Distributed System Design

COMP 6231 – Winter 2020

Concordia University

Department of Computer Science and Software Engineering

Instructor: R. Jayakumar

Distributed Event Management System(DEMS) – Assignment 2: Using CORBA

By: Sepehr Jalayer

Concordia ID: 40126236

Table of Contents

Overview

The distributed Event management system (DEMS) consists of three different servers which are located in different cities:

- Montreal(MTL)
- Sherbrooke(SHE)
- Quebec(QUE)

The clients of this system are of two types:

- eventManagers
- customers

We must ensure that these clients are connected to their own servers with Java RMI, and also the connection between our three servers are done through UDP/IP socket programming.

Manager specific functions:

- **addEvent():** managers can only add events in their own server
- **removeEvent():** managers can only remove events from their own server. *if an event was removed we must book another closest event for the customers registered in that event. !!Needs **UDP** for server-server connection.
- **listEventAvailability():** we must gather all events of a given type from all three servers. !!Needs **UDP** for server-server connection.

Client/Manager functions:

- **bookEvent():** customers can also book from other servers with a weekly 3 limit. !!Needs **UDP** for server-server connection.
- **getBookingSchedule():** show the customers booking schedule.
- **cancelEvent():** clients can remove an event from their own schedule. !!Needs **UDP** for server-server connection.
- **swapEvent():** clients can swap a booked event with another event. (a bookEvent + cancelEvent) -> needs to be atomic

Clients are recognized with their ClientID (8 character): serverID (3char) + clientType(C/M) + 4 digit identifier.

Events are recognized with their eventType: Conferences/Seminars/Trade Shows + their eventID(10 character): serverID (3char) + eventSlot (M/A/E) + eventDate (DDMMYY).

**Both servers and client maintain log files stored in the project directory.

Implementation

- Client – Server communication is done by CORBA and orb middle-ware
    - Server is run with these arguments: -ORBInitialPort 1050 -ORBInitialHost localhost
    - Client is run with these arguments: -ORBInitialPort 1050 -ORBInitialHost localhost
    - ORB runs with this command: start orbd -ORBInitialPort 1050
- Server – Server communication is done via UDP/IP Socket programming
    - Montreal UDP port: 8888
    - Quebec UDP port: 7777
    - Sherbrook UDP port: 6666
- To reduce the duplication code and facilitate changes and debugging we used single server implementation file and single interface implementation file.
- Both Server and Client maintain separate logfiles
- Server Log files are located under \project_directory\src\Logs\Server\serverName.txt
- Client Logs are located under \project_directory\src\Logs\Client\ClientID.txt
- We used concurrentHashMaps to store the data, to ensure maximum concurrency.
- We used synchronized blocks and methods in some cases to ensure thread safe operation
- The Most important part of the implementation was avoiding infinite loops in UDP calls specially in removeEvent() and listEventAvailability() methods.
- The Hardest method to implement was the removeEvent() when there were clients registered in the event and we some of them were from other servers.
- For the atomicity of the swap method, we booked the newEvent first (somewhat similar to reservation) then if it was a success -> we canceled our oldEvent. And if the cancel was not successful -> we canceled our formerly booked newEvent (cancelReservation)
- We added a shutdown() method for shutting down the ORB
- Added a test concurrency to client to check how our database is thread-safe or not? We concurrently request 5 book/cancel events to an event with 2 capacity

IDL interface definition

```
module ServerObjectInterfaceApp
{
  interface ServerObjectInterface
    {
        /**
        * Only manager
        */
        string addEvent(in string eventID, in string eventType, in long bookingCapacity);

        string removeEvent(in string eventID, in string eventType);

        string listEventAvailability(in string eventType);

        /**
        * Both manager and Customer
        */
        string bookEvent(in string customerID, in string eventID, in string eventType);

        string getBookingSchedule(in string customerID);

        string cancelEvent(in string customerID, in string eventID, in string eventType);

        string swapEvent(in string customerID, in string newEventID, in string newEventType, in string oldEventID, in string oldEventType);

        oneway void shutdown();
    };
};
```

