**COMP 512 Project deliverable #1 - Report**

Usage

*TCP implementation*: ant server, ant client. IPs/ports are properties in build.xml.

*WS implementation*: ant rms, ant middleware, ant client. IPs/ports are properties in build.xml.

General architecture

Our architecture is as follows:



The main point of interest here is the additional RM for customers. We obviously had to centralize the management of customers, so it was only a question of deciding whether it should be in a new RM or at the middleware. The option of having the customers managed in their own RM instead of the middleware was appealing to us because of the following reasons:

* The middleware is simpler, since it doesn’t have to keep track of the customers. This removes the need for extensive synchronization on the middleware.
* There is no new code to add to manage the customers at the RM.
* If, in the future, we need to have multiple middleware servers (say, for load balancing), we won’t have to make as many changes to our architecture.

Hence, some commands then require a slightly different handling. All the commands that only apply to one of the “FCR” (Flight, Car, Room) RMs are unchanged (e.g. newflight, deleteflight, queryflight). When a new customer is to be created, we only create it on the customer RM. We chose to do this so that we wouldn’t have “useless” customers on the other RMs in case the customer never got a flight, for example. Then, any command that concerns a customer (e.g. reserveflight) is executed in the following manner:

* The middleware asks the customer RM if the given customer exists.
* If no, an error is returned to the client.
* If yes, the middleware executes newcustomerid,<cid> on the appropriate FCR RM. This is to ensure that a customer with the given id exists on the RM, so we execute it indiscriminately and let it fail if it turns out the customer already exists on that RM. Then, we execute the original request and return the result to the client.

Some commands require the middleware to talk to several of the FCR RMs. These are handled separately. For example, the deletecustomer command is executed on all RMs.

For any of the commands that talk to several RMs, we assume that the data will not be modified between the time we check it and use it. For example, a case that could corrupt data would be if we try to reserve a flight, confirm that a customer exists, and before we can reserve the flight, someone else deletes the customer. We do not handle such cases, as they are not the focus of the assignment.

However, we handle concurrent modification of data structures on a single unit of the system. On the ResourceManager, the core shared data structure (RMHashtable) is already synchronized by virtue of extending Hashtable, so we do not have to synchronize direct accesses to it. Some data access commands might cause data corruption in a multithreaded environment, so we synchronize access to the RMHashtable in those cases. Some cases may lead to temporarily invalid data that is nevertheless fixed once all threads have executed, so we did not synchronize them. All the cases (whether explicitly synchronized or not) are document in the java file for ResourceManager.

The middleware implementation differs substantially between the web services and the TCP. Details on the synchronization issues of each can be found below.

Web services architecture details

For the web service implementation, we chose to have a middleware that would implement the ResourceManager interface. This allowed us to have re-usable ant build targets, as the procedure for building the server and the middleware was identical in many respects. The main difference between the two was the implementation of the ResourceManager itself.

On the RM side, ResourceManagerImpl is essentially unmodified. On the middleware side, the class exposes the ResourceManager web methods, and mostly forwards the requests to the appropriate RMs (with additional logic as required). In this architecture, there are shared objects: the ResourceManager proxies to the RMs. To synchronize access to those, we chose to implement a proxy pool (one for each RM). The pool provides methods for checking proxies out and back in, and is initialized when the ResourceManagerImpl is constructed. This has the main advantage of freeing us from the overhead of having to create a new proxy for every request. The pool maintains its available proxies in a queue that it synchronizes access to. A semaphore (counting the number of objects in the queue) is used to make threads wait for their proxy as necessary.

Note that the IP addresses/ports for the RMs are also in the ws/etc/middleware/web.xml file. This is because the middleware web service needs them and the only way we know of to give them to it is to put them in that file. Ideally, we should have an ant task that generates the file from the properties given in build.xml.

TCP architecture

In our TCP implementation, all requests (client to middleware and middleware to RMs) are strings. All responses (RM to middleware and middleware to client) are objects of type RMResult. This is a serializable union-like object. It contains a tag (which is an enum) describing the type it encloses and an object of the given type. We enforce that the data types used are: integer, boolean, string, RMHashtable and exception (which is used for errors). All of these are serializable, so there are no issues with transfers.

The socket handling is identical on the middleware and on the RM. In both cases, a server socket constantly awaits new connections. When a new connection is established, the server socket will immediately hand it off to a newly-spawned thread for further processing. On the middleware, this is taken care of by the ClientServiceThread class, which is responsible for triaging requests to their appropriate RMs (and executing multi-RM requests as needed). On the RM, the ResourceManagerImpl class itself is ran under a new thread.

One advantage of outsourcing the customers to the RM is that on our TCP middleware, there are essentially no shared data structures. Thus, no special synchronization is required.