CS425
MP2

Name	NetID
Funing Xu	fxu12
Wei Yang	weiyang4

Implementation of consistency models:
Two consistency models, Eventual Consistency and Linearizability Consistency are implemented via message passing. Specifically, unicast via TCP and UDP sockets, multicast using UDP unicast and total ordering multicast are implemented and used to develop both consistency models. Before we delve into the implementation detail of both models, we will briefly go through the fundamental components such as unicast, B-multicast, and total ordering multicast, which we will only mentioned them abstractly when describe the implementation of consistency models.

1. Unicast:
There are two types of unicast developed:

i) **Unicast via UDP socket:** Upon the start of server program, each server in the group creates a UDP socket and binds to it. When serverA needs to send message to serverB, it launches a new thread to send the message to serverB with specified IP and Port. A random delay will be added using a timer to start this thread.
ii) **Unicast via TCP socket:** Server and clients are communicated via TCP sockets. A server creates a TCP socket using its assigned IP and Port, and then binds and listens to it. A client creates and TCP socket and connects to it. After a client established a connection to a server, it will launch a thread to handle requests from this client. When server sends message to a client, or a client sends message to a server, one will launch a new thread to send the message using created TCP socket. Here no delay is be added.

2. Multicast:
To distinguish from total ordering multicast, we call a basic multicast as B-multicast. What B-multicast does is to unicast messages via UDP socket to all servers in the group.

3. Total Ordering Multicast:
A textbook total ordering multicast is used for our implementation. This implementation requires a sequencer to decide the order of messages delivering as well as to send out the sequencing numbers. Here we elect server1 as sequencer, which will initialize a consecutive increasing sequencing numbers start from 0.

When a sequencer receives a message, it multicasts an order message with the message ID to all servers in the same group. And then it will update the sequencing number.

When a server in a group starts, a sequencing number = 0 and two empty hold back queues are initialized. One of the queues is to store messages sent by other servers, and the other is to store messages from the sequencer.

Total ordering using a sequencer

1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
B-multicast($g \cup \{sequencer(g)\}$, <*m, i*>);

On B-deliver(<*m, i*>) *with* $g = group(m)$
Place <*m, i*> in hold-back queue;

On B-deliver(m_{order} = <"order", *i, S*>) *with* $g = group(m_{order})$
wait until <*m, i*> in hold-back queue and $S = r_g$;
TO-deliver m; // (after deleting it from the hold-back queue)
$r_g := S + 1$;

2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver(<*m, i*>) *with* $g = group(m)$
B-multicast(*g*, <"order", *i*, s_g>);
$s_g := s_g + 1$;

When a server receives a message from other servers, it firstly puts the message into the holdback queue. Then it checks sequence queue to see if the corresponding order message has already arrived. If it does arrive and the sequencing number is matched, this message will be delivered at the server, and the receive sequence number will also be incremented by 1.

On the other hand, when an order message is received, a server firstly checks the holdback queue to see if it contains the message with corresponding ID. If it does have this message and the sequencing number matches, then the server delivers and updates the local sequence number. Also, the server will continue to check the matched message ID and sequencing numbers and deliver as many messages as it can.

Consistency Models:

1. Eventual Consistency:
As a weak consistency model, Eventual Consistency guarantees that all processes will eventually have all shared objects in the same consistent state if no new updates are made. Two important properties should be mentioned for this model: 1) a shared object at a process can be in any state before it converges; 2) upon convergence, shared objects in all processes should have the same state.

With only write and read operations on variables stored in replica servers. We meet the first property by allowing a server to return its local or the oldest variable value among several replica servers to a client's read request. To meet the second property, we enforce the write operations to be totally ordered, so that concurrent writes will not result to inconsistent states between different servers.

A client connects to a server via TCP socket, and it sends out requests using three kinds of messages: 1) 'put' to write a variable with desired values; 2) 'get' to read a variable's value; 3) 'dump' to instruct a connected server to dump the value of all variables. Also, a user can input 'delay' command to freeze the client process for a certain amount of time so that is can accepts multiple 'get', 'put' and 'dump' commands.

