A Distributed Database System Draft

By k7y8, k0a9, c4wa9

*Pictures are hand drawn for now, we'll render proper ones for the final draft.

Introduction

For project two, we would like to work on prototyping a distributed database system. The system should provide reliable service to the clients querying 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 at the global scale. A client that reads and writes to a database must have an idempotent outcome. The sequential order of queries is maintained in the order that they are received by the DDBS.

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 a Distributed Database System (DDBS), where consistency is enforced across all nodes, and a disconnected node storing interrelated data with respect to another node in the network cannot guarantee consistency. However, a DDBS still attempts to provide high availability service to multiple clients concurrently. A centralized Database System cannot achieve high availability, because if it goes down, then none of the clients can access the database. On the other hand, if any node in the system goes down, the entire DDBS should not be affected significantly and is able to continue to provide services to clients.

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 by joining the tables. Different tables are stored in different server nodes, and all tables are replicated in at least 2 nodes. The DNS also contains multiple client nodes that query the server nodes. The clients do not know where each table is stored, therefore the DDBS also has a single directory lookup service node (DNS). The DNS records what tables are stored at each server node, and is contacted by a client every time a transaction is created in the client.

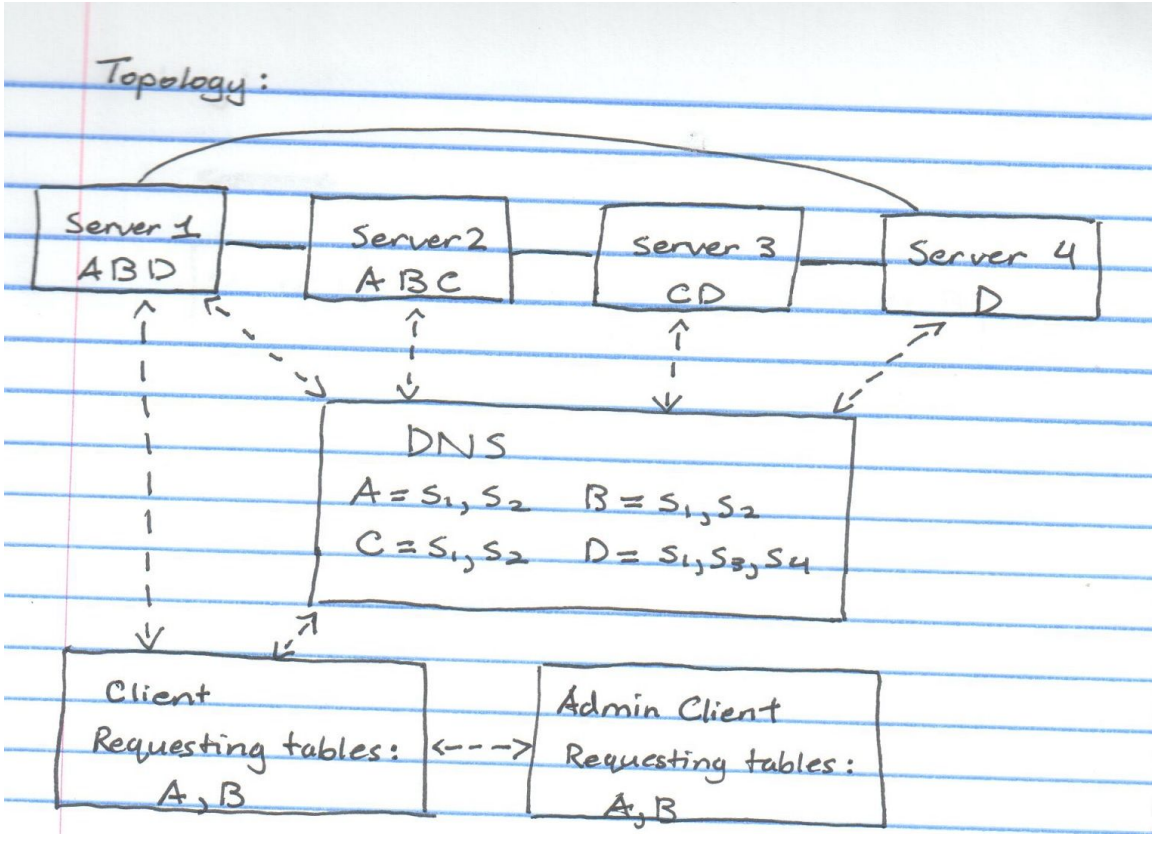
Note that both the client and the server need a database management system locally to manage transaction processing. For example, a client needs to read data from the server and process the operations locally (joining the tables). The server needs to check for integrity constraints on the received data before adding it to its table (inserting a row to the table). The

database management system is also responsible for communications of data with another remote node.

