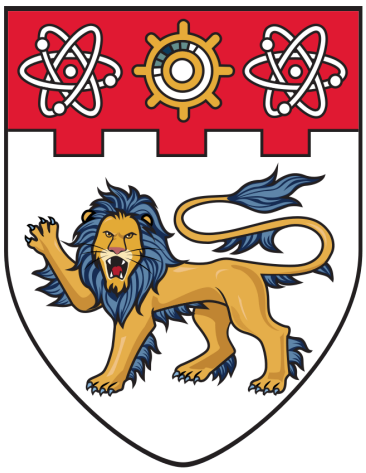

NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

CZ4013 Distributed System Assignment

Zhang Yongzhe	Yang Yubei
Client: Service1,2,3,4,5,6	Server: Service1,2,3,5,6
Server: Service4	Design UDP Communication
Design Message Structure	
Implement Invocation Semantics	
Conduct fault-tolerance Experiments	

Table 1: Contribution

SCHOOL OF COMPUTERSCIENCE AND ENGINEERING

2021

Table of Contents

Chapter 1 Introduction

This project consolidates the basic knowledge about interprocess communication and remote invocation by constructing client and server programs that use UDP as the transport protocol. A distributed facility booking system based on client-server architecture is implemented. The server program stores the information of facilities and booking information, and provides different services, including query, book, change booking,cancel booking and monitoring, which can be remote access by clients. The client program provides an interface for users to invoke these services.

1.1 Environment

The following are the necessary environment to run the system. Operating System: The facility booking system is developed and tested in a GNU/Linux environment. Programming Language: Java.

1.2 Assumption

Several assumptions have been made in developing the system as follows:

1. General Assumptions:
    - Client-Server Communication: All clients are assumed to know IP address and port number.
    - Request Concurrency: Server and client handle the user input in sequential order.
2. Client Assumptions:
    - User Interface: The user interaction to the system is solely done in the command-line interface.
    - User Input: There are minimum error checking (e.g. Variable Type checking).
3. Facility Assumptions:
    - Four different facilities of 2 types are provided: LT1, LT2, MR1, MR2.
    - Facility Timetable: (8:00am - 6:00pm, interval: 1h, total: 10 intervals)
4. Server Assumptions:
    - Server1 Query
        - A user can only query availability of a service within 7 days.
    - Server2 Booking
        - A user can only book a facility one day advance. (e.g. Today is Apr 1st. Can only book from Apr 2nd.)
        - A user can book a facility either 1 hour or 2 hours each time.
    - Server3 Change Booking Time
        - A user can only change booking time to other slots of the same day.

- Server5 Auto-Booking
  - A user can only choose the type of facility.
  - Server will return the nearest available slots in 2 facilities of the required type.
  - If there are no available slots for the next day in both 2 facilities, the booking is unsuccessful.
- Server6 Cancel Booking
  - Once the booking is canceled, it cannot be restored.

Chapter 2 Design

2.1 Architecture Design

We applied ECB design pattern on the structure of our program. Specifically, the client and server are classified in to 3 layers, which are entity, control and boundary.

2.2 Communication Design

2.2.1 Message Format

On the client side, we designed two types of messages, which are Request and Acknowledgement. And the request message is divided into 4 sections: message type, Message ID, Service ID, data shown in Table 2.1. And the usage of different sections are described in the below:

1. Message Type: An integer (0 or 1), which is used by the server to differentiate a message is a Request or Acknowledgement. If the value is 1, it indicates as a request message.

2. Message ID: An integer, which is used by the server to filter duplicated request messages by checking a hash table storing previous processed requests, when the At-Most-Once invocation is enabled.

3. Service ID: An integer, which is used by the server to dispatch the request to a corresponding service handler.

4. Data: See the Common Data Representation.

Msg Type	Msg ID	Service ID	Data

Table 2.1: Request Message

The Acknowledgement is divided into 2 sections: message type and status as shown in Table 2.2. And the usage of different sections are described in the below:

1. Message type: An integer (0 or 1), which is used by the server to differentiate a message is a Request or Acknowledgement. If the value of the message type is 0, it indicates as an acknowledgement message.

2. Status: An integer (0 or 1), which is used by the client to acknowledge whether it receives the response from server successfully (1) or not (0) within a timeout period (controlled by Param MAXTIMEOUTCOUNT and UDPTIMEOUT).

