ASSIGNMENT 2
Using CORBA

Project Test Cases to check concurrency:

Before going into the Architecture, below are the server performances I got :

Written a test case to create 10,000 rooms from AdminClient, worked very well without any memory issues, and log file was also updated successfully 10,000 times.

And I also have tested with 100,000 requests it went fine without any errors in 40-45 seconds. But get available time slots had taken more time as it includes UDP connections with other campuses

Type of operation	Time Taken	Number of Requests
createRoom	7 seconds	10,000
createRoom	40-45 seconds	100,000
getAvailableTimeSlot	17 - 19 seconds(Due to two extra UDP connections for every request)	10,000
changeReservation	11 - 12 seconds(as each request may need an extra UDP connection)	10,000

Note: in above test cases requests are sent concurrently using threads.

CORBA:

CORBA(Common Object Request Broker), is the OMG's specifications to support distributed computing between different programming languages on different platforms. So, different middleware developed by different vendors can interoperate.

How CORBA supports language-neutral communication:

Using CORBA a program from any vendor, any computer, any operating system, any programming language, and network can interoperate with a CORBA programs. We achieve interoperability between different CORBA systems by following

Language neutral IDL:

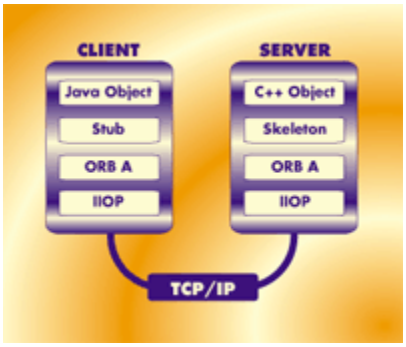
Every server object(s) declares its features through an interface file to the client, in CORBA we write this interface file in a language neutral way as shown below

ASSIGNMENT 2
Using CORBA

```
module admin{
  typedef sequence<string> List;
  interface AdminInterface{
    boolean createRoom(in string roomNumber,in string date,in List list_Of_Time_Slots);
    boolean deleteRoom(in string roomNumber,in string date,in List list_Of_Time_Slots);
    boolean resetCount();
  };
};
```

By using an IDL complier we can convert the above IDL file into a language specific interface file, along with language specific interface file IDL compilers also generate Adapter and helper classes in order to support communication between ORB and Object implementations.

Moreover, GIOP ensures interoperability of different vendors ORB's. The GIOP specification outlines the mapping of OMG IDL data types and object references to a common network representation known as CDR (Common Data Representation). CDR is an over-the-wire data format that handles marshaling of data types, byte ordering and other low-level operations to ensure that data is transported in a common standard format so that ORB invocations and information sent between machines on a network mean the same thing to both applications.



(credit: http://www2.sys-con.com/itsg/virtualcd/java/archives/0309/barlotta/index.html)

