COMP 512: Milestone #1

Jonah Caplan, Haitong Yang

1 Deliverables

The following deliverables are required for the first project milestone:

Distributed web services
Distribute a pre-existing server/client system built on web services.

TCP Implementation
Implement the same distributed system using TCP instead of web services.

2 Distributed web services

The distributed system consists of three components: the end-user client, the middleware (MW), and the resource manager (RM) servers. We chose to have four RMs: one for each type of reservation item as well as one for customers. The middleware consists of a client and a server both of which are modified versions of the original code provided. The main functionality that differentiates the MW from the RMs is the booking of itineraries which requires coordination between the MW and each RM.

This section will briefly describe the changes made to the client and RMs and then discuss the middleware design. We assume the reader is familiar with the original source code.

2.1 Client modifications

No modifications were required for the Client implementation. For convenience, the parsing of commands was separated from the main run loop in order to facilitate test writing by providing a code-based interface to enter commands.

2.2 RM server modifications

The server required several modifications to support the middleware. The original RM implementation assumed that customer information was located in the same hash table as the services. The reservation of items consisted of two main steps: check that the item was available and reserve it, then update the customer data structure so the reserved item is on their bill. In order to allow the middleware to coordinate this process among the RMs, it was necessary to seperate the reservation of items and the updating of customer data into separate methods and to expose these methods to the ResourceManager interface. It was also necessary to allow the MW to cancel a reservation (details in the section on MW). Methods were added to the interface to allow explicit updating of the customer reservations and cancelling reservations on specific items.

2.3 Middleware implementation

The middleware implementation is fairly straight-forward. It consists of a server that implements the ResourceManager interface. Upon a call from a client, the MW methods create a new client object to pass the request to the appropriate RM server.

The main difficulty in the MW design is to safely handle the itinerary requests which consist of one or several flights as well as a room and car reservation. The algorithm for reserving an itinerary is shown in Figure 1. The MW attempts to successively reserve each item in the itinerary. The itinerary is cancelled if any of the items cannot be reserved. In this case the RMs must be notified for each item that was successfully reserved.

A similar approach was used for deleting customers. In this case, the customer's bill was retrieved from the client RM and each item was cancelled.

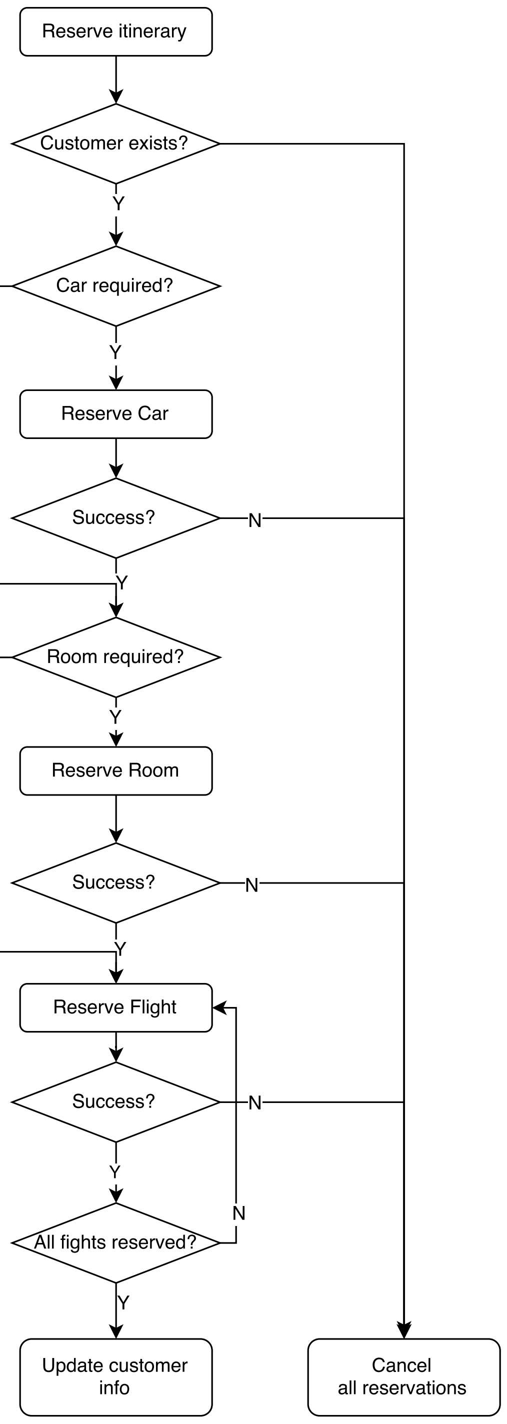

Figure 1: Algorithm for reserving itinerary.

