Homework 2 - Distributed Shared Memory

- Course : **Operating Systems**
- Deadline : **2018/05/17 23:59**
- Demo : **2018/05/18 16:30 - 17:30**
- Contact : Che-Pin Chang (張哲彬), TA, jalex.cpc@gmail.com

Overview

For inter process communication, there are two mechanisms can meet the need that includes message passing and shared memmory approaches. In distributed computer environment, we can also use **distributed shared memory** and **message broker** to meet the need of cross-machine communication. In this assignment, we are going to implement a client-server communication service, which is based on a distributed shared memory architecture.

Virtual Bank

You are the owner of a digital bank in network and provide financial services to your customers. For serving, there are multiple available web ATMs (called socket server) that customers can access them via specific client (called socket client). To make sure that accounts' data can be consistent across different ATMs, you need a sharable account manager system to handle it.

Tutorials

For implementing our virtual bank service, we divide specifications into three part, and each part is a upgrade from the former.

Basic : One-to-one client-server with blocking Req / Res

In this part, we are going to build a simple client-server service that only support one client to one server communication. This service can be divided into 3 components.

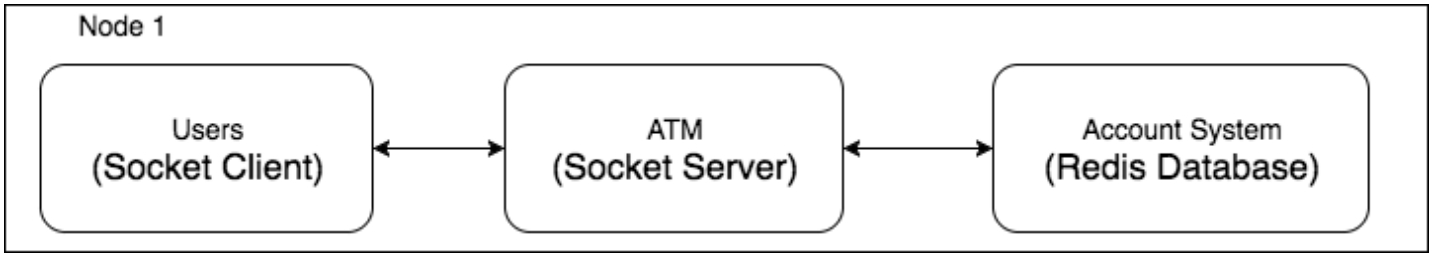


Figure 1. System overview of one-to-one client-server

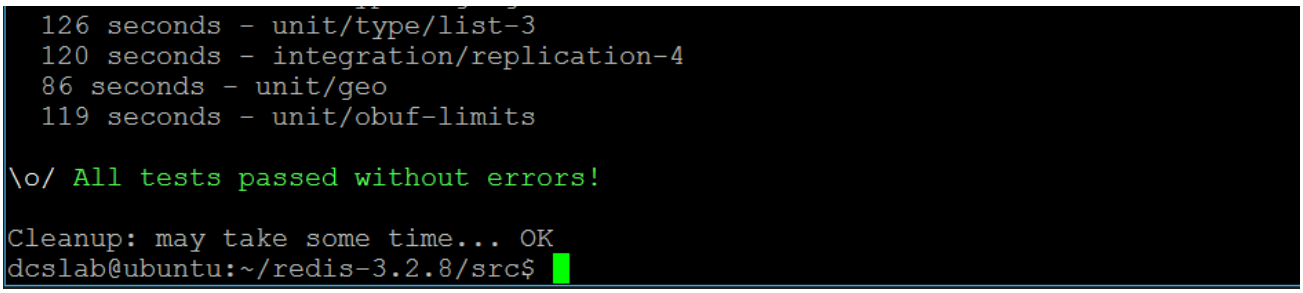
- Account System
  - Account system is a distributed shared memory service that used to manage all sharing data. e.g. transaction logs, and accounts' deposits.
  - We choose Redis database as distributed data sharing mechanism to build our account system. The reasons are that:

    1. Redis is a open source, in-memory database. All operations of Redis are in memory, and make it is fast enough to achieve high speed access across multiple nodes.
    2. Redis had already support built-in replications, transactions, and orders consistency. It would decrease the difficulty to handle cross-server communication and race condition problem.