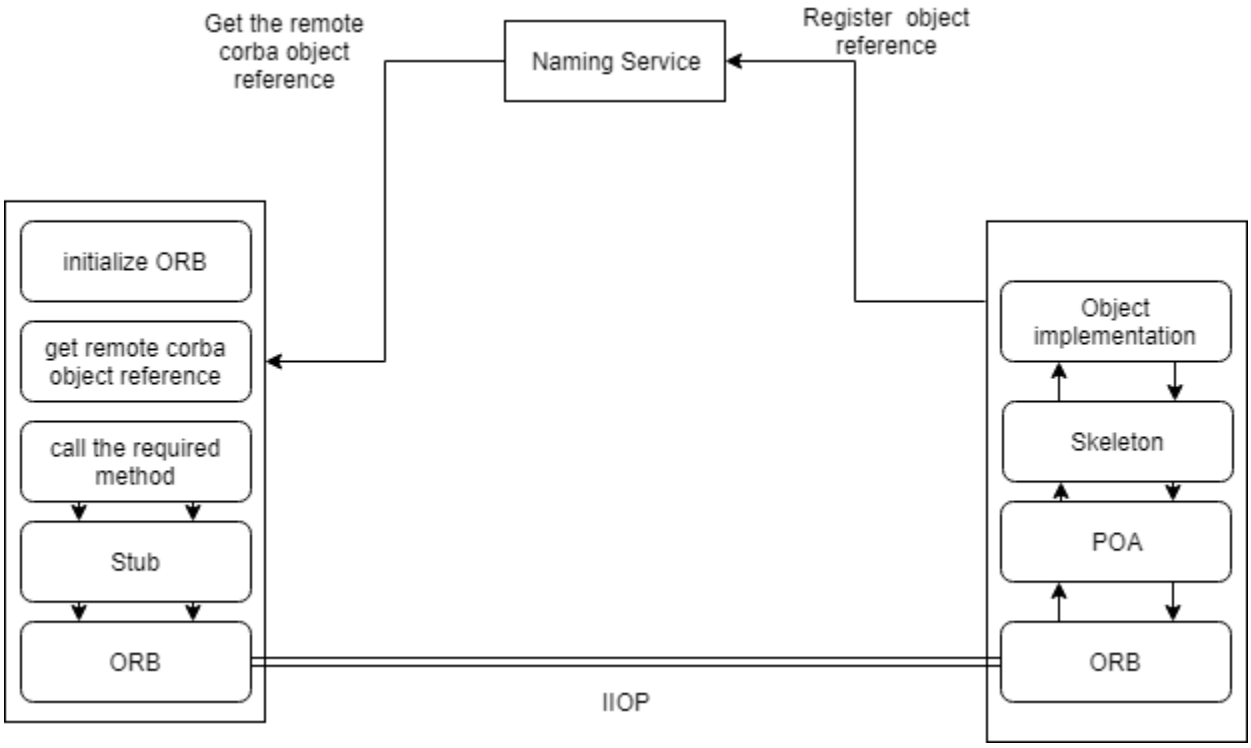
Object reference is used to identify a distributed object , as CORBA is language independent it uses IOR for interoperability.

Project Architecture:

This project runs the naming service orbd on port number 1050, hence we have to initialize the orb with naming service host and port details, as shown below

-ORBInitialPort 1050 -ORBInitialHost localhost

ASSIGNMENT 2
Using CORBA



Below are two important aspects I have taken care in this project:

1. Even though ORB runs every client request on a separate Thread we have to take care about synchronizing the data as our HashMap database is static and has to be shared among the multiple clients at a time. To handle this situation I have used Locks in java instead of synchronized block.

2. Each server in every campus has below programs running concurrently:
   - ORB Server -To respond to Student and Admin requests.
   - UDP Server -To respond to UDP Client requests from other servers.
   - UDP Client -To communicate with other campuses servers.

   To handle the UDP requests I have used the thread pool to reduce the thread creation time.

To achieve High Concurrency:

While designing the project I have consider the concurrency and efficiency as main aim, so in order to achieve them I have used **Locks** , **Thread Pool** and **Local Variables**.

ASSIGNMENT 2
Using CORBA

About ReadWriteLock:

Instead of using synchronized blocks which allows only one thread for read or write operations I have used ReadWriteLock in the project. An RW lock allows concurrent access for read-only operations, while write operations require exclusive access.

Using ReadWriteLock we can allow the multiple threads to read while there is a no write operation. Whenever there is a write operation it blocks all read operations and remaining write operations to prevent data race conditions and deadlocks.

Hence I utilized this methodology to achieve high concurrency in the project.

About Thread pool:

While designing high concurrent UDP server we have to use separate thread to handle a request. Preferring to create Thread-per-Request is good way to deal with multiple client requests but this model does not scale for huge number of requests.

Let consider below code snippet which creates a new thread for every request

```
while(true){
     DatagramPacket request = new DatagramPacket(buffer, buffer.length);
     aSocket.receive(request);
     new Thread(new Runnable() {
          @Override
          public void run() {
               handleRequest(request,aSocket);
          }
     }).start();
}
```

This above snippet worked good for 3000 requests sent from a for loop, but I have tested it with 5000 request where I have faced java.lang.OutOfMemoryError as shown below.

