Design and Implementation of a Distributed Reservation System

Nan Zhu
School of Computer Science
McGill University
nan.zhu@mail.mcgill.ca

September 25, 2013

Abstract

This report presents the design and implementation of part 2 in the 1st deliver of the course project in COMP512. We mainly talked about the architecture, design considerations and detailed working mechanism of the system. We build the system with Java NIO (Non-blocking IO or New IO) subsystem instead of the usual one-thread-per-socket architecture. This design can avoid the large context switching overhead when the concurrent connection number is large (usually larger than 2X of the core number of the server). To handle the sharing data like customer information, we build a special resource manager which manages the all data instances (performing like a database), while the flight, car, room, customer resource managers process the requests by querying the database and caches the mostly recent used results in the memory.

1 Architecture

Figure 1 shows the system architecture. The DataStore is a server-end program which has the global data instances; Resource Managers are also server-programs each of which handles the specific resource management tasks, like flight/car/room/-customer reservation/management; Middleware is to deliver the requests for reservation to the corresponding resource managers; Client is to generate and invoke the reservation requests. ReservationMessage subsystem defines different types of messages communicating between components.

All these programs are build on Java NIO for better performance under large number of concurrent connections. I encapsulate the Java NIO APIs with an additional API layer to mask all network relevant details to business logic of DataStore/ResourceManager/Middleware/Client. Figure 2 shows how I decouple the network communication mechanism and the business logic. NIOReactor is a server-end program which listens for the incoming connections and organizes the processing works into pipeline, it also can be configured to be in both of the server and client role. When Reactor is in the client role at the same time with the server role, it additionally invoke and listen for the connections to remote servers. This design enables me to reuse network communication code for DataStore/ResourceManager/Middleware. Even NIOReactor contains the implementation of invoking connections to servers as a client, I build an independent NIOClient module to support the applications which runs as a "pure" client. The most important difference between NIOClient and the client part of

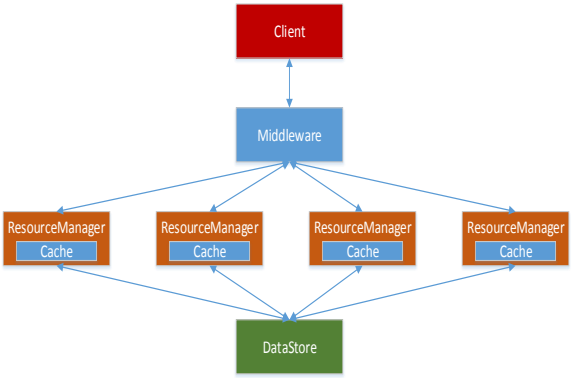

Figure 1: System Architecture

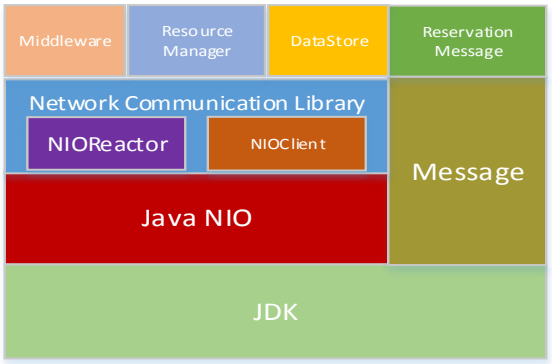

Figure 2: Software Building Blocks

NIOReactor is that when receiving a response from the server the NIOReactor forwards it to the NIOClient while NIOClient performs local processing, like interprets and displays it to users. The Message module defines a base class for all messages between NIOReactor and NIOClient as well as the methods to *serialize* and *deserialize* them.

2 NIO Network Module

2.1 Why Java NIO

The NIO network module is built with Java NIO. In classic network program, the server-end program always starts a new thread for each socket to process the request while keeping the main thread unblocked. The shortcoming of this architecture is that the thread number is increased proportional to the number of concurrent connection numbers. Too many threads in the system can involve large overhead on context-switching. When a socket is blocked on read/write operation, it is scheduled out from the running queue while the other thread will be scheduled from pending, if any. A context switch happening requires a certain amount of time for doing the administration - saving and loading registers and memory maps, updating various tables and lists etc.

