CONCORDIA UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

COMP 6231 - SUMMER 2019 Instructor and Coordinator: Sukhjinder K. Narula, R. Jayakumar

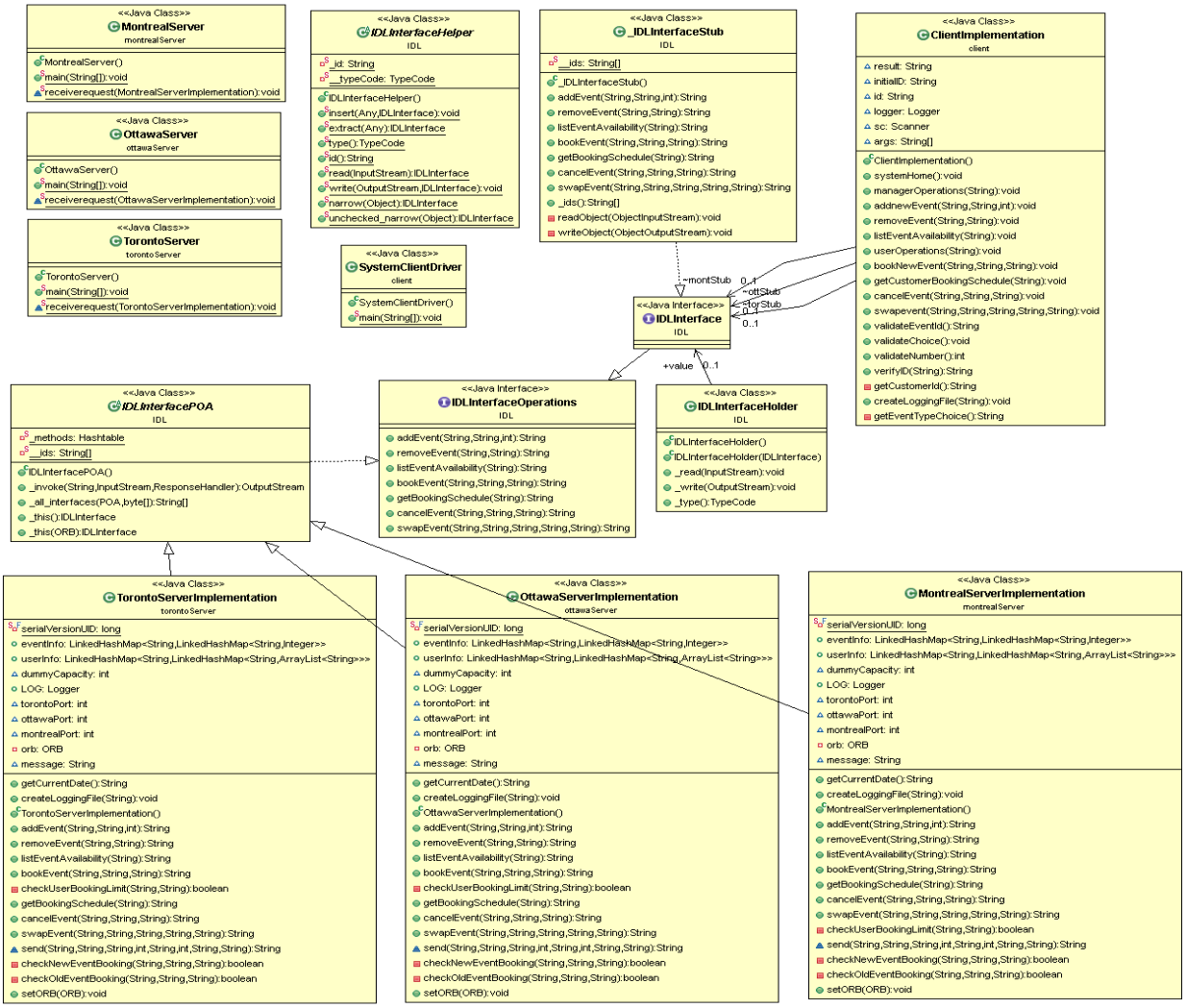
ASSIGNMENT 2

Student Id – 40093384 Name- Yogesh Nimbhorkar

Requirement: In this Assignment, we have been asked to implement a Distributed Event Management System with the help of the CORBA which is the technology widely used today in industries to implement distributed systems. In addition, we are required to add a new method which is Swap Events. This method will be used by the users to exchange the already booked events with different events in the system.

ARCHITECTURE

Class Diagram



We have three servers: Montreal server, Toronto server and Ottawa server. I have designed an IDL interface which consists of all the methods of customer and manager. This interface is then compiled using IDL compiler to generate stubs and skeletons. Client use this information to know the method structure and definition. In turn, Server implements the methods of this interface.

CLIENT

I have structured the Customer Side in the client package which contains the SystemClientDriver class and ClientImplementation class. Client class founds facilitated naming services / registries for all the three servers for various systems i.e Montreal server, Toronto server and Ottawa server and likewise look for the binded objects of the servers from their separate registries to pronounce the stubs on the customer side thus characterizing the way to make RPC utilizing CORBA.

