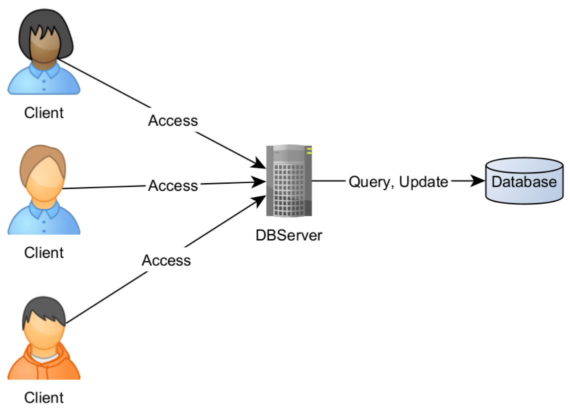
SE424: Distributed Systems	**Recitations 8**
Semester 1 5785	**22 Dec 2024**
Lecturer: Michael J. May	**Kinneret College**

Distributed Mutual Exclusion

In this recitation we will build tools that simulate the mutual exclusion algorithms we learned about in lecture. The system setup is as shown below.



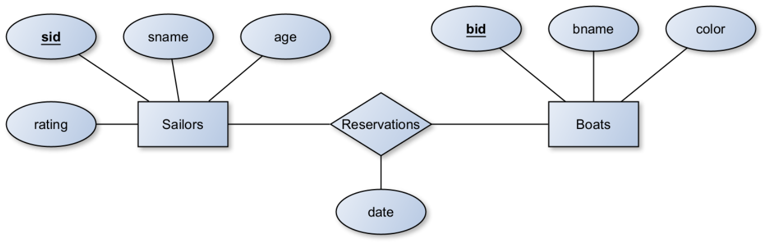
The starter code as given has no protection on the database, so multiple users can access and modify the database at once. This can lead to problems with data consistency. In this recitation we'll add mutual exclusion algorithms to ensure that only one client updates the database server at once.

1 Database Structure

There is a database with sample data that is made available by a database server. Clients can access the database server and thereby read or update the data in the database. The database contains information about sailors, boats, and reservations. The database schema is as follows:

Sailors (<u>sid</u>, sname, age, rating)
Boats (<u>bid</u>, bname, color)
Reservations (<u>sid, bid, date</u>)

The entity relationship diagram for the schema is shown in the following figure:



The database engine backend is a SQLite database file that is found locally on the server.

2 Client Interaction

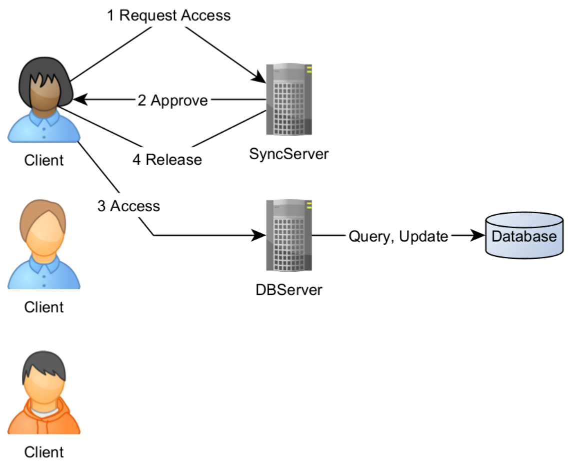
Clients interact with the server using RMI via an interface called `SailorsAccess`. The access class offers an RMI interface for the following operations:

1. List all sailors
2. List all boats
3. Add a new sailor
4. Add a new boat
5. List all reservations for a given sailor (by sailor id)
6. Swap the sailor on an existing reservation with a different sailor

The client knows where to find the RMI server by receiving two parameters from the command line - the IP address of the database server and the name of the SailorsAccess object on the server. Both parameters are required for the client to run.

3 Adding Centralized Mutual Exclusion

As a first step, we'll build the centralized mutual exclusion algorithm explained in class. The `SyncServer` will serve as the single point of contact to allow a Client to interact with the database server. Before a Client can access the `DBServer`, it must first ask the `SyncServer` permission first. After it receives approval, the Client can proceed to use the database. When it's done, it sends the `SyncServer` a release message.



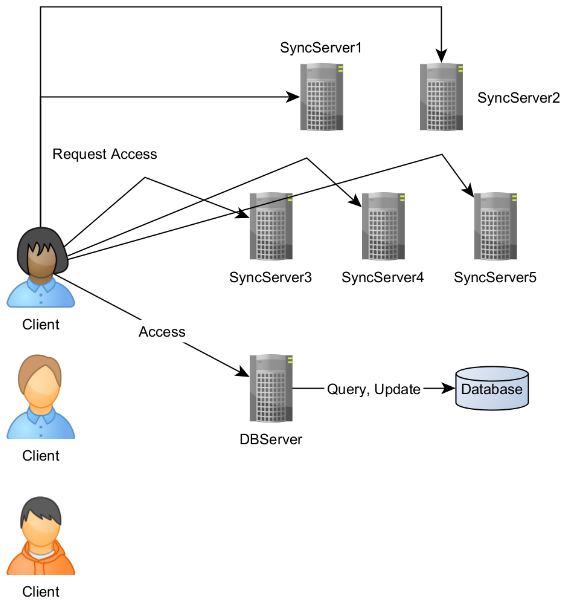
If a Client contacts the `SyncServer` when another Client is currently using the `DBServer`, the `SyncServer` waits to respond until it is possible to say yes.

3.1 What to do

1. Fill in the logic in the `SyncServer` package to support the functionality described. You can implement the logic using RMI or using socket communication, whichever you prefer.
2. Modify the Client program to ask permission from the `SyncServer` before it contacts the `DBServer`
3. Add a Release command to the Client menu that causes the Client to release the lock on the `DBServer`.

4 Adding Decentralized Mutual Exclusion

For our second step, we'll build the decentralized mutual exclusion algorithm explained in class. We'll make $n = 5$ copies of the SyncServer that all operate independently. In order for a Client to interact with the DBServer, it must first gather approvals from a majority ($m > n/2$) of the `SyncServers`. Let's start with a majority of $m = 3$. After a Client receives approval from m SyncServers, it can access the database. After a Client finishes using the DBServer



The SyncServer will respond with an approval or denial immediately when contacted, so Clients are not left hanging. If a Client tries to gather a majority, but fails to do so, it should wait 30-60 seconds and then try again automatically.

4.1 What to do

1. Modify the logic in the `SyncServer` package to support the functionality described. You can implement the logic using RMI or using socket communication, whichever you prefer.
2. Modify the Client program to ask permission from the majority of `SyncServers` before it contacts the `DBServer`
3. Add a Release command to the Client menu that causes the Client to release the locks on the `DBServer` at all appropriate `SyncServers`.
4. Modify the Client program to print a failure message if it doesn't get a majority and then retry in 30-60 seconds - add a random wait time timer.

4.2 Experiments

Experiment 1 Experiment with having more than one Client attempt to gather a majority at once. Can you see how starvation would occur?

Experiment 2 Add logic to the `SyncServer` to have it "forget" its state every so often, allowing it to Approve more than one Client at a time. How often does the `SyncServer` have to reset to lead to an unsafe mutual exclusion situation?