The Topology of the DDBS

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 Z. The contents of the two tables Z must be identical at all times when the servers are servicing requests from clients. Each server knows about the IP address of a DNS node, the server connects to the DNS transiently to inform that a new table is added to the server or a table has been deleted. 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 terminated. Clients can also communicate with each other transiently. The locking states of tables are communicated between the clients, clients that try to lock the same table for a transaction must make an agreement to ensure that only one client holds the lock at one time.

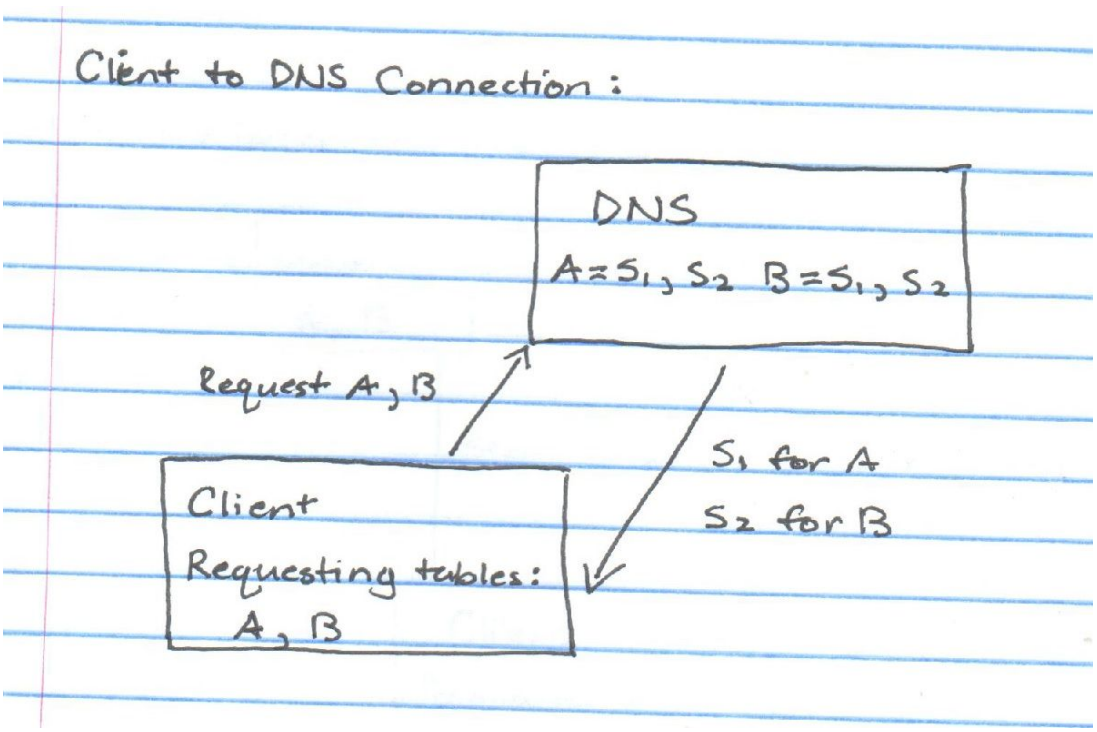
Our model assumes that all nodes (the servers, the clients, and the DNS) can crash, and there are crash recovery mechanisms for each type of node. In addition, we do not have a threat model since we assume that all nodes in the DDBS are trusted. Aside from the networking components that we discuss below, our application design is minimal. We are using our own database design. 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. A transaction that the client creates is a struct that stores all necessary fields for each operation in the transaction.



The figure above illustrates that the distributed database contains 4 tables: A, B, C and D. Tables are replicated. The DNS 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 transient connections (sends a message and then disconnects).

Client-DNS Communications

The DNS server maintains a map of database tables to the server IP addresses. The client queries the DNS server, and the DNS will route the client to the appropriate server.