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread
        at java.lang.Thread.start0(Native Method)
        at java.lang.Thread.start(Unknown Source)
        at client.UDPServer.main(UDPServer.java:35)
An unrecoverable stack overflow has occurred.
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  EXCEPTION_STACK_OVERFLOW (0xc00000fd) at pc=0x0000000071ed2177, pid=24752, tid=241516
#
```

ASSIGNMENT 2
Using CORBA

Why OutOfMemoryError: Creating a thread is an expensive task because each thread has to create its own memory and stack. Creating a thread pool saves us from the OutOfMemory errors as some predefined threads are cached and reused from the thread pool. To provide high server availability I have used Thread Pool in Java.

After using the thread pools I have verified with 100,000 requests and it had worked without any errors.

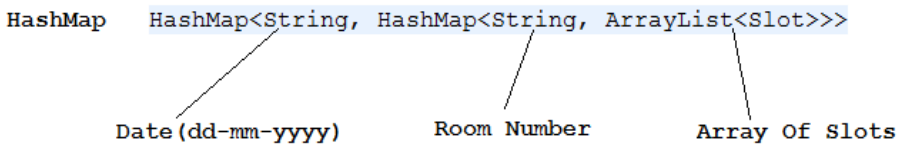
Local Variables:

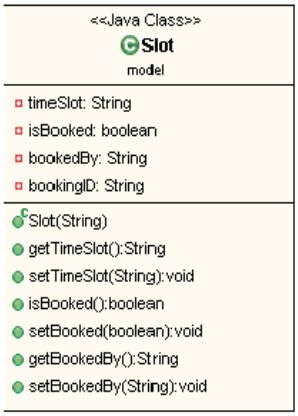
Achieving concurrency through local variables is little bit tricky. As each thread has its own memory space, we can use this space to create local copy of small portion of shared data in the thread so we can release the read lock on the shared data as soon as possible which gives a chance to another thread to read the shared data.

However we should not have to copy huge memory into the local variables which consumes again huge amount of memory . Here the principle is releasing a lock as soon as possible increases efficiency.

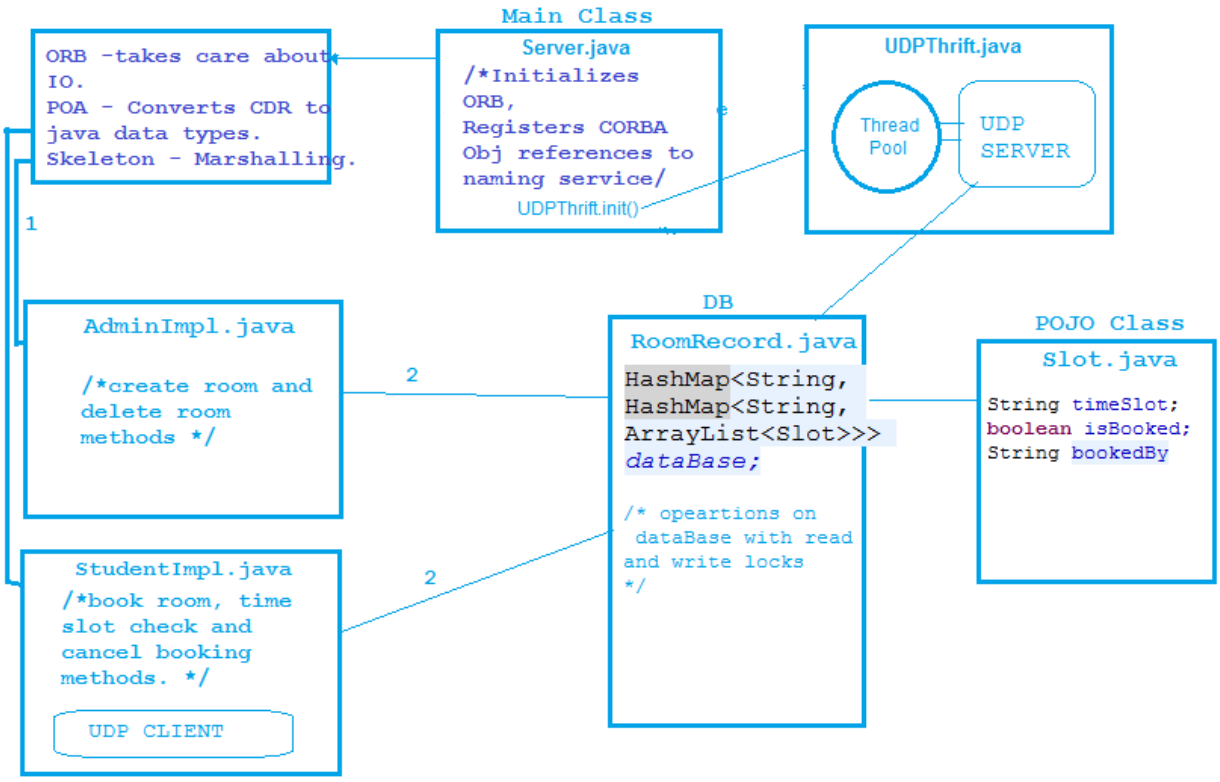
Example: Copy the array of slots in a particular room from shared Hashmap(database) into local variable and remove the lock on shared data this gives high availability of shared data to remaining clients .

Data Structure of DB(HashMap(dataBase)): The sub HashMap key is room number so we can identify rooms on a particular date very easily. A room has Array of Slots where Slot is a POJO class as shown below.





ASSIGNMENT 2
Using CORBA
Server Abstract Architecture



Above Server architecture shows how exactly HashMap(dataBase) is shared among AdminImpl, StudentImpl, UDP Server and Client classes. To overcome data race situations, all DB operations are done in RoomRecord.java which is a final class. So all DB related operations take place in centralized way to ensure that Read and Write locks are done precisely.

