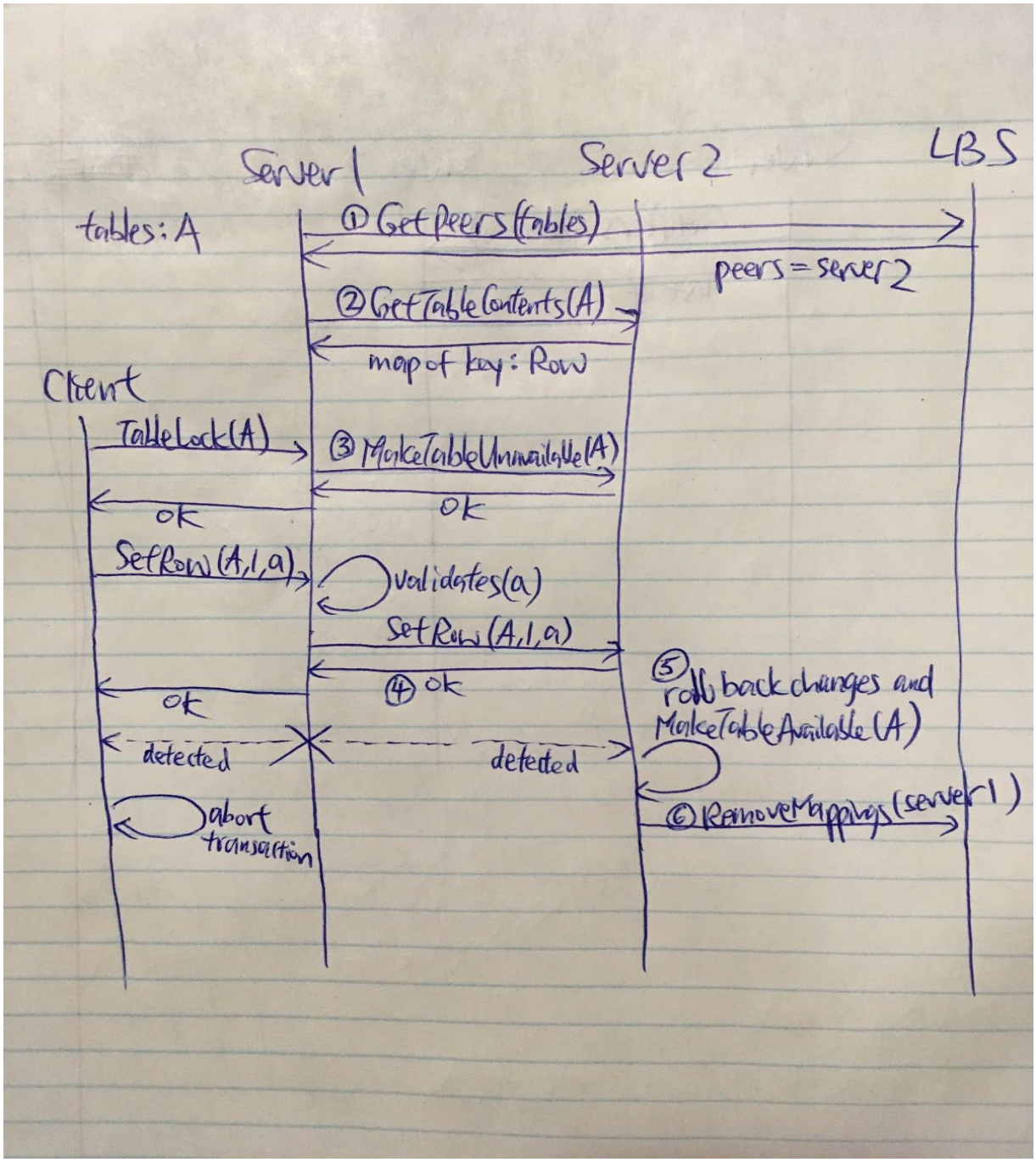
Command	Parameters	Description	Response
AddMappings	tableName string	Add table to server mapping to the Load Balancer Service.	Can return connection-related Errors
RemoveMappings	s Server	When a Server crashes, its connected peers detects the crash and calls RemoveMappings to remove all table mappings for the it on the Load Balancer Service.	Can return connection-related Errors.
GetPeers	tableNames []string	Called by a Server to get a list of peers to connect to.	Return a list of peers. Can return connection-related Errors.
GetServers	tableNames []string	Call by a Client to get a list of Servers to connect to.	Return a list of tableName:IP mappings Can return connection-related Errors.