Msg Type	Msg ID	Status

Table 2.2: Acknowledgement Message

On the server side, to improve the efficiency and better use of available channel bandwidth, we piggybacked the acknowledgement on the response message. And the response message is divided into 3 sections: ACK Status, Processed Status, Data as shown in Table 2.3. And the usage of different sections are described in the below:

1. ACK status: An integer (0 or 1), the piggybacked acknowledgement is used to indicate whether the server receives the request from client successfully (1) or not (0).
2. Processed status: An integer (0 or 1), which is used by client to tell whether the request is processed by server successfully(1) or not(0). Client will invoke different functions to display correct or error messages.
3. Data: See the Common Data Representation.

ACK Status(Piggyback)	Processed Status	Data

Table 2.3: Response Message

2.2.2 Marshal / Unmarshal

1. Primitive Type: int. Marshalling an integer into a byte array of size 4. To obtain the *i*th byte in the array, we can right shift the int by 8 * (4 - i) number of times. Unmarshalling the byte array back to an int can be left shifted the *i*th byte in the array by 8 * (4 - i) number of times, and then perform OR operation.
2. Non-primitive Type: String. Marshalling/unmarshalling a String depends on the number of characters. Each byte represents one character of a String.

2.2.3 The Common Data Representation

1. The Big-Endian ordering is used when we marshal/unmarshal data.
2. We assumed that the client and server have common knowledge of the order and types of the variables in the data section.
3. A data session contains different variables depends on the service ID. We inserted the length of a variable followed by that variable shown in Table 2.4.

len1	var1	len2	var2	...

Table 2.4: Data Section

Chapter 3 Services

3.1 Service1: Query Facility Availability

The request and response message has the following format shown in Table 3.1.
This service allows users to query the availability of a facility in the next 7 days. Client needs to provide facility name(e.g. LT1) and number of days he wants to query. Service1 will return a response with available intervals of the required facility within required days. Requests will always be processed correctly, so there is no fail case for this service and response status = 1.

Service1: Query Facility Availability				
Request		ReadIn Syntax Check	**Response**	
Facility Name	String	Facility name exists.	**Available intervals**	String
# of Days	Int [1,7]	Number of days in range [1-7].		

Table 3.1: Service1

3.2 Service2: Book a facility

The request and response message has the following format shown in Table 3.2.
This service allows users to book a facility. Client needs to provide facility name, date to book, slot starting time and ending time. Service2 will return booking information if there is no collision, or return error information if there is collision. Specifically, in success case(1), server response status =1. In the unsuccessful case(2,3,4), server response status = 0. A BookingInfo String contains the following information. (e.g."01-20210322-LT1-1113-1511")

Service2: Book a facility					
Request		ReadIn Syntax Check	**Response**		Case Comment
Facility Name	String	Facility name exists.	**BookingInfo**	String	(1) Success
Date	String	Date in correct String format. Date in range [next day to one week].	**Complete collision**	String	(2) Unsuccess
Start time	int	In range [8-18]	**Partial Collision(1)**	String	(3) Unsuccess
End time	int	In range [8-18]	**Partial Collision(2)**	String	(4) Unsuccess

Table 3.2: Service2

3.3 Service3: Change booking time

The request and response message has the following format shown in Table 3.3.
This service allows users to change the previously booked facility to another time slot within the same day. Client needs to provide BookingID and the offset he wants to change. Service3

will check if the BookingID exists, the offset makes timeslot out of 8am-6pm bound, or the intended change slot has collision with other booked slots. If all requirements are met, Service3 will return a new BookingInfo. Else, corresponding error messages are returned. Specifically, in success case(1), server response status =1. In the unsuccessful case(2,3,4), server response status = 0.

Service3: Change booking time					
Request		ReadIn Syntax Check	**Response**		Case Commen
Booking ID	int	Positive Int	**BookingInfo**	String	(1) Success
Offset of change	int	Positive/ Negative Int	**BookingID not found**	String	(2) Unsuccess
			Offset outof bound	String	(3) Unsuccess
			Change has collision	String	(4) Unsuccess

Table 3.3: Service3

3.4 Service4: Monitor

The request and response message has the following format shown in Table 3.4.
This service allows users to monitor the availability of a facility over the week. Users can indicate the facility and duration that they want to monitor on the client side. The server will register each client when receive the monitoring requests. And then, when a facility has a new booking, change or cancel, the server will notify the clients which are registering under this facility by using callback mechanism.