2.4 Synchronization issues

The code provided ensured that only one thread can have read or write access to the RMs hash table at a time. This is not sufficient as the following example will illustrate. Consider the code in Listing 1. First, the object `curObj` must be retrieved from the hash table and we know that this is itself a safe retrieval and `curObj` may not be corrupted. At line 6 there is a check to see if `curObj` is null. This check is potentially unsafe because it could be that `curObj` is not null at line 6 but is null at line 11. It may be tempting to synchronize on `curObj` itself, but this is potentially a null pointer exception if the synchronization occurs before the check at line 5. Postponing the synchronization block until line 10 results in the exact same problem as before. It is necessary to synchronize all accesses to objects from the moment their reference is retrieved until the actions that must be taken with them are completed. The updated code is shown in Listing 2.

Listing 3 shows a potential alternative to Listing 2. We chose not to implement this technique because we are less confident in its safety (i.e. that the mutex implementation guarantees that there will only be one release per successful acquire) and unsure of its performance benefit. The idea is to first acquire the mutex and then retrieve the reference to an object. The method synchronizes on the object once it is sure that this is safe and then releases the mutex. This allows other threads to access the table.

There is another issue regarding the deletion of clients. It is possible that a flight could be deleted after the MW retrieves the customer data but before the flights can be cancelled. It does not seem like an adequate solution to force a synchronized block at the middleware level. This may be less of a synchronization issue and more of a domain specific issue. An argument could be made that this type of error must be caught by the middleware and appropriate actions (e.g. cancellation or refund) should be initiated. Note that this problem does not create a runtime error but we do feel that some extra actions should be taken by the middleware in case the cancellation of a reservation fails.

Listing 1: This code has several potential synchronization issues.

```java
 1  protected boolean deleteItem(int id, String key) {
      Trace.info("RM::deleteItem(" + id + ", " + key + ") called.");
 3    ReservableItem curObj = (ReservableItem) readData(id, key);
      // Check if there is such an item in the storage.
 5    if (curObj == null) {
        Trace.warn("RM::deleteItem(" + id + ", " + key + ") failed: "
 7          + " item doesn't exist.");
        return false;
 9    } else {
        if (curObj.getReserved() == 0) {
11        removeData(id, curObj.getKey());
          Trace.info("RM::deleteItem(" + id + ", " + key + ") OK.");
13        return true;
        } else {
15        Trace.info("RM::deleteItem(" + id + ", " + key
              + ") failed: " + "some customers have reserved it.");
17        return false;
        }
19    }
    }
```

Listing 2: Almost the entire method body must be synchronized to prevent potential null pointer exceptions.

```java
protected boolean deleteItem(int id, String key) {
  Trace.info("RM::deleteItem(" + id + ", " + key + ") called.");
  synchronized (syncLock) {
    ReservableItem curObj = (ReservableItem) readData(id, key);
    // Check if there is such an item in the storage.
    if (curObj == null) {
      Trace.warn("RM::deleteItem(" + id + ", " + key + ") failed: "
        + " item doesn't exist.");
      return false;
    } else {
      //Actions on curObj
    }
  }
}
```

Listing 3: A potentially faster synchronization method that uses a mutex and allows the main computation to be synchronized only to the object being modified.

```
protected boolean deleteItem(int id, String key) {
  try {
    mutex.acquire();
  } catch (InterruptedException e) {
    e.printStackTrace();
    return false;
  }
  ReservableItem curObj = (ReservableItem) readData(id, key);
  // Check if there is such an item in the storage.
  if (curObj == null) {
    Trace.warn("RM::deleteItem(" + id + ", " + key + ") failed: "
        + " item doesn't exist.");
    return false;
  } else {
    synchronized (curObj) { //synchronize on the curObj
      mutex.release();   //safely release the mutex,
                //now other threads can access hash table
      //do operations on curObj
      //...
    } //end sync block
  }
}
```

3 TCP Implementation

Our design goals for the TCP based system were to reuse as much of the original code as possible and to simplify the communication protocol. We wanted to avoid spending time debugging the passing of messages between processes and instead focus on the actions taken by each process based on the information contained in the message.