Difficult part in the project:

Implementing Thread Pool and Read and Write locks on DB is little bit easy. However, I have faced difficulty in below scenario:

From Server1 we have to communicate with remaining two servers using UDP. So to get data from Server2 and Server3 we send two requests concurrently from Server1, by using two threads at a time, these two threads send the request concurrently and get the result concurrently, I mean in a overlapping time period. Threads work in asynchronous fashion, so to

send data back to student client we have to block the main thread until we get the results from both two servers.

Now the question is how to wait in main thread until both two threads fetch the results from other campuses?.

The answer is CountDownLatch, CountDownLatch initializes with a count and await() method block until the count becomes zero, so we have to count down by using countDown() method in threads.

Concurrency Test Cases:

If four requests are sent concurrently to book a slot then 1st one has to be success and remaining have to fail, below is the code and result to send 4 concurrent request to book a room.

```
for(int i =0;i<4;i++){
    int o = i;
    new Thread(new Runnable() {
        @Override
        public void run() {
            StudentInterface st = getRemotereference("dvl");
            System.out.println(st.bookRoom("dvl", "dvls4567", "34", "23-10-2017", "12:30 - 13:30"));
        }
    }).run();
}
```

```
<terminated> TestSlots [Java Application] C:\Program Files\Java\j
dvl_23-10-2017_34_1
fail
fail
fail
```

ASSIGNMENT 2
Using CORBA

The below test case sends 10,000 requests to get Available time slots, and it went successful.

```
public static void main(String[] args) throws InterruptedException {
    connectToServer(args);
    String[] servers = {"dvl","kkl","wst"};
    for(int i =0;i<10000;i++){
        int o = i;
        new Thread(new Runnable() {
            @Override
            public void run() {
                String randomServer =servers[new Random().nextInt(servers.length)];
                StudentInterface st = getRemotereference(randomServer);
                System.out.println(st.getAvailableTimeSlot("23-10-2017"));
            }
        }).run();
    }

}
```