Java NIO is introduced to Java standard library since Java 1.4. In Java NIO, a *Channel* represents an open connection to an entity such as a hardware device, a file, a network socket, or a program component that is capable of performing one or more distinct I/O operations, for example reading or writing. The program based on Java NIO uses a *Selector* to monitor multiple *Channel*s, which register with the channel with their interest operations, e.g. READ/WRITE/CONNECTION/ACCEPT. The selector will pick those channels which are ready for their interest operations and pass them to the user-implemented handlers. Algorithm Z describes this procedure.

Java NIO performs much better then standard Java IO under large number of concurrent connections especially when the connections are long ones. I consider that a client connection will keep alive until the response traverse back from the ResourceManager via Middleware, I choose Java NIO to handle network stuffs.

2.2 Implementation of NIO Network Library

The NIO Network Library consists of 3 main components: NIOReactor, NIOClient and Message.

NIOReactor is a base class for server programs. It contains two selectors to monitors the channels which are connected from clients to it and the channels which are oriented by itself to other remote servers respectively. The second selectors is disabled by default, but users can call the API *setClientRemoteEnd(String servername, String IP, int port)* to enable it, where servername is the identifier of a server, IP and port are the location information of the server. Reactor maintains such a HashMap¡String, SocketChannel¿ to store the channels from it to remote servers. The network programming details are masked from the users. To use NIOReactor, users extends it and only implements the abstract method, *dispatch(Message msg)* which describes the handler of different message types. , where the msg is the message received from either the clients or the remote servers.

NIOReactor provides the following APIs for users:

- reply(Message msg): the program returns msg to the end which sent it to the server;
- replyAll(Message msg): the program sends msg to all the clients connecting to it;
- forward(String servername, Message msg): the program sends the msg to the server whose name is servername;

NIOClient is a base class for the client program. It contains only one selector to monitor the channels connecting to the servers. Similar to NIOReactor, the users of this class only need to extends this class and implements the abstract method *dispatch(Message msg)*.

NIOClient provides the following APIs for users:

- send(Message msg): the program sends msg to the remote server;

3 Business Components

The client request is sent from the client to Middleware which forwards to the corresponding ResourceManager, which either forwards it to the DataStore or handles it locally based on the types of the message and its own status.

The functionalities of these 4 processes have been introduced in the prior paragraph, here we mainly describes the interaction between ResourceManager and DataStore.

Generally speaking, ResourceManager can be taken as the cache for DataStore. There are four types of messages which involves the interaction between them:

- Query: When ResourceManager receives such a message, it firstly queries its local memory, if it keeps such a record, it directly response to the client; otherwise, it queries DataStore which returns a QueryResponse, this response will make the ResouceManager inserts a new entry to its local memory;
- Add: When ResourceManager receives such a message, it firstly inserts a new entry in its local memory and forwards it to DataStore;
- Delete: When ResourceManager receives such a message, it aggressively delete the corresponding entries in its memory to avoid the data inconsistency (delete can be failed on DataStore) and forwards it to the DataStore;
- Reserve: When ResourceManager receives such a message, it forwards to the DataStore which returns a response containing the remaining number of the resources, the ResourceManager receives the response and update its local memory.

There are some special issues to clarify: the Middleware randomly chooses a ResourceManager to forwards Itinerary requests to the DataStore; DataStore returns a ItineraryResponse containing all resource amount to all resource managers.

4 Test Method

- in the root directory of source code, type "ant";
- go to the "out" directory;
- run 1) java -jar production_datastore.jar 127.0.0.1 5005, start the datastore instance; 2) java -jar production_rm.jar 127.0.0.1 5001 config/rm_config.xml(this method should be executed for 4 times on different ports for start resource managers for different resources, and config/rm_config.xml is a XML file contains the address of datastore); 3) java -jar production_mw.jar 127.0.0.1 5000 config/mw_config.xml, start the middleware and loads the config file containing addresses of resource managers; 4) java -jar production_client.jar 127.0.0.1 5000, start the client and connects to middleware.