Figure 1 - ServerObjectInterface.idl

Class Diagram

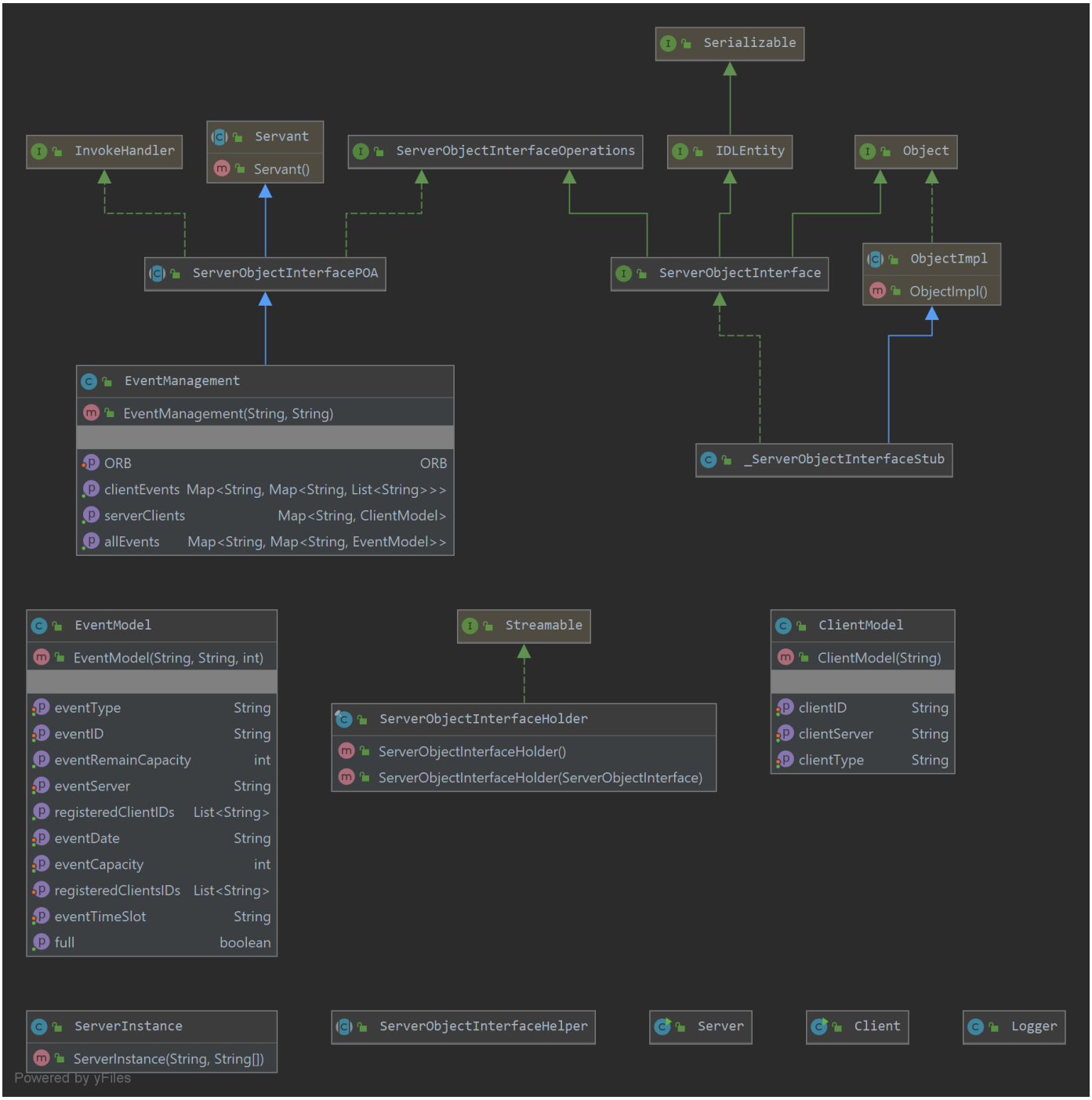


Figure 2 -Class Diagram with properties

