CS 171 Fall 2024
Programming Assignment II

Assigned: October 16, 2024
Due On: Thursday October 31, 2024, 11:59 PM

Recall that a Dictionary is a simple data structure that stores a list of pairs (key, value), where each key is unique and has a value associated with it. For example, a dictionary of student phone numbers would have the perm number as the key and the student's phone number as the value. A simple dictionary service could store the dictionary in an array and a server and would support 2 operations: insert (**int** perm, **int** phone), which would insert this pair in the dictionary and lookup(**int** perm), where the server returns the phone associated with this perm number, if the perm number is in the dictionary. In PA1, we addressed how to scale a dictionary by sharding or partitioning it over 2 different servers. In this aassignment, we will explore how to make the dictionary fault-tolerant by replicating it on multiple servers and load balance the read requests among the two servers. Hence if one server fails (crashes in this case), clients can still access the data in the dictionary from other servers.

1 Project Overview

In this assignment, you will develop a simplified version of a replicated dictionary service. The focus will be on load balancing rather than fault-tolerance. The Dictionary Service will be implemented using 2 servers. We will assume 3 clients. Clients issue commands insert or lookup. Clients must ensure that their updates (hence insert operations) do not interfere with each other. Hence, before a client executes an insert operation, it must receive mutual exclusive access to the dictionary service from the other clients. In this assignment, you will use a **variation** of Lamport's Mutual Exclusion Protocol which is equally correct, but helps in ensuring determinism and makes it easier to grade.

When a client wants to execute an insert operation, it runs the mutual exclusion protocol (inserts in queue, sends request,and waits for replies), and when it has permission (has received all replies and is at the head of the queue), it sends the insert(perm, phone) operation to **both** dictionary servers. The two servers add (perm, phone) to their local copy of the dictionary, and return SUCCESS to the client. Once the client receives **both** SUCCESS messages, it issues its RELEASE message according to Lamport's mutual exclusion protocol.

When a client wants to perform a lookup operation, the request is sent **directly to one of the servers** based on the below criteria:

- For client1 and client2, they send the lookup(perm) request to the primary server.
- For client3, it sends the request to the secondary server.

The server executes the operation and returns the corresponding phone or NOT FOUND to the client. When the client receives the response, it prints it on its terminal.

In this simplified assignment, you will not have to handle any actual failures. The focus is simply of maintaining the 2 copies of the dictionary, and load balancing the lookup operations on the two servers (lookups are typically a lot more frequent than insert operations) So, do not worry about failures or what to do when a failure happens. We will deal with this later.

1.1 Variation of Lamport's mutual exclusion

This assignment requires you to implement a modified version of Lamport's mutual exclusion algorithm for insert operation. This variation maintains correctness while being more deterministic, which will aid in grading. The core structure of the algorithm remains the same, with the key difference being that Lamport timestamps are **only incremented** when **requests are received or new requests are sent**. The following example clarifies this approach:
Consider a system with 3 processes: P1, P2, P3.

1. P3 sends a REQUEST (1,3) to P1 and P2.
2. Upon receiving the request, P1 and P2 increment their Lamport timestamps to (2,1) and (2,2), respectively.
3. P1 and P2 then send REPLIES to P3, without changing their Lamport timestamps.
4. P3 receives REPLIES from both P1 and P2 and accesses the resource, again without incrementing its Lamport timestamp.
5. After using the resource, P3 sends RELEASE messages to P1 and P2, and none of the processes update their timestamps.
6. P2 subsequently sends a new REQUEST (3,2) after incrementing its Lamport timestamp.
7. P1 and P3 receive this new REQUEST from P2 and update their timestamps to (4,1) and (4,3), respectively.
8. The algorithm continues with P1 and P3 sending their REPLIES, P2 accessing the resource, and finally sending RELEASE messages to P1 and P3.