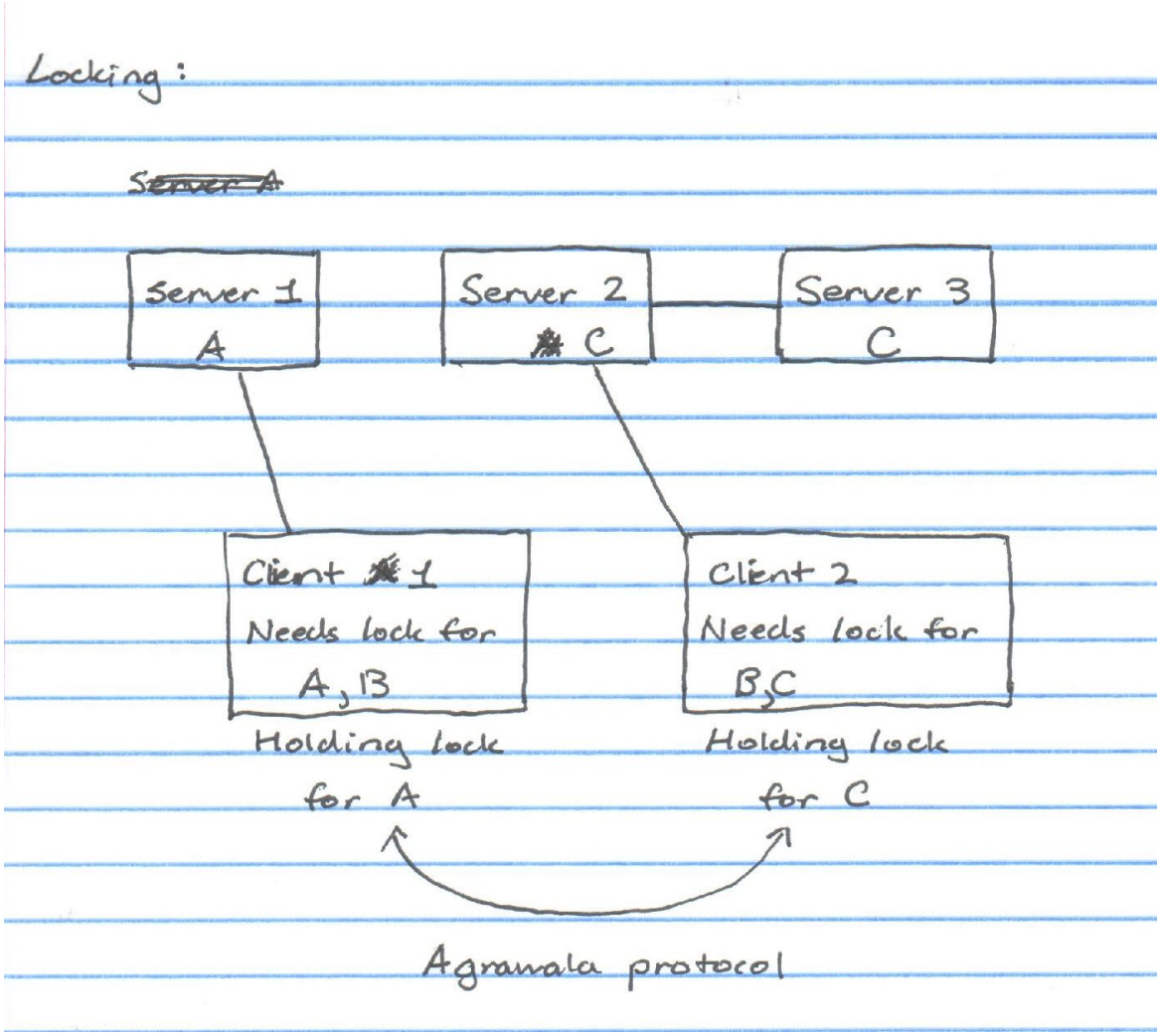


Client-Client Communications

There are two types of clients in the DDBS, a normal client and an admin client. The normal client will be able to perform read and write operations within a transaction, this includes reading rows from tables, inserting and deleting rows of a table. The admin client can additionally create a table on a server and deleting a table. Both the normal client and the admin client needs mutual exclusion from other clients during transactions. To address this issue, we will implement the Ricart-Agrawala algorithm. For each table that a transaction needs, a lock must be acquired. With Ricart-Agrawala, a client will send a message to all other clients, and acquire the lock for the table after it has received a reply from all other clients. The other clients reply once they have released the lock for the table or reply immediately if they do not hold the lock.

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 commit protocol. The client cannot initiate the transaction until it has all the locks needs. However, as soon as a client receives a lock for a table, it will contact the server to say that it has acquired the lock. The reason is that 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.