SERVER

In this case, I have designed six server classes for all the three servers respectively.

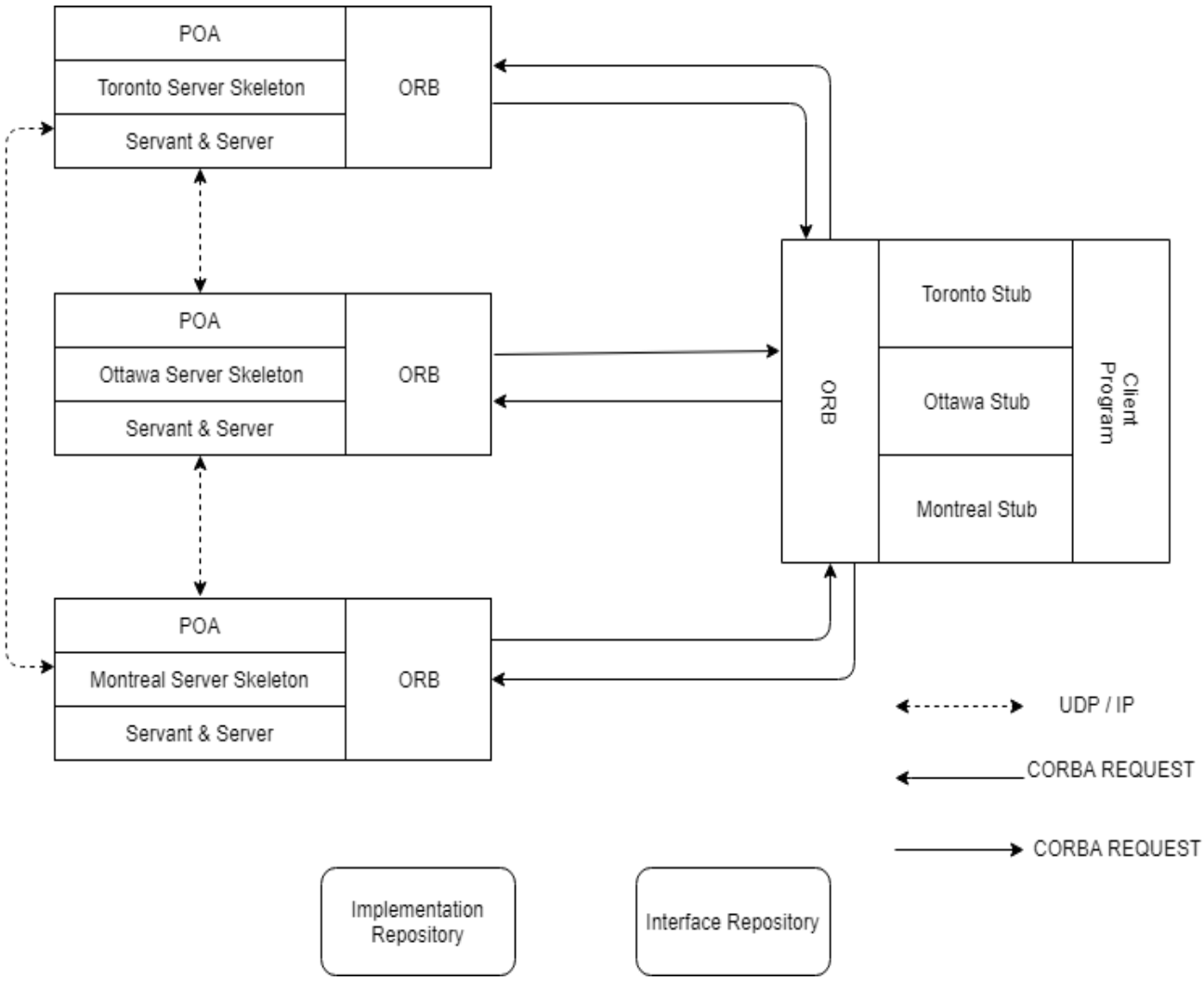
- MontrealServer.java
- MontrealServerImplementation.java
- OttawaServer.java
- OttawaServerImplementation.java
- TorontoServer.java
- TorontoServerImplementation.java

All the three server classes implement an interface IDLInterface.idl and the implementation classes for all the three servers as MontrealServerImplementation.java , OttawaServerImplementation.java and TorontoServerImplementation.java and the functionality is common for all the three servers as they all have similar capabilities. These servers can also communicate with each other using UDP and exchange messages.

Servant: 3 implementations classes are the servant which implements the IDLInterface.idl Interface. Each server object is an instantiation of this classes.

Server: Each server class is responsible for initializing the ORB and registering the reference of the servant object and binding it with the naming service using rebind () method. It also supports UDP communication which is required to communicate between different servers thus enabling server to server communication possible and it is relatively faster.