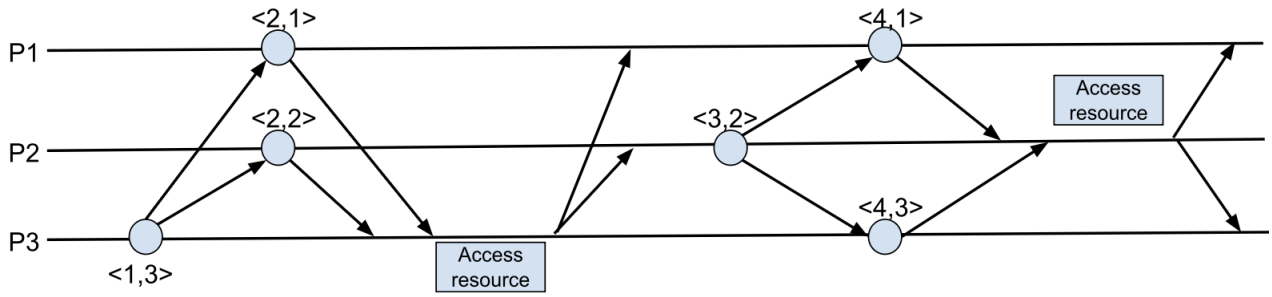

Figure 1: Causality diagram depicting the variation of Lamport's mutual exclusion algorithm.

2 Implementation Details

- You can decide how to implement the data structure for storing the dictionary at both the primary and secondary servers. Needless to say, this data structure must support both insert and lookup. For this assignment, we do not care how efficient each of these operations is implemented.

3 System Configuration and User Interaction

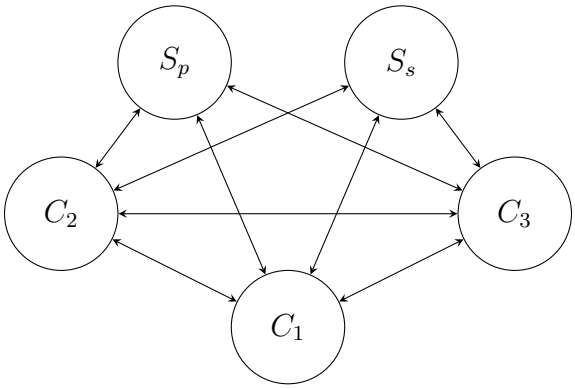

Figure 2: Three clients connected to two servers.

1. There are exactly 3 client processes and 2 server processes as shown in Figure 2. All processes should run on the terminal. No front-end UI is necessary.

2. All of your processes should be able to take in command line parameters for a port number. Your programs will all receive the same port number from the autograder, and your programs can use the next 20 port numbers for any pre-defined ports as you see fit. See Section 5 for more details.

3. Each client process has a unique id: $\boldsymbol{C_1}$, $\boldsymbol{C_2}$, and $\boldsymbol{C_3}$. The two servers have id $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$. When starting a client process, it should connect to both the servers at pre-defined ports. The two servers do not need to be connected.

4. Each client must handle the following input from the user:

   - *lookup* + the perm of interest. The client displays the corresponding phone or NOT FOUND on its terminal.
   - *insert* + perm + phone. The client displays SUCCESS on its terminal.

   Each primary server should handle the following input:

   - *dictionary*: The server simply displays the current state of its dictionaries.

   Both clients and servers should also handle the following input to gracefully exit:

   - *exit*. This should close all socket connections, and flush the standard output. Then, it can call system exit to gracefully exit the program.

5. To simulate a distributed environment, you can run all 5 processes on the same machine. To simulate message-passing delay over the network, you should **add a 3 second delay before processing a received message**.

6. Use message-passing primitives TCP for inter-process communications. TCP ensures FIFO communication, and hence satisfies the requirements of Lamport's Mutual Exclusion Protocol.

7. There is no restriction on the choice of programming language. However, you must use socket programming for communication to simulate an actual network environment.

4 Example Scenarios

When running the five entities – S_p, S_s, C_1, C_2, C_3 as separate terminal processes, the output in each terminal should appear as shown below for the specified sequence of commands.