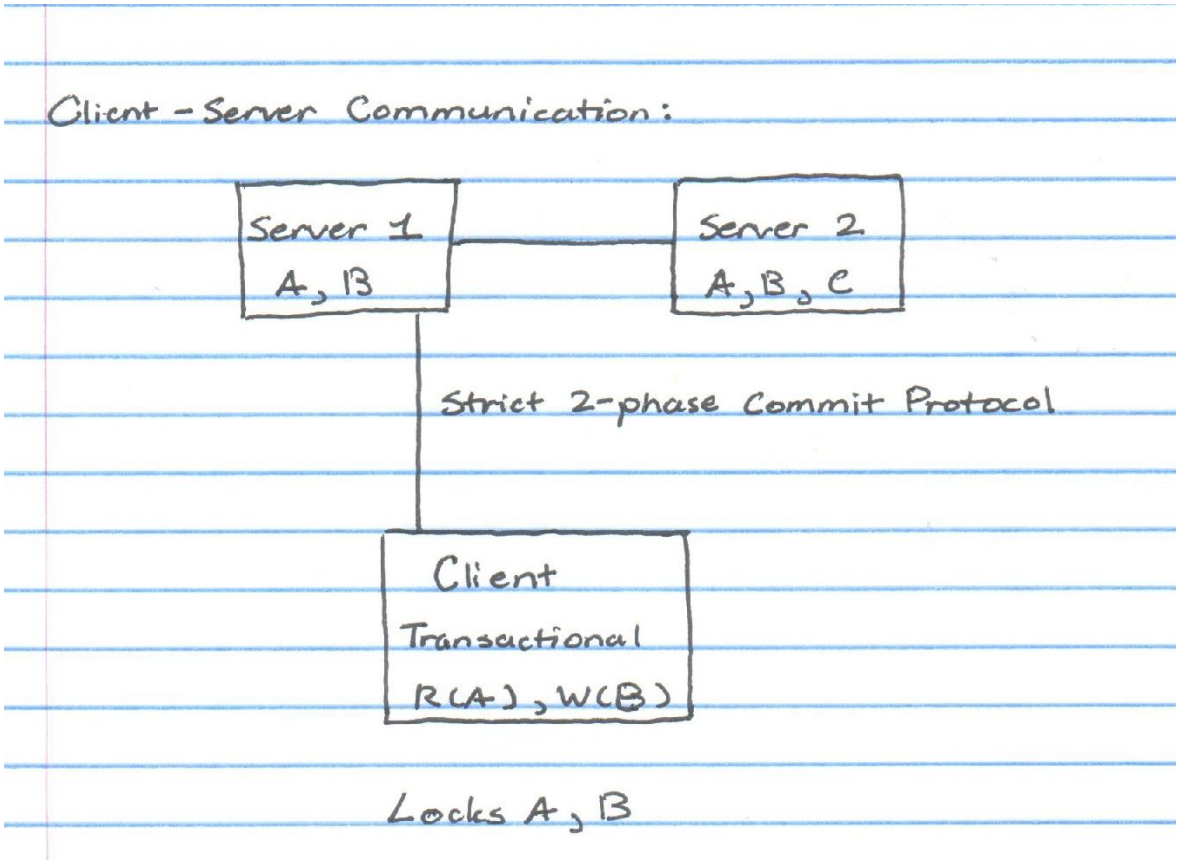
The side effect of Ricart-Agrawala is that once a table is locked, no copies of the table can be accessed by any clients not holding the lock. Therefore a client that holds one lock locks all copies of the table on multiple servers. A side-effect of the locking protocol is that a serial order of transactions is maintained in the database.



The figure above illustrates that clients requesting the same table lock communicate through the Ricart-Agrawala algorithm. Clients holding a lock are connected to the server bidirectionally (and send heartbeats).