Space-Time Diagrams

Client-Server Communications



Server-Server Communications



Notes for Diagrams:

1) When a new Server joins the network, it calls AddMappings(server, tableNames) to register its tables, so that when a Client requests a table, the Load Balancer Service can direct the Client to connect to the Server.

2) When a Server is initialized, the Server does not have any table contents for its tables, so it talks to each connected peer by calling this function to retrieve the tableRows contents of each table. When the Server crashes and restarts, call this function as part of the recovery mechanism.

3) The Server will then call SetTableUnavailable(tableName) to each of its peers to prevent the replicated tables from being accessed concurrently.

4) However, if only the connection between the Server and its peer is lost, the Server does not propagate this error back to the Client.

5) On the other hand, if the client crashes, then the Server will abort the transaction and unlock all the tables that the client possesses the lock for. During TableUnlock(tableName), the Server will contact its peers by calling SetTableAvailable(tableName) to make the replicated table available for use by other clients.

6) When a Server crashes, its connected peers detect the crash and call RemoveMappings(server) to remove mappings on the Load Balancer Service, so that a Client will not be directed to a crashed Server.

Testing

We will build applications on the client. The applications will issue different transactions that will be handled by the client. We will log our executions on each node. We will attempt to use GoVector and Shiviz to visualize the events in the DDBS and check that the order of events is correct. To test for crash and recovery, we will set breakpoints at different time points of the program, and force crashing the nodes. All nodes will be deployed on Azure, each node will be using a separate VM with distinct IP addresses.

Additional Features

If time permits, we will design a new protocol for table locking. In the optimized version of the locking protocol, transactions are labelled with a priority number. Transactions with higher priority do not wait at the end of the queue. We reschedule the queue to allow high priority transactions to hold locks before other transactions, regardless of the arrival time of the transaction.

To prevent complexity in our system, we do not allow table creation and table deletions.
We have also considered using Ricart-Agrawala Protocol to ensure global mutual exclusion. However, to achieve this we need all-to-all connection between the clients, which is too much overhead.
We avoided the Durability property for the transaction, because we can restore table states from its peers. We assume that at least one copy of each table is available at all times.

Summary

Challenge	Technique	Benefit
Transactions needs to be consistent, and atomic	Two phase commit protocol. Allow servers to talk to one another when table data is updated.	Global consistency always guaranteed
Transactions needs to independent	Locks at the servers, strict two-phase locking protocol	Global mutual exclusion
Deadlock prevention	Replication of locking states and timeouts	Prevent long wait times due to deadlocks
Improve workload balance for each table	Distributed tables and replication	Availability and reliability
Server failure detection and recovery	Heartbeat and use of LBS as a center for server management	Prevent network partitioning

TimeLine

March 9th, 2018, Friday	Everyone: - Final Project Proposal Submitted Rock: - Start building client Ashley - Start building server Haoran: - Start building load balancer
March 12th, 2018, Monday	Rock - Client can parse transactions, and connect and call server API. - Implement data structures for client transactions. Haoran - Implement data structures for load balancer and tables-to-servers ip mapping - Implement peer list for server - Implement server locks list

	- Implement server transaction manager
	- Implement lock manager
	- Implemented API:
	  - AddMapping, DeleteMapping, GetPeers
	
	Ashley
	- Implemented API:
	  - ConnectToServer

March 19th, 2018, Monday	Ashley
	- Implemented API:
	  - GetRow, DeleteRow, SetRow, GetTableContents
	- Server failure detection and recovery
	
	Haoran
	- Implemented API:
	  - TableLock, TableUnlock, TableAvailable, TableUnavailble
	- Implement 2-phase locking protocol
	
	Rock
	- Implement 2-phase commit protocol

March 23rd, 2018, Friday	- Have scheduled meeting with TA on that day - Start testing - Start final report
March 30th, 2018, Friday	- Feedback from TA - Bug fixing - Work on final report
April 6th, 2018, Friday	- Final code due - Final report due

SWOT Analysis

Strength	**Weakness**	**Opportunity**	**Threat**
We've worked together on project 1.	Not proficient in GoLang	Implement a system to resolve a current issue	Might not implement to full extent
We have flexible schedules.	We have assignments/exams	Tutorials for protocols we will use	

	during last few weeks		
We all have experience programming in groups and on large projects.	We only have 3 members	Get to learn more about distributed databases, which will be valuable in future	
Haoran is interested in locking.	Little knowledge about distributed database systems.		
Everyone has experience with failure recovery.			

References

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