4 Communication protocol

Very little information needs to flow between processes and code already existed to manually parse command line entries from the client. It made sense to have the client send unparsed commands directly over TCP and move the pre-existing code for parsing commands to the RMs. Setting up the command parsing on the RM was very simple because the client code was already designed to interface with the `ResourceManagerImpl` class. The middleware would then parse the first argument in the command and forward it to the correct RM. The RM then fully parses the command and updates its data structures accordingly. In the case of reserving an itinerary, the MW needed to parse the entire command and build new commands to make reservations with each RM. The RMs always return a string representing the original integer or boolean return types.

The server classes that were added are shown in Figure 3. The main Server class has a `ServerSocket` object and is responsible for waiting for new clients. The server creates a new `ServerThread` when a client is accepted. The `ServerThread` is responsible for the specific client socket and parsing the client requests. The parsed arguments can then be used to interact with the `ResourceManagerImpl` as was the case with the web services.

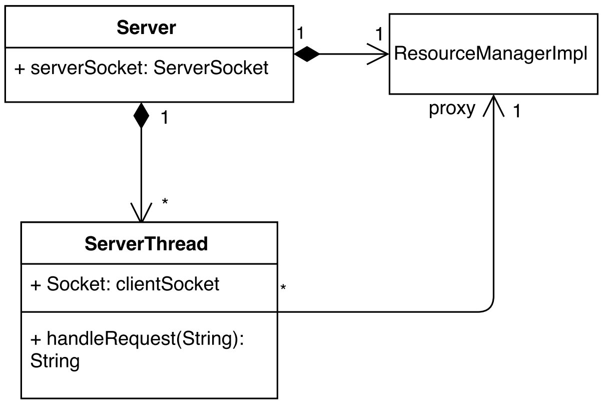

Figure 2: The additional classes for the RM servers.

The middleware consists of two classes. The `Main` class functions like the `Server` class and waits for client connections before creating threads. The `TCPMiddlewareServerThread` (maybe some refactoring is still in order) is responsible for forwarding the request to the appropriate RM. A new client socket is created for each RM. The MW thread mostly sends commands without looking at their contents. The exceptions are `deleteCustomer` and `reserveItinerary`. Both these functions require several interactions with all the RMs.

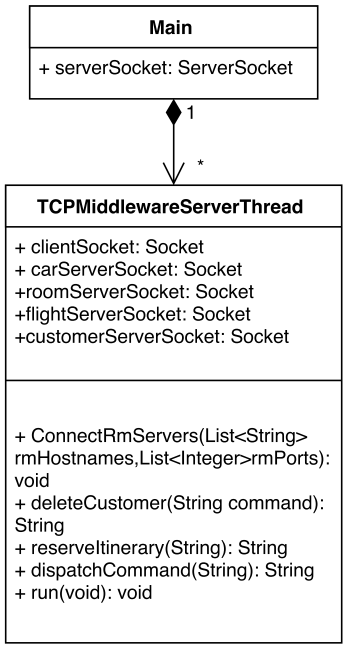
Main
+ serverSocket: ServerSocket
1
*
TCPMiddlewareServerThread
+ clientSocket: Socket
+ carServerSocket: Socket
+roomServerSocket: Socket
+flightServerSocket: Socket
+customerServerSocket: Socket
+ ConnectRmServers(List<String> rmHostnames,List<Integer>rmPorts): void
+ deleteCustomer(String command): String
+ reserveItinerary(String): String
+ dispatchCommand(String): String
+ run(void): void

Figure 3: The class diagram for the middleware.