  - Installation
  - Validation

    ```
    $ sudo apt install tcl
    $ cd <directory_of_redis>
    $ make test
    ```

    Result

- User client
  - User client are represented customers whom want to access your web ATMs and manage and manipulate thier deposits.
  - There are 4 actions that customers would wnat to do:

command format	description	example
init <account> <initial deposits>	Registering a user account with initial money.	init Alice 1000
save <account> <money>	Saving money to a specific account.	save Alice 2000
load <account> <deposits>	Loading their deposits from a specific account.	load Alice 500
remit <src_account> <des_account> <money>	Remitting money from their account to a specific account.	remit Alice John 500

- In addition to customers' actions, there is an extra command sent by the client - **end**.
  - This command means that whole transaction operations are finished.
  - After sending 'end' command, the client would expect to recieve a summary report about transactions. This report would have following information:
    1. Sorted account list with thier deposits
    2. Success rate of transactions.
  - This command is considered as a signal that would not be counted as a transaction operation.
- The client should have the capability to read a input file, which contains commands by line.
- Example

```
// input
init Alice 1000
init John 1000
init Andy 500
save Alice 1000
save Andy 2000
load John 1000
remit Alice John 1500
remit Andy John 2500
end

// output
alice : 500
andy : 0
john : 4000

success rate : (8/8)
```

- Web ATM server
  - Web ATM server is represented as a indivisual financial service agent that processs customers' requests and check if these requests are legal or not.
  - When the ATM server recieve a request, it should to validat the request and does coresponding actions.

recieved commands	**error detection**	**operations**
init	1. Check if account name has registered.	Set account name in account system with specific deposits.
	2. Check if initial deposit is non-negative integer.	
save	1. Check if account name has registered.	Increase deposits in specific account.
	2. Check if money is non-negative integer.	
load	1. Check if account name has registered.	Decrease deposits in specific account.
	2. Check if money is non-negative integer.	
	3. Check if there has enough money in specific account.	
remit	1. Check if source account name has registered.	Decrease deposits in source account. Increase deposits in destination account.
	2. Check if destination account name has registered.	

	3. Check if money is non-negative integer.	
	4. Check if there has enough money in specific account.	
end		Retrieve all accounts' info and the transaction summary from Account system. Sending the report back, and reset account system.

- Notice that.
  - The name of an account is lowercase in account system. Hence, 'Alice', 'aLice', and 'ALICE' are all equal to 'alice'.
  - During demo, you can not reset your ATM server. And, we would give you demo input files in serial.

    ```
    $ python3 server.py &
    ...execute server in background
    $ python3 client.py demo1
    .......report1
    $ python3 client.py demo2
    ......report2
    $ python3 client.py demo3
    ```

- Example of error detection

```
// input
init Alice 1000
init alice 500 // error for registered account
init JOHN 500
init john 1000 // error for registered account
save aLice 1000
load alIce 1000
load joHN 1000 // error for not have enough money
save joHn 500
load john 1000
remit alice BoBo 1000 // error for unregistered account 'BoBo'
end

//output
alice : 1000
john : 0

success rate : (6/10)
```

- Blocking Req / Res comminucation flow
    - In the basic mode, there are only one client and one server. And, when the client send a request, it would wait the response back and then send the next command.
    - Since all commands are sent in sequential, there is no risk for unconsistency.
    - You could use **TCP/IP** socket to achieve comminucation, but not limit to.

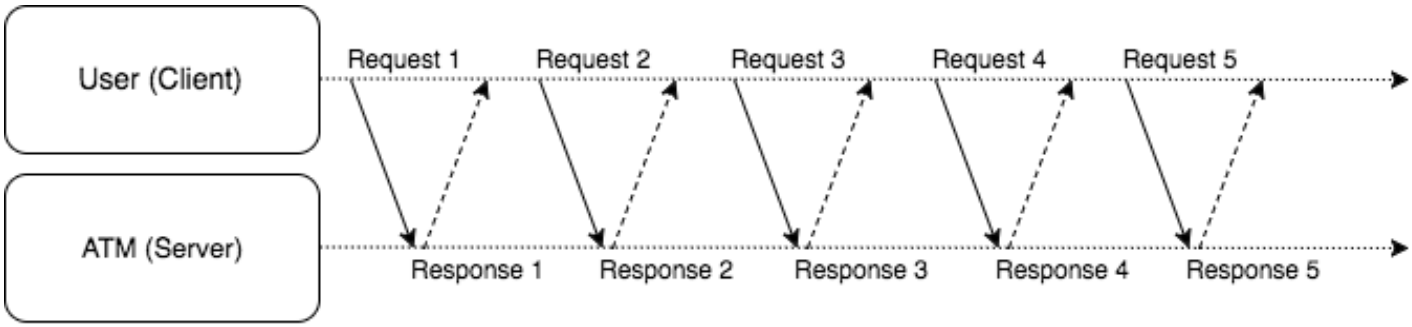


Figure 2. Flow diagram of blocking req / res in one-to-one client-server mode.