**The Full dependencies and methods of each class is shown below.

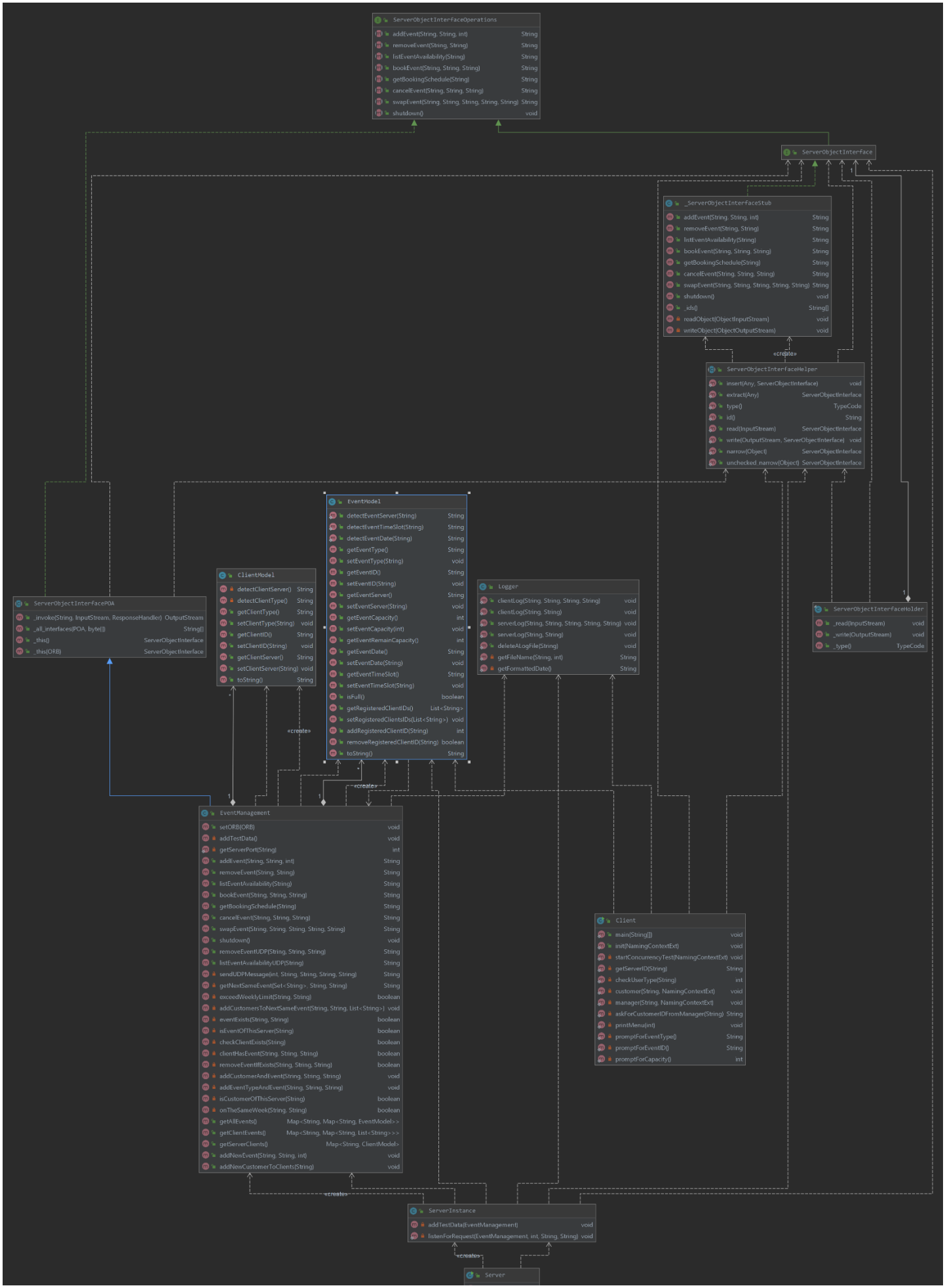
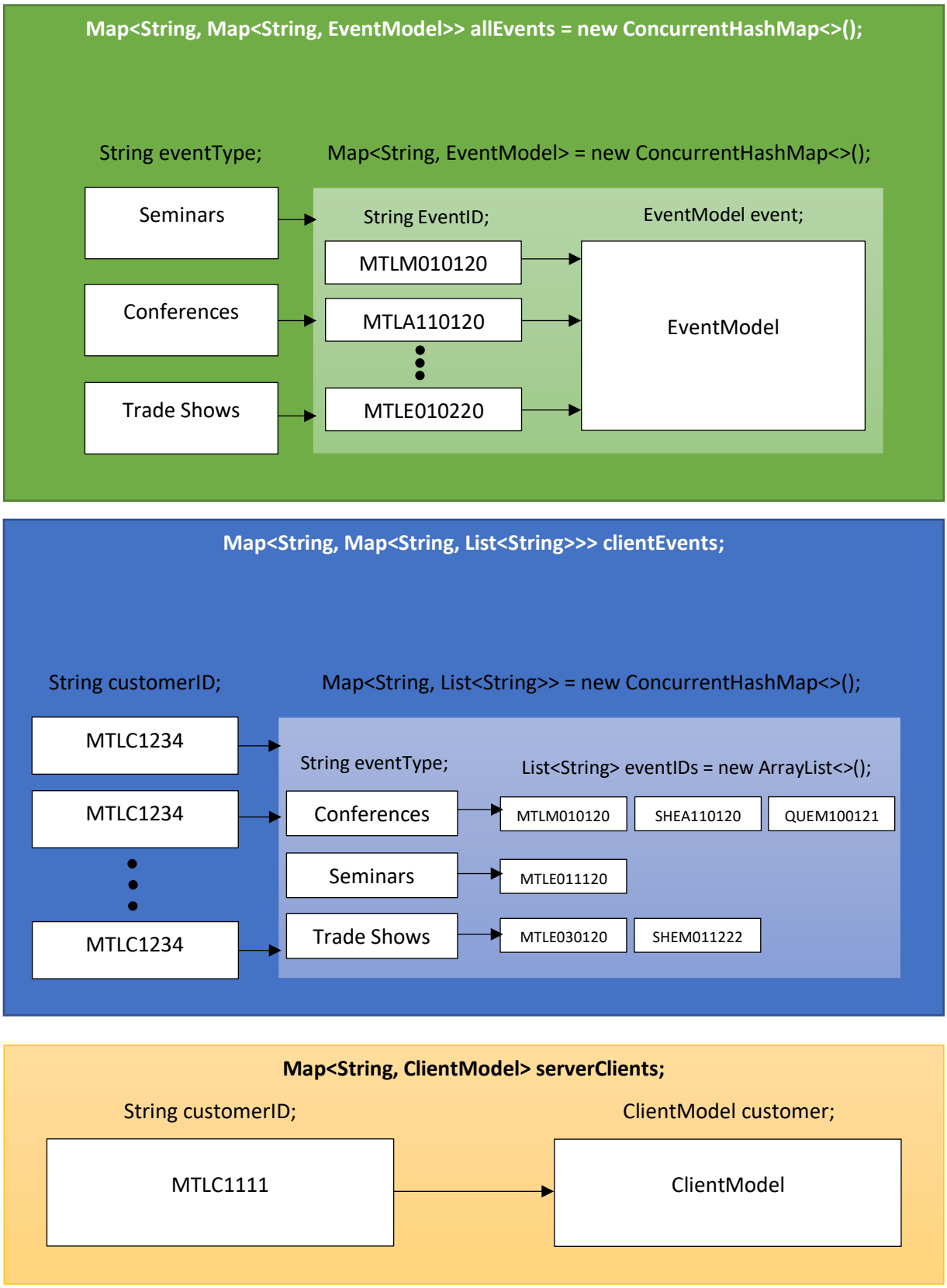