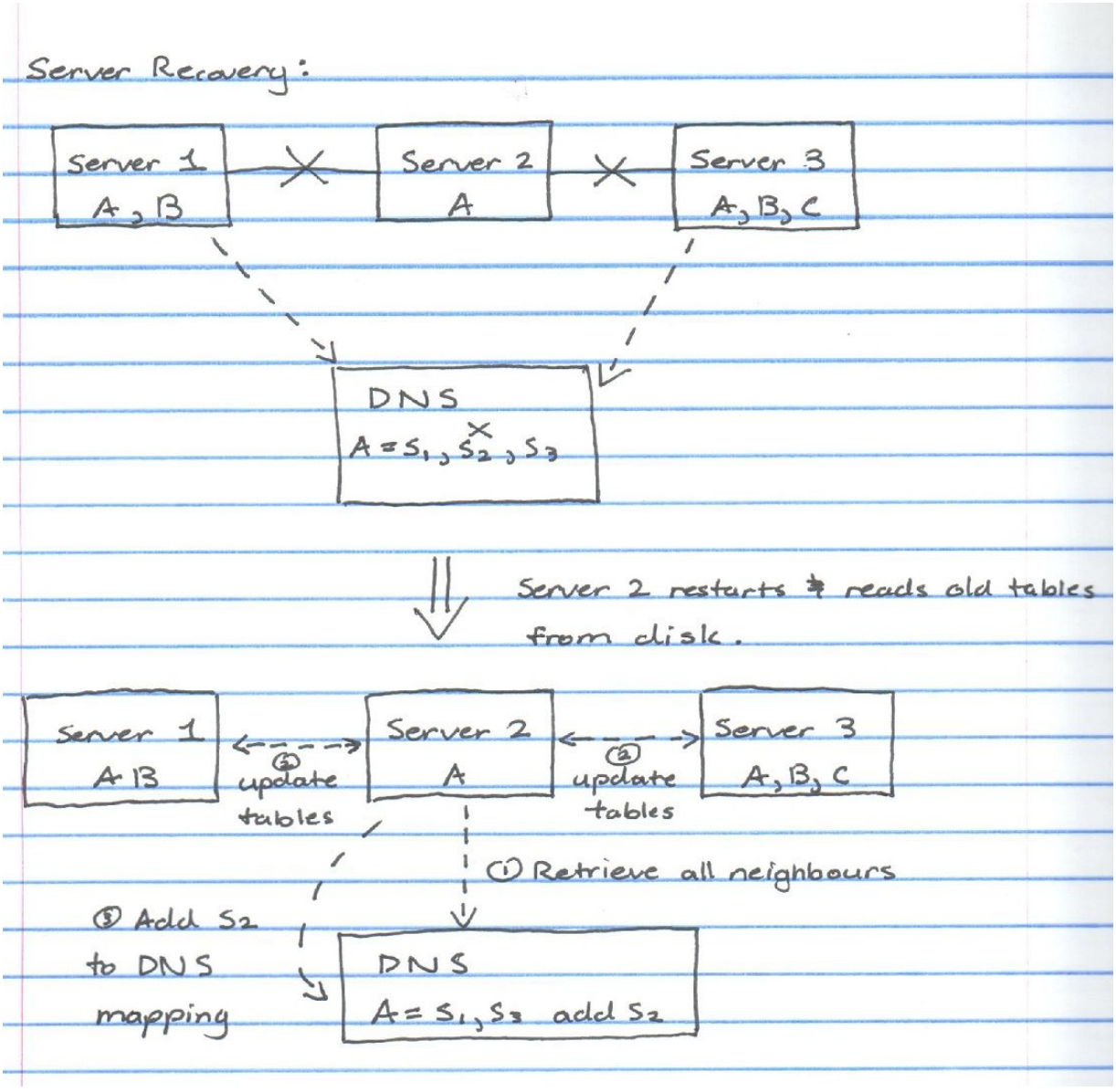
Client-Server Communications

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. A transaction cannot succeed until all servers have stored the new content.



Server-Server Communications

The connections between servers allow failure detection. When a server crashes, its neighbours detect the crash and the inform the DNS. The DNS removes the crashed server's ip from all entries in its table. In order for the server to recover, it must store its tables to disk. When the server restarts, it will read its tables from disk, and contact the DNS to receive the list of servers (with has least one table in common) to connect to. The crashed server will receive all updates from its connected neighbours since the crash. The restarted server then informs the DNS to register its tables in the map.



Testing

We will build applications that use the client. The applications will issue different transactions that will be handled by the client. We will attempt to use GoVector and Shiviz to visualize the events in the DDBS and check that the order of events is correct.

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.

Summary

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

TimeLine

March 2nd, 2018, Friday	Project Proposal Draft
March 9th, 2018, Friday	- Final Project Proposal - Build server, DNS, and client, and establish connections between them - Data structures

March 19th, 2018, Monday	- Table locking - Client-Server data modification - Server failure detection and recovery
March 23rd, 2018, Friday	- Schedule meeting with TA on that day - Test plans - 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 during last few weeks	Tutorials for protocols we will use	No research potential
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	