Upgrade 1 : One-to-many client-server with blocking Req / Res

Now, we are going to upgrade the service to one-to-many client-server mode. In this mode, the client should have the capability to commnunicate with multiple servers.

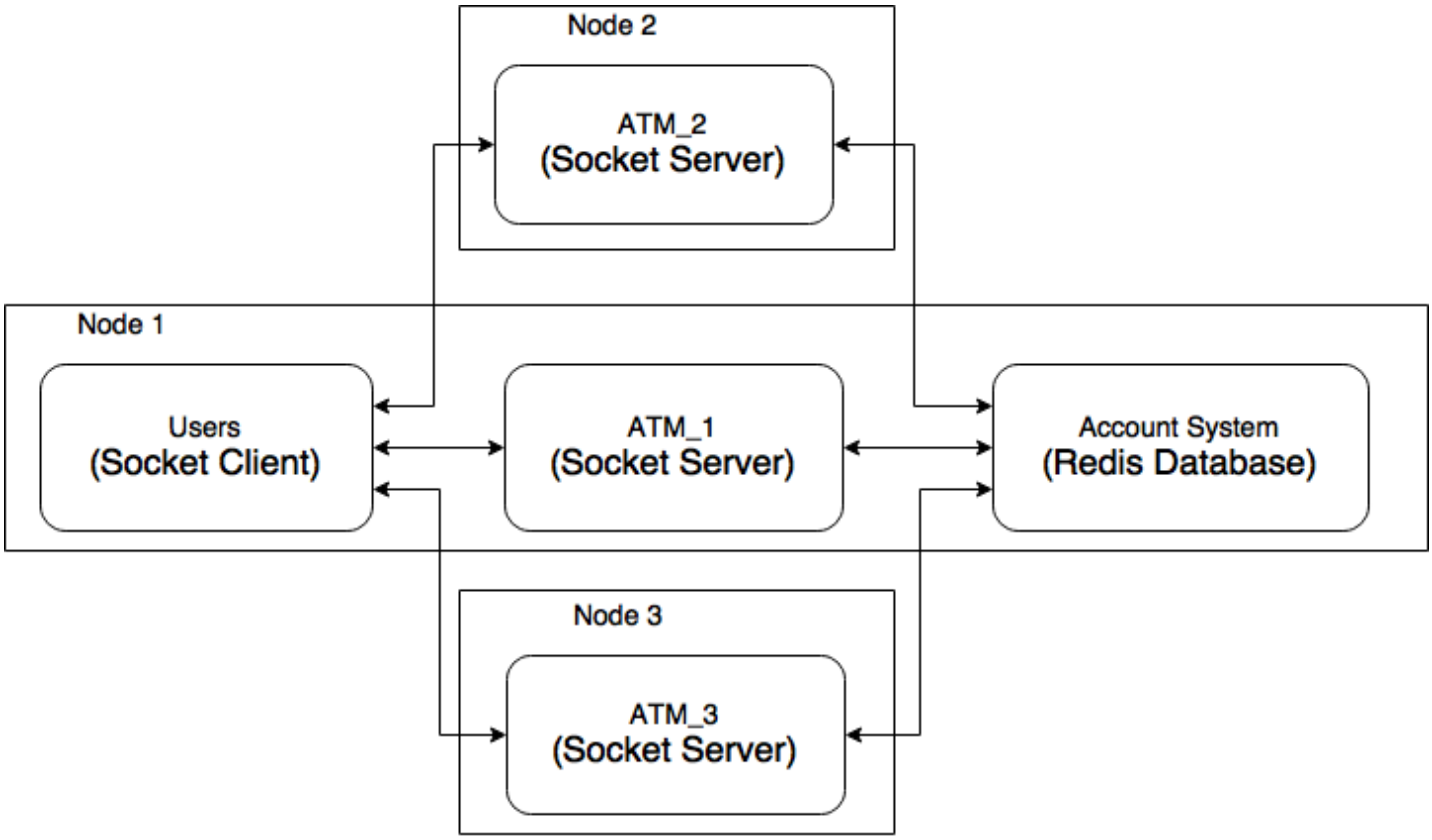


Figure 3. System overview of one-to-many client-server

- Create another two nodes for other ATM servers, named **ATM_2** and **ATM_3**.
- The client would send commands to different ATM servers in circular orders.
    - e.g. ATM1 -> ATM2 -> ATM3 -> ATM1 -> ATM2 ... and so on.
    - The executing commanad line is still simillar to the basic version.

```
node1:$ python3 server.py &
node2:$ python3 server.py &
node3:$ python3 server.py &
...execute all servers in background
node1:$ python3 client.py demo1
.......report1
node1:$ python3 client.py demo2
......report2
node1:$ python3 client.py demo3
```