The possible downside of transmitting information in this manner is that manually constructing and parsing strings may prove cumbersome and error prone if large data types were required. Packing information into objects and relying on java to serialize their represention would be easier to manage if the data passing

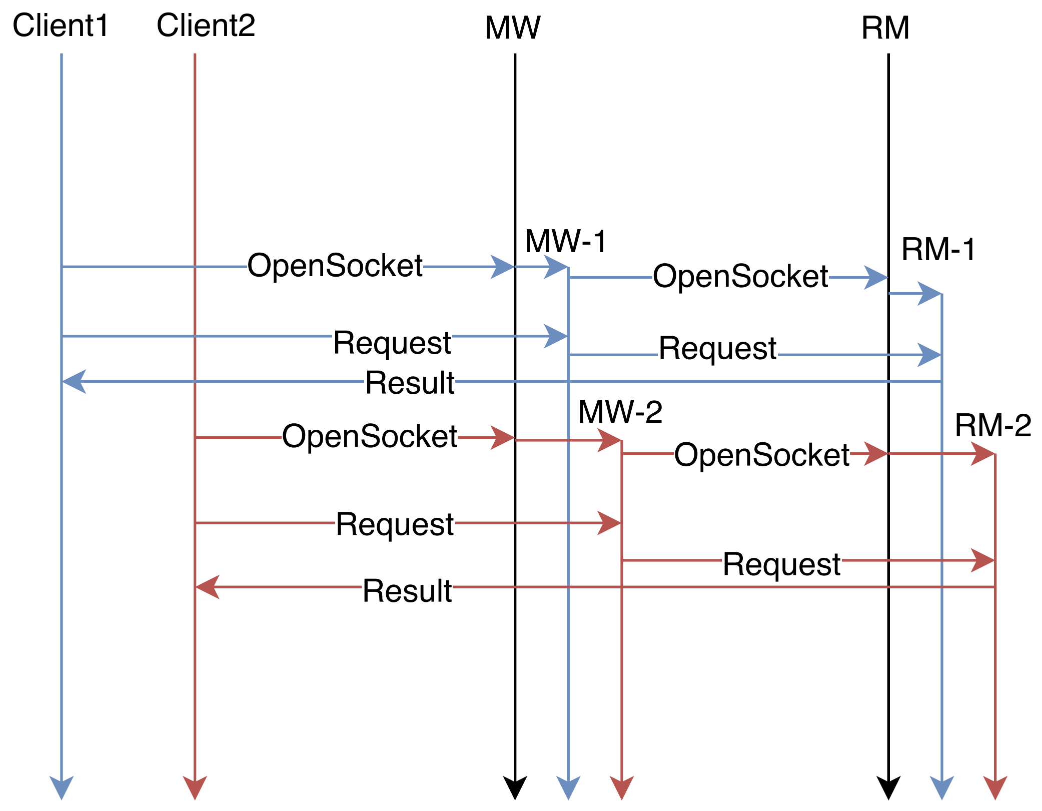

Figure 4: Two clients communicate with RM through the MW at same time.

between processes was more complex. However, we think our current implementation is fairly lightweight and effective given that parsing code was already provided and the small message sizes. If we had more time or if this were a real (i.e. not class) and possibly agile design project, it might still have made sense to start with this design as a transitional step towards a more robust implementation using objects as we don't want to change too much of the implementation in a single iteration.

5 Thread management in middleware and resource managers

A new thread must be created in the MW or RM when a client connects to the MW or the MW connects to the RM for a specific client, respectively. This allows as much work to be done concurrently per client as possible despite the fact that some RM methods must be synchronized. The basic flow is shown in Figure 4. Both clients must initially pass through the main thread when initiating communication. The main MW thread then creates a dedicated thread to deal with that client. The same process occurs between the MW and the RM. The client communicates directly with the dedicated thread when it makes future requests.

6 Testing

The testing that we have done so far could be more extensive and automated. We also were not able to do any performance profiling. It would be interesting to profile the difference in performance between TCP and web-services quantitatively however we have noted that web services are observably at least ten times slower. We have written two main tests that are compatible with both implementations. The first test (`TestSingleClient`) simply checks that commands are executing correctly, returning both true and false depending on the status of the system. The second test (`Test01`) checks that the system can handle multithreading. Five client threads are created and each thread attempts to reserve as many itineraries of one car, room, and plane, as possible until there are no more resources left. Each thread then immediately deletes its customer. The test checks that after all that all the resources have been properly accounted for after all the customers have been deleted. The tests cover the basic functionality of the system and were useful in debugging despite their simplicity.