Service4: Monitor				
Request		ReadIn Syntax Check	**Response**	
Facility Name	String	Facility name exists.	**Available intervals in 7 days**	String
Duration(in second)	Int	Positive Int		

Table 3.4: Service4

3.5 Service5: Auto booking facility

The request and response message has the following format shown in Table 3.5.
This service will auto select a facility with one hour which is the most recent available for the user. Client needs to provide facility type. Service5 will check if there are any left available slots of the required facility type on the next day. If there is, return the most recent available slot of 2 facilities. Else, return an error message. Specifically, in success case(1), server response status =1. In the unsuccessful case(2), server response status = 0.

Service5(Non-idempotent): Auto booking facility					
Request		Comment	**Response**		Case Comment
Facility Type	int	1: Lecture theater 2: Meeting room	**BookingInfo**	String	(1)Success
			No Available Slot	String	(2)Unsuccess

Table 3.5: Service5

3.6 Service6: Cancel Booking

The request and response message has the following format shown in Table 3.6.
This service allows users to cancel a previous booking. Client needs to provide the Booking ID which is received in Service2 or Service3. Service6 will check if this Booking ID exists. If it exists, return a successful cancellation message to the client. Else, return an error message. Specifically, in success case(1), server response status =1. In the unsuccessful case(2), server response status = 0.

Service6(Idempotent): Cancel Booking				
Request		**Response**		Case Comment
Booking ID	int	**ChangeInfo**	String	(1)Success
		BookingID not found	String	(2)Unsuccess

Table 3.6: Service6

Chapter 4 Fault Tolerance

4.1 Design Ideas

The UDP transport protocol is naively designed for low latency but not fault-tolerant transmission. UDP has no handshaking mechanism to guarantee reliable connections between clients and servers as compared to TCP. We implemented some techniques such as time-out, retransmit requests/responses, maintain histories and filter duplicated requests to guarantee the system to be fault-tolerant to message loss. Firstly, we design a three-way handshaking communication between servers and clients. When the server receives the response from clients, it processes the request and sends a response message piggybacking ACK to the client. At last, clients will send a ACK message to acknowledge whether it received the response successfully or not.

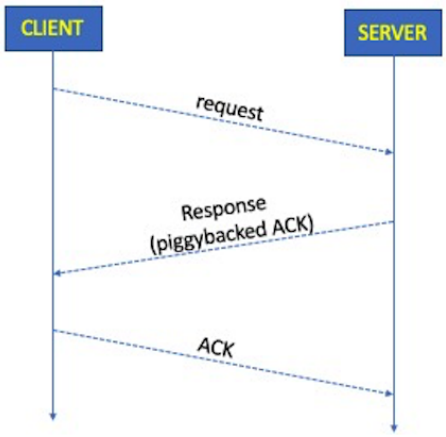

Figure 4.1: Design

We considered three scenarios of message loss during transmission between server and client.

4.1.1 Scenario 1: Request Loss

This scenario is analyzed based on the assumption of Response and Acknowledgement are never lost. As shown in Figure 4.2, to detect the request loss, we implemented a timeout mechanism on the client side. If a client doesn't receive any replies from the server after a certain time, the client will send an acknowledgement message with status 0 (NAK). When the server receives a NAK message from the client, it will send a piggybacked NAK back to the client, So, the client will resend the request. This process keeps repeating until the client receives a piggybacked ACK message from the server.

4.1.2 Scenario 2: Response Loss

This scenario is analyzed based on the assumption of Request and Acknowledgement are never lost. We also used the timeout mechanism on the client side to detect the response message loss during transmission. As shown in Figure 4.3, the differences with the scenario 1 is after the server (if enable At-Most-Once) receives a NAK message from client, the

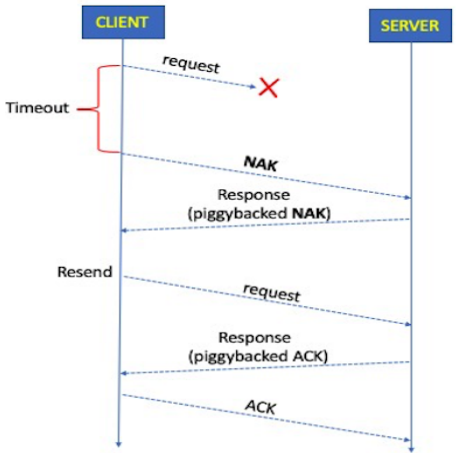

Figure 4.2: Request Lost

server will first get the message ID from the NAK message (more detailed in the Message Structure) , and then search in a maintained Hashtable with the messageID to check whether a previous processed request with the same message ID can be found. If found, the server just resend the response message with piggybacking ACK. If not found, the server will just send a NAK back to the client (At-Least-Once just simply sends a NAK without searching in the Hashtable).