4.1 Sequence of commands

```
C3: insert 3 3
wait for time > 9 seconds
C2: lookup 3
wait for time > 3 seconds
C1: insert 1 1
wait for time > 9 seconds
Sp: dictionary
Ss: dictionary
```

4.2 Expected output

4.2.1 Output for S_p

```
Successfully inserted key 3
3
Successfully inserted key 1
dictionary
{(1, 1), (3, 3)}
```

4.2.2 Output for S_s

```
Successfully inserted key 3
Successfully inserted key 1
dictionary
{(1, 1), (3, 3)}
```

4.2.3 C_1

```
Received request (1, 3)
Received release for request (1, 3)
insert 1 1
Sending request (3, 1) to Client 2
Sending request (3, 1) to Client 3
Received reply for (3, 1), incrementing reply count to 1
Received reply for (3, 1), incrementing reply count to 2
Sending operation 'insert 1 1' for request (3, 1) to primary server
Sending operation 'insert 1 1' for request (3, 1) to secondary server
Response from Server 2 for request (3, 1) : Success
Response from Server 1 for request (3, 1) : Success
Sending release for request (3, 1) to Client 2
Sending release for request (3, 1) to Client 3
```

4.2.4 C_2

```
Received request (1, 3)
Received release for request (1, 3)
lookup 3
Response from Server 1 for operation 'lookup 3' : 3
Received request (3, 1)
Received release for request (3, 1)
```

4.2.5 Output for C_3

```
insert 3 3
Sending request (1, 3) to Client 1
Sending request (1, 3) to Client 2
```

```
Received reply for (1, 3), incrementing reply count to 1
Received reply for (1, 3), incrementing reply count to 2
Sending operation 'insert 3 3' for request (1, 3) to primary server
Sending operation 'insert 3 3' for request (1, 3) to secondary server
Response from Server 2 for request (1, 3) : Success
Response from Server 1 for request (1, 3) : Success
Sending release for request (1, 3) to Client 1
Sending release for request (1, 3) to Client 2
Received request (3, 1)
Received release for request (3, 1)
```

5 Submission Instructions

- Files should be submitted individually, without zipping them or placing them inside another folder.
- Do not submit any executables or binaries. Your code can compile into executables or binaries, but you shouldn't be submitting any. Our autograder will check for this.
- Our autograder will start your processes in the following order: primary server, secondary server, client 1, client 2, and then client 3.
- Your Makefile should include the following rules:
  - compile (Can be left empty depending on programming language used)
  - primary
  - secondary
  - client1
  - client2
  - client3
- Your Makefile and program should be able to accept command line parameters for a port number variable named PORT. (For autograder reasons, it must be spelled exactly "PORT"). You are free to use the 20 ports starting from PORT for any predetermined sockets however you see fit. For example, if PORT = 9000, then your programs can use port numbers 9000 to 9019 however you'd like. We will invoke your programs via "make primary PORT=9000", "make secondary PORT=9000", and so on with the same port number for all of your processes. Examples of how to configure your Makefile can be found in Section 5.1.
- We will improve the Gradescope autograder by using different port numbers for different test cases to avoid issues with the kernel taking time to clear used sockets. Also, this allows us to have Gradescope run multiple tests simultaneously (on the same machine) to improve the speed of the autograder.

5.1 Makefile Examples

Below are some examples of what your Makefile could look like (with proper tabbing):

```
# Note that for autograder reasons,
# the following variable must
# be spelled exactly PORT!
PORT ?= 9000
# Default min port=9000 if
# not provided

# Compile command:
# make compile
compile:
    g++ -o server server.cpp
    g++ -o client client.cpp

# Run command:
# make primary PORT=9020
primary:
    ./server primary $(PORT)

# Run command:
# make secondary PORT=9020
secondary:
    ./server secondary $(PORT)

# Run command:
# make client1 PORT=9020
client1:
    ./client 1 $(PORT)

# Run command:
# make client2 PORT=9020
client2:
    ./client 2 $(PORT)

# Run command:
# make client3 PORT=9020
client3:
    ./client 3 $(PORT)
```

```
# Note that for autograder reasons,
# the following variable must
# be spelled exactly PORT!
PORT ?= 9000
# Default min port=9000 if
# not provided

# Compile command:
# make compile
compile:
# Nothing to compile

# Run command:
# make primary PORT=9020
primary:
    python3 -u server.py primary $(PORT)

# Run command:
# make secondary PORT=9020
secondary:
    python3 -u server.py secondary $(PORT)

# Run command:
# make client1 PORT=9020
client1:
    python3 -u client.py 1 $(PORT)

# Run command:
# make client2 PORT=9020
client2:
    python3 -u client.py 2 $(PORT)

# Run command:
# make client3 PORT=9020
client3:
    python3 -u client.py 3 $(PORT)
```