Below is the architectural diagram of my application –



This diagram illustrates simple client server architecture using CORBA design. Below are the main points that focuses on this architecture –

1. **POA** is the server-side proxy at each of the three servers.

2. All communications between client and server happens using Language independent Requests and replies which is only through ORB. All requests at client-side replies from server side are sent through ORB.

3. All messages exchanged between different servers using UDP/IP protocol whereas CORBA is responsible for communication between client to server.

4. Server and clients use a common Interface and implementation repository.

5. Naming service is used for the servers to store the servant references and client to access those references.

The following classes are generated after compiling IDL Interface- _IDLInterfaceStub.java, IDLInterface.java, IDLInterfaceHelper.java, IDLInterfaceHolder.java, IDLInterfaceOperations.java and IDLInterfacePOA.java.

1. _IDLInterfaceStub.java- Client-side proxy also known as stub

2. IDLInterfaceHolder.java- Holds a reference to an object that implements IDL Interface

3. IDLInterface.java- It is the java version of IDL interface.

4. IDLInterfacePOA.java- The Skeleton also known as server-side proxy combined with POA (Portable object adapter)

5. IDLInterfaceHelper.java- It provides the functionality to support a CORBA object

6. IDLInterfaceOperations.java- It contains the methods of IDL interface generated in JAVA language.

There are seven methods of user and manager that have been implemented. The significance of each method is mentioned below –

Manager Role:-

Add Event –

When an event manager invokes this method through the server associated with this event manager (determined by the unique eventManagerID prefix), attempts to add an event with the information passed, and inserts the record at the appropriate location in the hash map. The server returns information to the event manager whether the operation was successful or not and both the server and the client store this information in their logs. If an event already exists for same event type, the event manager can't add it again for the same event type but the new bookingCapacity is updated. If an event does not exist in the database for that event type, then add it.

Remove Event –

When invoked by an event manager, the server associated with that event manager (determined by the unique eventManagerID) searches in the hash map to find and delete the event for the indicated eventType and eventID. Upon success or failure it returns a message to the event manager and the logs are updated with this information.

If an event does not exist, then obviously there is no deletion performed. Just in case that, if an event exists and a client has booked that event, then, delete the event and take the necessary actions.

List Event Availability –
When an event manager invokes this method from his/her branch's city through the associated server, that branch's city server concurrently finds out the number of spaces available for each event in all the servers, for only the given eventType. This requires inter server communication that will be done using UDP/IP sockets and result will be returned to the client. Eg: Seminars - MTLE130519 3, OTWA060519 6, TORM180519 0, MTLE190519 2.

User Role:-

1. **Book Event** –
   When a customer invokes this method from his/her city through the server associated with this customer (determined by the unique customerID prefix) attempts to book the event for the customer and change the capacity left in that event.
2. **Get Schedule** –
   When a customer invokes this method from his/her city's branch through the server associated with this customer, that city's branch server gets all the events booked by the customer and display them on the console.
3. **Cancel Event** –
   When a customer invokes this method from his/her city's branch through the server associated with this customer (determined by the unique customerID prefix) searches the hash map to find the eventID and remove the event.

4. **Swap Event** –
   A customer might invoke this operation to change an event(belonging to an event type) he/ she has already booked. In this case, the current_city_branch_server (which receives the request from the customer) first checks whether the customer has booked the old event, then checks with the new_city_branch_server (on which the new event has to be booked) whether there is available capacity for the new event, and if both checks are successful then atomically book the customer for the new event and cancel the old event for the customer.

Data Structures: -

1. **Evnet Information HashMap** – This is a nested hashmap which stores the information of an event in the system. It contains key as Event Type and its value as a hashmap which in turn stores Event Id as a key and capacity as the value.

```
public HashMap<String, HashMap<String, Integer>> eventInfo = new HashMap<String, HashMap<String, Integer>>>();
```

2. **User Information HashMap** – This map contains user ID as the key and sub hashmap of Event Type and array list of event ids as values.

```
public HashMap<String, HashMap<String, ArrayList<String>> userInfo = new public HashMap<String, HashMap<String,, ArrayList<String>>>();
```

TEST SCENARIOS:

Created a test class with 7 threads all trying to perform operations concurrently, three threads are trying to exchange an event and three threads are trying to book event. The last one is trying to print user schedule. All operations are happening concurrently, so they have to be synchronized. The expected output can be random.

Difficult Points from this Assignment: -

1. Designing an application using CORBA and replacing the old RMI syntax with CORBA syntax. Faced few problems on starting naming service so it was bit difficult.

2. Implementing multi-threading to increase the system performance and allow multiple users to allow operations at the same time.

3. Making the Exchange Operation atomic as If either cancel or book fails, the other operation also should not happen.

References: -

- http://www.ejbtutorial.com/corba/tutorial-for-corba-hello-world-using-java
- https://www.geeksforgeeks.org/multithreading-in-java
- https://docs.oracle.com/javase/7/docs/technotes/guides/idl/corba.html
- https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture