Nwatu Victor Praise

40206992

Distributed Movie Ticket Booking System Using Java IDL

Requirements

The system consists of 3 servers (cinemas) with each server having an admin with customers associated with the servers. It is designed to meet certain requirements such as:

For Customers:

- Customers can book movies from their respective cinemas.
- Customers can also book movies from other cinemas.
- Customers can cancel movie tickets.
- Customers can view booked tickets.
- Every customer interaction is recorded on a logfile (one per customer).
- Every customer should be able to exchange tickets

For Admins:

- Admins can create movies for their respective cinemas.
- Admins can delete movies from their respective cinemas.
- Admins can get movie availability across all cinemas.
- Every Admin interaction is recorded on a logfile (one per admin)

For Server:

- A log file that records every operation performed

There are certain limitations such as:

- Customers cannot book the same movie from different cinemas.
- Customers can book at most three movies from different cinemas.
- Customers cannot book more than available tickets.
- Customers can only cancel tickets that was booked by them.
- Admins can only create movies one week from the current date.
- If an Admin deletes a movie, then all customers booked for the said movie will be booked to the next available movie provided there are enough tickets.

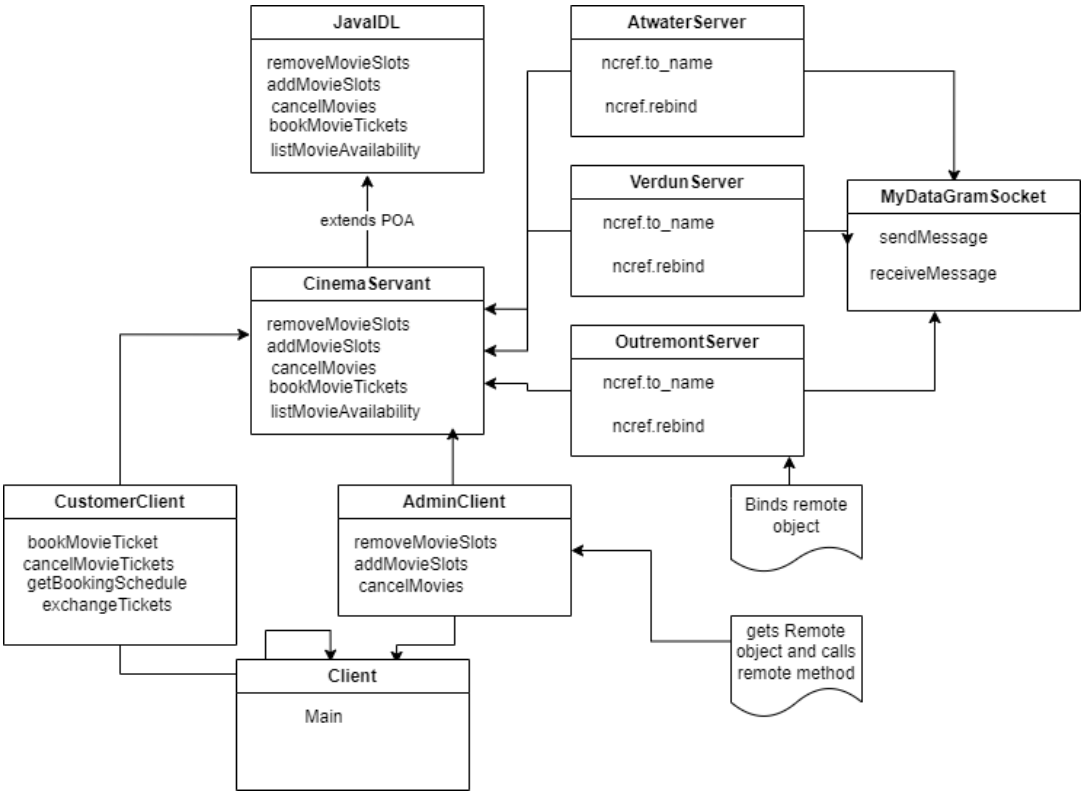
Design

The system is built using Java IDL to allow remote procedure calls. Java IDL enables objects to interact regardless of programming language, Java IDL provides an Object Request Broker, or ORB which enables low-level communication between Java IDL applications and other CORBA-compliant applications. the three cinema servers are responsible creating instances of the ORB and

for binding the remote objects by name (ncref.rebind(path,href)). The customer and admin get the remote object for the appropriate cinema and invoke remote methods implemented in RMI Implementation.

DMTBS.idl is the idl file with all the methods declared, CinemaServant.java which extends the CinemaPOA and has all the methods defined. AtwaterServer.java, VerdunServer.java, OutremontServer.java with each representing a cinema, binds the remote object by name while CustomerClient.java and AdminClient.java invoke the remote methods. Cinema.java has the main function, instances of AdminClient and CustomerClient are created in Cinema.java and used to interact with the system. To ensure a customer cannot access admin privileges, all client methods are called in CustomerClient while admin in AdminClient.

With the requirements in mind, the system is designed such that on the creation of a CustomerClient object a user_name is passed as a constructor argument for example "CustomerClient("ATWC2345")" which signifies the customer's cinema ("ATWC2345 for Atwater", "VERC2345 for Verdun") from the CustomerClient class, the username is used to make a connection (lookup) the corresponding cinema, the CustomerClient then provides methods that calls the remote methods defined in DMTBSIMPL.java (bookMovies() which makes a call to the remote method bookMovieTickets()). This is the same for admin on creation of AdminClient instance, a user_name corresponding to cinema is passed as an argument with the AdminClient class providing methods that calls the remote methods defined in DMTBSIMPL.java. Next, we will look at the requirements and how they are met.



Class diagram

For Admin:

- **Requirement 1: Admins can create movies for their respective cinemas.**
  For this, an addMovieTickets method that returns a Boolean and takes as parameters (movieName, movieId, numberOfTickets) is implemented in DMTBSIMPL, a nested hashmap stores all the supplied information returning a message that signifies the movie has been created. If the movie with movieID already exists, then number of tickets is updated. (One hashmap per server). When movieID is passed, a check is performed to ensure the date does not exceed one week from the current date. From the AdminClient, a method called addMovies which takes the same parameters makes a call to the remote method and passes the parameters as arguments. The addMovies method returns a message to the user (based on return value of addMovieTickets) informing the user if the movie was created successfully or not.
- **Requirement 2: Admins can delete movies from their respective cinemas.**
  For this, a removeMovieSlots method that takes movieName and movieId is implemented, this method searches the hashmap for said movieName then movieId and deletes movieId along with the number of tickets. To fulfill one of our limitations, an arraylist that keeps track of movies booked along with customer who booked them is implemented. Upon successful deletion, arraylist is searched for deleted movieId and all customer that has booked a movie with the movieId are booked for the next available

movie provided there is sufficient tickets (a rebookMovie method is implemented for this regard)

- **Requirement 3: Admins can get movie availability across all cinemas.**
  For this to be done, the various server will need to communicate with each other. To achieve this, a MyDatagramSocket.java class which is responsible for sending and receiving messages is implemented (sendMessage(), receiveMessage()). A listMovieAvailability() method which takes movie name as a parameter is implemented. When movie name is passed, it is sent to other servers using the sendMessage method and the servers search through their hashmaps and returns all available instance of particular movie along with movieId and number of tickets. From AdminClient, a public method listMovies() which takes movieName as a param makes a call to the remote method listMovieAvailability and prints out result.

With this, all admin requirements along with limitations have been met. Next, we discuss the customer requirements.

For Customer:

- **Requirement 1: Customers can book movies from their respective cinemas.**
  This requirement is met through a bookMovieTickets method (returns a boolean) that takes as param movieName, movieID, customerID, numberOfTickets. This method searches hashmap for movie name and id, if records exist, the number of tickets in hashmap is updated according to numberOfTickets supplied as a param, then a record of request is added to arraylist. To handle one of the limitations, a check is done to ensure requested tickets do not exceed available tickets. If it does, then an error message "Requested tickets exceed available tickets" is displayed both on the client and server side. If the record does not exist a "Movie does not exist" message is once again displayed on the client and server side. From CustomerClient, a method bookMovies takes same parameters and makes a call to the remote method bookMovieTickets is implemented, upon success, a "movie booked successfully" message is displayed and then a hashmap responsible for saving history on the client end is updated.

- **Requirement 2: Customers can also book movies from other cinemas.**
  It should be possible for a customer from one cinema (server) to book a movie from another cinema provided movie exists and there are enough tickets. For this, from CustomerClient, a bookFromOtherCinema method takes as param cinemaName, movieName, movieID, numberOfTickets. Just like the listMovieAvailablity for the admin, MyDatagramSocket is used to send a message to the appropriate server (customerId, movieName, numberOfTickets and movieID) the movie is then booked (provided earlier mentioned conditions are satisfied) and recorded in arraylist.
  To satisfy our limitations, before a movie is booked from another cinema, the earlier mentioned hashmap which saves history on the client end (one per customer) is first checked to ensure the movie has not already been booked. If it has, an error message

notifies the user that the same movie cannot be booked across multiple cinemas. A count variable is also used to ensure a customer can only book a maximum of three movies from other cinemas.

- **Requirement 3: Customers can view booked tickets.**
  For this, a getBookingSchedule which takes customerID as a param searches the arraylist on the server end and returns all records that match customerID. For simplicity's sake, since each customer already has a hashmap with the respective history recorded, this hashmap is used to display the booking schedule. The method is also called getBookingSchedule but it takes no param.

- **Requirement 4: Customers can cancel movie tickets.**
  This requirement has certain conditions that must be met, firstly we ensure the movie that the user wants to cancel was booked by the user and which cinema the movie was booked from. To do this, from the CustomerClient, a cancelTicket method which takes movieID, movieName, numberOfTickets as param is implemented, the method checks hashmap to ensure the movie with movieID has been booked by the customer, if it has, the method checks the movieID to determine what cinema the ticket was booked from. After determining that, it calls on the remote method cancelMovieTickets which takes the same parameters as earlier mentioned along with the customerID. The remote method first checks the movieTicket hashmap and ensures the movieName with movieID exists then it adds the supplied numberOfTickets to the already existing numberOfTickets and records operation in arraylist. On success or failure, a message is displayed both on the client and server side. For the second condition, if the movieID corresponds to another cinema then a cancelMovieFromOtherCinema method is called which calls the cancelMovieTicket method of cinema (server).

- **Requirement 5: Customers can rebook movie tickets.**
  This requirement utilizes requirement 4 when a customer wants to exchange tickets. The customer passes the old movie id, new movie id, new movie id, movie name, and the number of tickets. The exchangeTicket() function accepts all these as parameters. Firstly a check is performed to ensure the customer has actually booked the movie with the old movie id, if the customer has, then a check is performed to ensure the new movie is available (enough tickets), then, if the movie is available then the cancel movie tickets function is called to cancel the old movie then the customer is rebooked with the new movie.

When each server begins running, a text file that keeps track of all operations performed on the server is generated. This also applies to the creation of a customer or admin, a text file corresponding to the user is generated and records all operations performed by the users. Also, to ensure concurrency, a synchronized keyword is placed in front of every method.

With this, all requirements and limitations have been implemented, next, we move to the verification (testing) phase.

Testing

#	Method Name	Scenario	Action	Expected outcome	Status
1		Server startup	Run servers	Servers (cinemas) should start displaying an appropriate message	pass
2		Client connection	Create AdminClient passing adminID	A connection is made to the corresponding cinema	pass
			Create CustomerClient passing clientID	A connection is made to the corresponding cinema.	pass
			Log file	Log file for each client and admin along with all three servers is created	pass
3	addMovieSlots	Adding movies (Admin)	Valid parameters are passed	Movie is created and added to hashmap along with a success message	pass
			Movie date exceeds on week from current date	An error message is displayed (“can only create movie one week from current date”)	pass
			The same parameters passed again	Only number of tickets is updated	
			Invalid Parameters	An error message (“Movie creation unsuccessful”)	pass
			Log file	With each interaction, both server and admin logfile is updated	pass

4	removeMovieSlot	Removing movies (Admin)	Valid parameters are passed	Movie with specific id is removed from hashmap	pass
			Movie does not exist	Error message (“movie does not exist is displayed”)	pass
			Log file	With each interaction, both server and admin logfile is updated	pass
		Rebooking tickets (Admin)	Sufficient tickets	When movie is canceled, all customers who booked specific movie are automatically booked for the next available movie	pass
			Insufficient tickets	If next available movie has insufficient tickets an error message (“insufficient tickets available”) is displayed	pass
5	listMovieShowsAvailability	Admin requests availability of certain movie	Movie exists	Server sends a message to other servers and the number of tickets and movie id of the particular movie across servers is returned	pass
			Movie does not exist	Error message (“movie does not exist”)	pass
6	bookMovieTickets	Book movie from same cinema (Customer)	Valid parameters	Movie is booked and hashmap ticket number is updated, transaction is recorded in both server’s ArrayList and the client’s hashmap	pass
			Not enough tickets	Requested tickets exceed available	Pass

				tickets, error message (“not enough tickets”) is displayed	
			Movie id does not exist	User request invalid movie, an error message (“movie does not exist”) is displayed	pass
		Book movie from different cinema (Customer)	Valid parameters	Just like booking from the same cinema, all three scenarios must hold	pass
			Weekly limit exceeded	Client cannot book more than 3 movies from other cinemas per week. If the limit is exceeded error message (“Weekly limit exceeded”) is displayed	pass
			Same movie is already booked	If movie is already booked from another cinema, an error message (“movie already booked”) is displayed	pass
7	getBookingSchedule	Get movies booked by a customer	CustomerClient calls method	Returns all movie booked by the customer	Pass
8	cancelMovieTickets	Cancel movie from the customer’s cinema with movieID	Valid parameters	Movie is canceled and the ticket number of hashmap is updated. A success message is displayed to the user	pass
			Invalid movieID	If the provided movieID is of a movie that customer has not booked, then an error message (“Invalid	pass

				movieID") is displayed	
			Log file	With each interaction, both server and admin logfile is updated	
		Cancel movie from other cinema with movieID	Valid parameter	All conditions for canceling from same cinema must be met	pass
			Update appropriate hashmap	The hashmap of other cinema should be updated	Pass
9	exchangeTickets()	Swaps tickets for customer	Customer booked movie	A check is first performed to ensure the customer has booked the movie he wants to exchange	pass
			Enough tickets	After the first case passes, another check ensures the movie customer wants to swap with has enough tickets to accommodate the swap	pass
			Cancellation successful	If there are enough tickets, the cancel ticket function is called and then ticket exchange is done	pass
			Invalid parameters	If at any instance any of the above mentioned conditions fail, the appropriate error message is displayed	
10		Log file	Records history	With each interaction from customer and admin, their appropriate log file along with the	pass

				server log file is updated.	