Figure 3 - Class Diagram with full dependencies

Data Structures

All the data is maintained within each server, using three Map structures shown in the figure below.



Test Scenarios

#	Type of Test	Scenario	Cases
1	Login	UserName	1.Event Manager ID 2.Customer ID
2	Menu Items		
3		Logout	1.Log out menu Item
4	Event Manager	addEvent()	1.invalid EventID -> not added 2.new EventID -> added 3.Existing EventID (LowerCapacity) -> not allowed 4.Existing EventID (HigherCapacity) -> capacity Updated 5.Duplicate Event -> not happening 6.EventID of Other Servers -> not allowed
5		removeEvent()	1.invalid EventID 2.EventID not exist 3.Event without anyone registered -> removed event 4.Event with someone registered -> Removed event + registered to same eventType if possible (UDP if needed) 5.EventID in other servers -> not allowed
6		listEventAvailability()	1.list all events of a given type from all three servers (UDP needed) 2.Event type is forced correctly with showing only options available
7		Ask for customerID	1.Access Customer methods
8	Event Manager + Customer	bookEvent()	1.on own server -> allowed 2.if event is full -> not allowed 3.on other servers -> only three in a week (UDP needed) 4. invalid EventID -> not allowed
9		getBookingSchedule()	1.Show booking schedule of customer 2.invalid customerID -> not allowed 3.customer not exist ->ok
10		cancelEvent()	1.cancel on own server -> ok 2.cancel on other server -> ok(UDP needed) 3.cancel a not registered event -> error shown 4.invalid eventide -> not allowed

11		swapEvent()	1.swap with a non-existent event -> not allowed 2.swap a not registered event -> not allowed 3.swap with a event from other server -> same conditions as a book event 4.swap with an event from other server in same week (weekly limit reached)