Upon start of server program with Eventual Consistency model, a server takes arguments of W and R that specify the number of ack messages that are needed to indicate a write/read operation is completed.

When a server receives a 'put' request, it total ordering multicasts a 'w' message to all servers including the sender itself. When it total ordering deliver the message, it will check to see if the total amount of w_ack >= W and this message has been replied to client yet, if both are true, then the server will send a message to client to indicate its request is finished. If not, then it will wait for other w_ack messages to come. A server received 'w' message will unicast a 'w_ack' message to the sender of 'w' message.

When a server receives a 'get' request, it B-multicasts a 'r' message to all servers. Then when the server itself receives 'r' message, it will check to see if the total amount of r_ack >= R and this message has been replied to client yet, if both are true, it will send a message with the latest variable's value to client to indicate its request is finished. If not, the server will wait for other r_ack messages. A server received 'r' message will unicast a 'r_ack' message to the sender of 'r' message.

When a server receives a 'r_ack' message, if the 'r_ack' carries a variable's value with older timestamp, the server will update its local variable. It will also check to see if total amount of r_ack >= R. If true, then the server will send out a message with the latest variable's value to client.

Both of deliver and finish of a request by a server will incur a print to a local log file at this server.

When a server receives a 'dump' request, it will simply print out all values of its stored variables to the screen.

2. Linearizability Consistency:

Linearizability Consistency model has permutation of an operation that is per-process order-preserving, valid, and real-time order-preserving. Algorithm3 satisfies linearizability:

When operation Write(X,v) is invoked at process pi:

(step w5) pi performs a totally-orderd multicast of Write(X, v).
(step w6) When totally-order multicast message Write(X,v) is delivered to process pi, write value v to the local copy of X at pi.
(step w7) Return Ack() indicating completion of the Write(X,v) operation

When process pi is delivered totally-ordered multicast message Write(Y, w) for multicast performed by some other process pj:
(step w8) Write value w to the local copy of Y at pi.

When process pi invokes operation Read(X):
(step r3) pi performs a totally-orderd multicast of Read(X).
(step r4) When totally-ordered multicast of Read(X) initiated by pi is delivered to process pi, read the value of local copy of X at pi; suppose that the value read is v.
(step r5) Return Ack(X,v) to pi as the response to Read(X), which also indicates the completion of the Read(X) operation.

When process pi is delivered totally-ordered multicast of Read(Y) initiated by some other process pj:
(step r6) No action is needed. Discard the message.

We implemented Linearizability Consistency model using Alogrithm3.

When a server receives a ‘put’ request, it total ordering multicasts a ‘w’ message to all servers including the sender itself. When it total ordering deliver the message, the server will send a message to client to indicate its request is finished.

When a server receives a ‘get’ request, it total ordering multicasts a ‘r’ message to all servers including the sender itself. When it total ordering deliver the message, the server will send a message with the variable’s value to client to indicate its request is finished.

Implementation of Server and Client:

1. Server Implementation:
When start a server program, it launches a thread for UDP socket to communicate to other servers, a thread for TCP socket for each connected client. A new thread is created to handle messages coming from either TCP or UDP socket.

2. Client Implementation:
When start a client program, it firstly tries to connect to a specified server with IP and Port. If this server is not available, then it will go through servers listed in config file until an available server is found and re-connect to it. When a connected server is crashed, the client process will abort a new command input by user, then it will go through servers listed in config file until an available server is found and re-connect to this server.

Input and Output Files:

Input Files:
Same ‘config’ file is used as in MP1. This ‘config’ files takes format:

min_delay max_delay
server1 IP1 Port1
server2 IP2 Port2
…

Each server will take a serverID argument when user run the program, after which the server will extract its corresponding IP and Port from the config file. Each client will also be given a serverID that it will connect to and send request, and the corresponding IP and Port from config will also be retrived. In total we specified 10 servers in the config, and each server is able to take requests from multiple clients that connect to this server via the same IP and Port. To run these servers, there is no need to start all of 10 servers at once. In fact, only server1 is needed to be active when performing consistency models as well as to reply to client’s requests. This is achieved from handling crashed of a server and to re-connect to the next available servers listed in config file.