- Notcie that
    - Although, there are several servers to recieve commands, the result of every input files should be consistent with one-to-one version.
    - Since we still use blocking req /res communication mode, there is no risk for race condition and reorder problem.

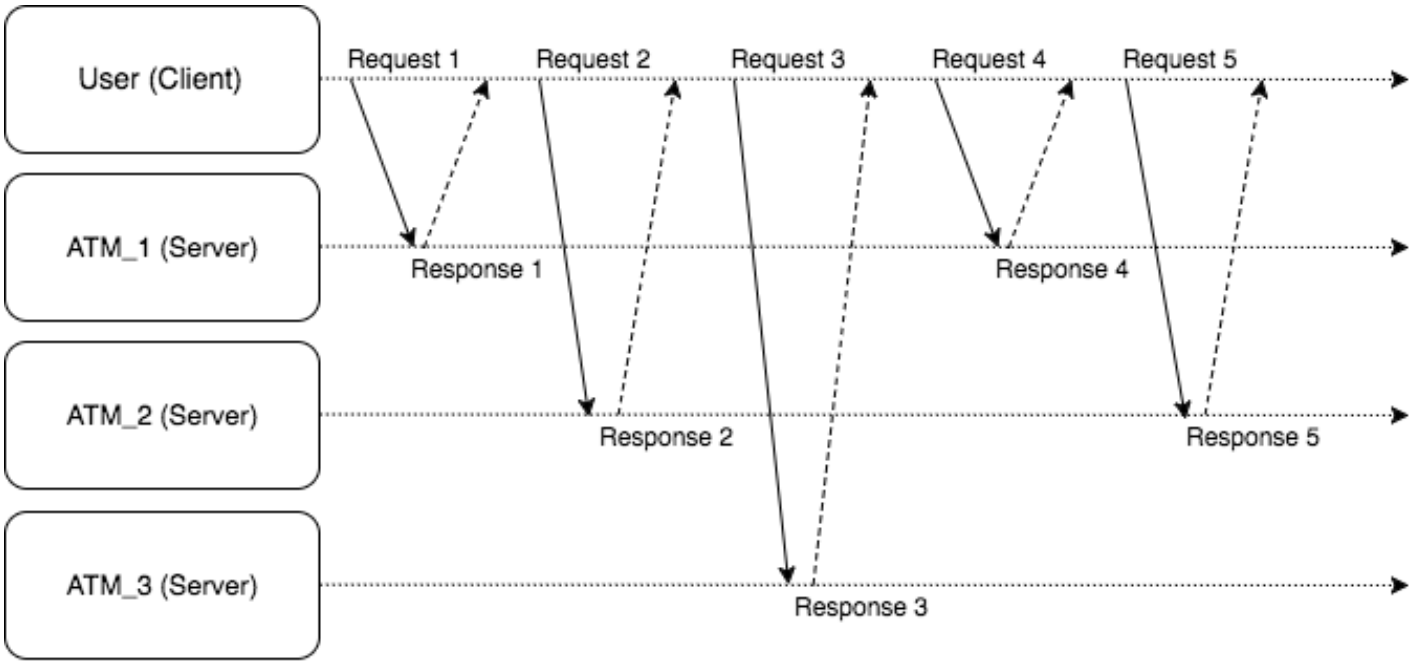


Figure 4. Flow diagram of blocking req / res in one-to-many client-server mode.

Upgrade 2: One-to-many client-server with non-blocking Req/Res

Now, let us upgrade the service again. In this upgrade, we discard the blocking req / res comminucation flow mode. Insteadly, we set a specfic **time interval T** to trigger the event of sending request.

- Notice that,
  - The client would still send commands to different ATM servers in circular orders but cannot to wait the response back.
  - You should to define a suitful time interval T to make all commands are still in sequential order and finish commands as soon as possible.
  - Since there is no promised sequential consistency, please taking care of race condition problem.
  - You can refer dislock, which is a distributed lock mechanism supported by redis, to help maintain accounts' info.