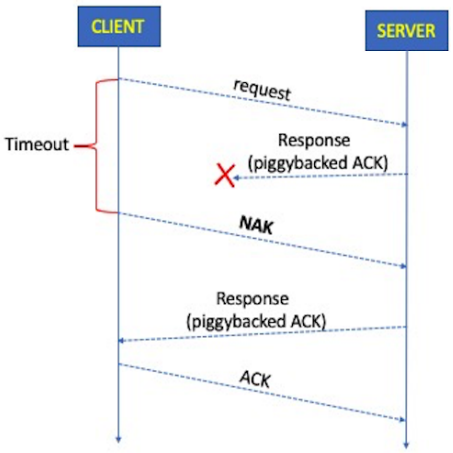

Figure 4.3: Response Lost

4.1.3 Senario 3: Acknowledgment Loss

This scenario is analyzed based on the assumption of response and request are never lost. As shown in Figure 4.4, The purpose of the Acknowledgement is to acknowledge the client has successfully received the reply from the server or not. After receiving an ACK from client, server can remove the previous record in the Hashtable(if enable At-Most-Once). So, if the acknowledgment message is lost, the Hashtable storing previous processed records will never be removed. So, our proposed solution for this issue is to implement a timeout mechanism on the server side to auto remove a previous record within a certain time to avoid the Hashtable keeps growing. But, from the experience, the loss of an acknowledgement message won't cause any operations to produce unexpected results.

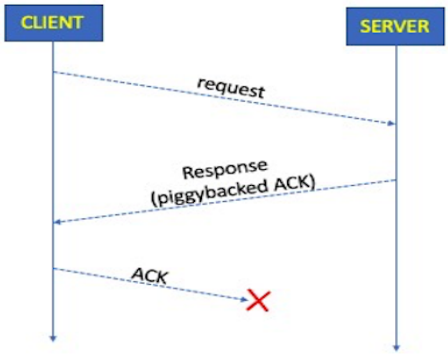

Figure 4.4: Acknowledge Lost

4.2 Experiments

4.2.1 Non-Idempotent

We use our Service 5 (Auto-booking function) to conduct the three message lost scenarios with two different invocation semantics under the failure rate is 50%. As shown in Table 4.1, we verified that the At-Least-Once semantics will invoke multiple times of bookings with randomly picking up the latest available slots on the server side under the response-loss scenario only.

	At-Least-Once	At-Most-Once
Request loss	Auto select one latest available slots	Auto select one latest available slots
Response loss	(**Unexpected**)Auto select **multiple** latest available slots	Auto select one latest available slots
Acknowledgement loss	Auto select one latest available slots	Auto select one latest available slots

Table 4.1: Non-Idempotent

4.2.2 Idempotent

We use our Service 6 (cancel bookings) to conduct the three message lost scenarios with two different invocation semantics under the failure rate is 50%. As shown in Table 4.2, We verified that all the scenarios can produce correct operational results regarding of the invocation semantics.

	At-Least-Once	At-Most-Once
Request loss	Booking is cancelled successfully	Booking is cancelled successfully
Response loss	Booking is cancelled successfully(server) (Booking ID is not found,client)	Booking is cancelled successfully
Acknowledgement loss	Booking is cancelled successfully	Booking is cancelled successfully

Table 4.2: Idempotent

Chapter 5 Conclusion

We designed and implemented a distributed booking system fault-tolerating to the message loss during transmission with UDP protocol. We analysed and experimented three different message loss scenarios and implemented several mechanisms including timeout, re-transmit request/response messages, filter duplicated requests. Finally, we compared two different invocation semantics under three different message loss scenarios and found that the non-idempotent operations could produce wrong results when applied At-Least-Once semantic under the response-loss scenario.