Output Files:
Log files are generated by each server, which prints information as following:

1. SessionID of the current session. (For the mp, can just use any ID)
2. Integer ID of the client.
3. Request type, which is either "get" or "put".
4. Variable name.
5. Logical timestamp.
6. Either "req" or "resp", which denote request/response.
7. Value (could be empty)

How to Run:

1. Server:
usage: server.py [-h] id consistency W R

positional arguments:
id process id (1-10), default=1
consistency consistency model (eventual/linearizability), default=eventual
W number of w_ack indicates finished, default=1
R number of r_ack indicates finished, default=1

Note that serverID = 1 indicates sequencer

2. Client:
usage: client.py [-h] serverID

positional arguments:

serverID server id, default=1

Note that log file 'output_log*.txt' will be created in the same directory. New log info will be appended to the previous log file, you can delete log files before start the new run.

How to Test:

Test Client Commands:

1. Read a variable:
>> get x

2. Write to a variable:
>> put x 1

3. Dump variable status:
>> dump

4. Delay time in ms and input multiple commands:
>> delay 10000
>> get x
>> put x 1
>> get x
>> dump

Test Eventual Consistency:

Start four servers at different terminals:
>> python server.py 1 eventual 2 2
>> python server.py 2 eventual 2 2
>> python server.py 3 eventual 2 2
>> python server.py 4 eventual 2 2

Start four clients at different terminals:
>> python client.py 2
>> python client.py 2
>> python client.py 2
>> python client.py 2

At client1:
>> put x 1
>> get x

At client2:

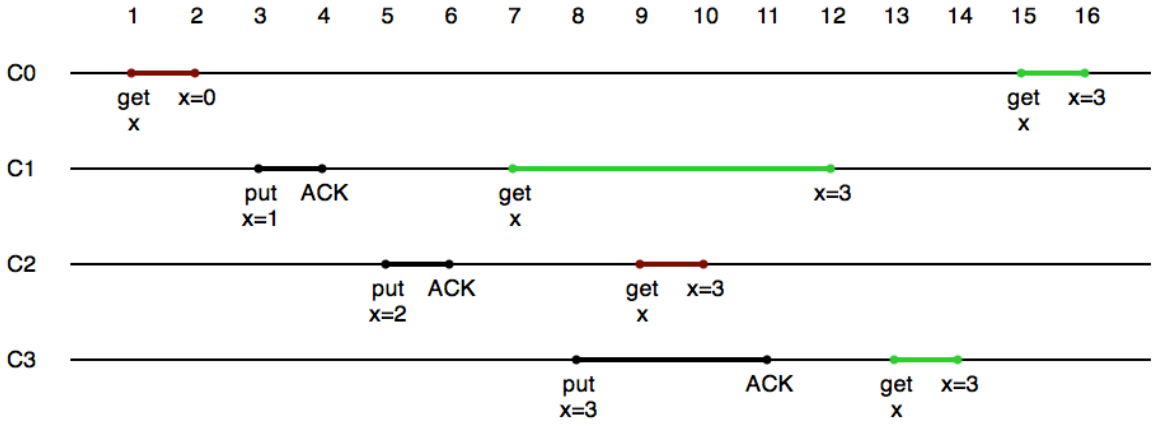
>> get x
>> put x 2
>> get x

At client2:
>> get x
>> put x 3
>> get x

At client3:
>> get x
>> put x 4
>> get x

Output log file at server2 ‘output_log2.txt’:
2,0,get,x,1459795534546,req,
2,0,get,x,1459795537608,resp,0
2,1,put,x,1459795539762,req,1
2,1,put,x,1459795541307,resp,1
2,2,put,x,1459795542188,req,2
2,2,put,x,1459795543419,resp,2
2,1,get,x,1459795547442,req,
2,3,put,x,1459795547497,req,3
2,2,get,x,1459795548847,req,
2,2,get,x,1459795548849,resp,3
2,3,put,x,1459795548972,resp,3
2,1,get,x,1459795553286,resp,3
2,3,get,x,1459795554637,req,
2,3,get,x,1459795554638,resp,3
2,0,get,x,1459795556552,req,
2,0,get,x,1459795559929,resp,3