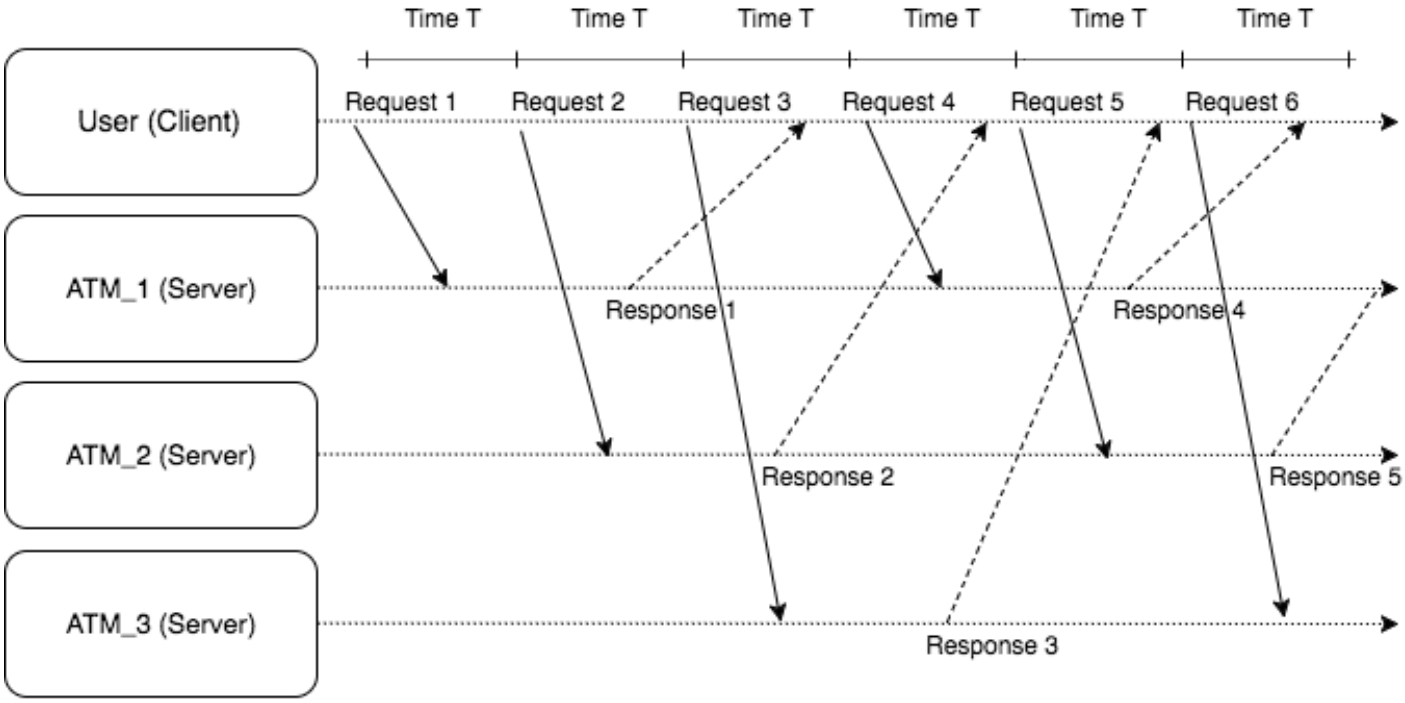


Figure 5. Flow diagram of non-blocking req / res in one-to-many client-server mode.

File Submition

- In this assignment, you need to upload your source code of client and server to E3 campus in the deadline. You only need to upload the highest version. e.g. you have basic v1, upgrade1 v2, and upgrade2 v3, please upload v3.
- The format is **<student_id>_client.xx** and **<student_id>_server.xx**. If you have more than two files, please zip them as **<student_id>_code.zip**.
- TAs would validat your source codes by cheating detection. Please finish the assignment by indivisual.

Grades

For demo, TAs will prepare serveral demo files to test your service. You only need to use your service with highest versoin during demo.

Complete all demo tests:

- Basic version would get **60%**
- Upgrade 1 would get **75%**
- Upgrade 2 would get **90%**
- Upgrade 2 and its time interval T is **less than 10 ms**, would get **100%**

References

- Redis

    - Introduction
    - Dist Lock

- TCP/IP Socket Programming
    - Example in C/C++
    - Example in Java
    - Example in Python