Plot from visualization tool:



Test Linearizability Consistency:

Start four servers at different terminals:
>> python server.py 1 linearizaibility 2 2
>> python server.py 2 linearizaibility 2 2
>> python server.py 3 linearizaibility 2 2
>> python server.py 4 linearizaibility 2 2

Start four clients at different terminals:
>> python client.py 2
>> python client.py 2
>> python client.py 2
>> python client.py 2

At client1:
>> put x 1
>> get x

At client2:
>> get x
>> put x 2
>> get x

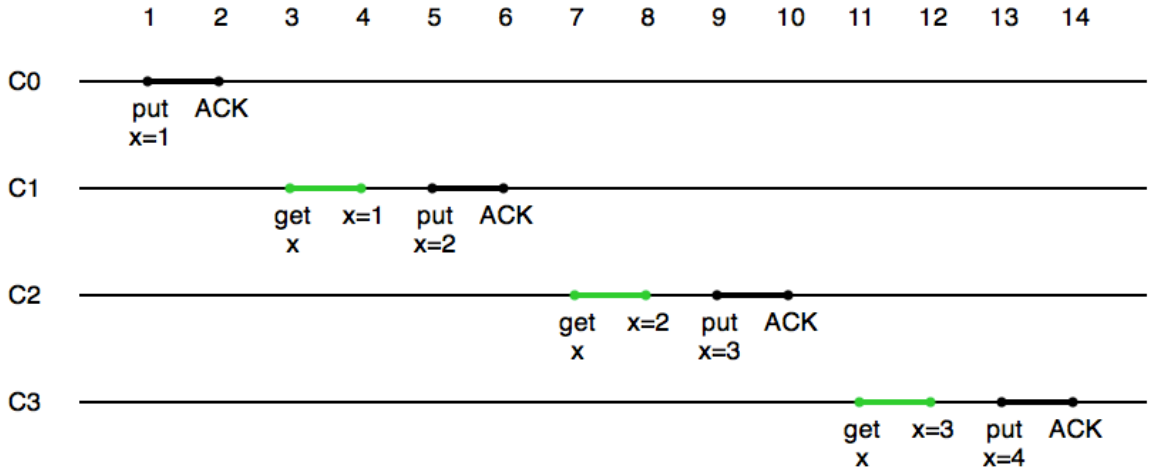
At client2:
>> get x
>> put x 3
>> get x

At client3:
>> get x
>> put x 4
>> get x

Output log file at server2 ‘output_log2.txt’:
2,0,put,x,1459796099675,req,1
2,0,put,x,1459796099677,resp,1
2,1,get,x,1459796105217,req,
2,1,get,x,1459796105218,resp,1
2,1,put,x,1459796105219,req,2
2,1,put,x,1459796105221,resp,2
2,2,get,x,1459796105958,req,
2,2,get,x,1459796105960,resp,2
2,2,put,x,1459796108561,req,3
2,2,put,x,1459796108562,resp,3
2,3,get,x,1459796109783,req,
2,3,get,x,1459796109785,resp,3

2,3,put,x,1459796114127,req,4
2,3,put,x,1459796114129,resp,4

Plot from visualization tool:



Division of Labors:
Funing Xu:
Responsible for developing modules: Unicast, Total Ordering Multicast, Read from config files, and Message format. Also responsible for coding style and consistency model dessign and concurrency control with shared variables.

Wei Yang:
Responsible for developing modules: Eventual Consistency, Linearizabilitiy Consistency, Server program, and Client program. Also responsible for report writing.