COMP 512: Final Report
Distributed Reservation System

Jonah Caplan, Haitong Yang

1 Overview

This report presents a distributed reservation system that was derived from a centralized implementation. We implemented a middleware (MW) layer between the client and four resource managers (RMs), shown in Figure 1. The RMs were responsible for maintaing records of flights, rooms, hotels, and customer histories. We further added support for transactions, two phase commit (2PC) protocol, and recovery for the RMs. The implementation for each modification will be discussed in the following sections.

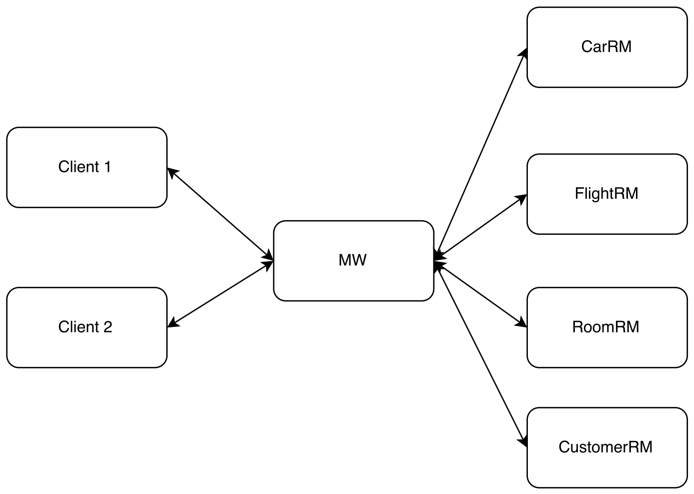


Figure 1: The system consists of a middleware layer and four resource managers.

2 System Architecture

2.1 Customer RM

The first design decision we made in distributing the reservation system was to create a fourth RM for customer information. We intended that MW design would be simplified if we did not interweave coordination and data storage responsibilities in the same unit. An RM class, separate from the MW class would likely have been implemented in the MW layer if they had been on the same computer for this reason. It made sense to place the RM in a separate process with a web service interface so that customer transactions were handled in the same fashion as other RMs by the MW.

One problem with placing the client information in a separate server is that in a real application most transactions would likely only involve one client. The provided code retrieves client information for almost every command and many of these requests are likely redundant. The cost of these redundant client requests is significantly higher with our implementation as opposed to leaving it on the MW. Our preferred solution would be to rewrite the original API in such a way that transactions must be structured such that relevant client information is retrieved once at the beginning and written once at the end of a transaction.

2.2 Separation of Concerns

Figure 2 shows a more detailed view of the system architecture. The implementation basically follows the design patterns discussed in the lectures. We decided to break up the transaction manager (TM) responsibilities across the MW and RM. The TM in the MW is responsible for providing new transaction identifiers, keeping track of which RMs have been accessed by a transaction, and maintaining a timer for each transaction. These concerns naturally lend themselves towards a central location in the system. The TM located near the RM is responsible for maintaining rollback information for each transaction, thus avoiding the need to send data from the RM to the MW.

One change we would likely make given a chance to redo things is to connect the MW to the lock manager (LM) indirectly through the TM. The middleware should ideally only field client requests and forward them to the resource manager. The lock management implementation in the MW produced very tangled code that would be more difficult to maintain in a realistic system. The placement of TM code was rather targeted and providing the TM with a bit more information about the commands would have allowed it to interface with the LM in a much more organized fashion.

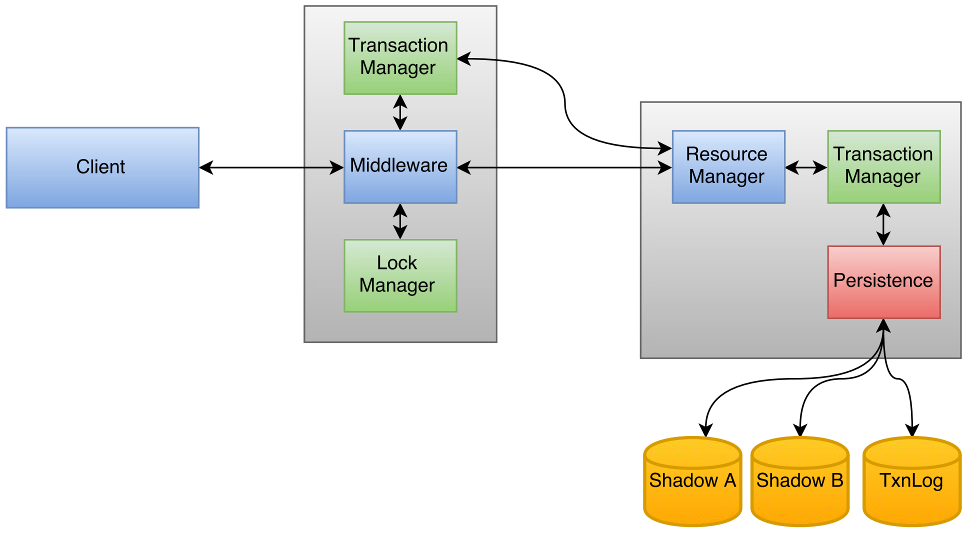


Figure 2: Detailed view of the system architecture

2.3 Transactions and 2PC

In our system transactions immediately overwrite data in main memory. A list of rollback information is kept for each record modified by the transaction. No writes to the main table are required in the case of a commit. For aborts, changes to a record made by a transaction are overwritten with the stored rollback information. A sequence diagram for a transaction that reads and writes shows the interaction between all the main components of the system (MW,RM,TM,LM) in Figure 3.

The full 2PC protocol is also shown in Figure 3. When the client commits, the request is immediately passed along to the `TMClient` (the TM in the MW layer is considered client-side relative to the RM). The `TMClient` must then ask the RM to prepare to commit. The `TMServer` handles this by first preparing a new shadow copy and then updating the transaction log. Once all RMs have successfully prepared to commit, the `TMClient` sends the commit command to each RM, at which point the shadow pointer and transaction logs are updated. Finally, the locks are released at the MW.

3 Reliability

3.1 Time-outs

A timer is maintained for each transaction in the system both in the MW and the RM. In writing this report, we realize that the RM timer might be problematic as a transaction may be quite active while going a long time without interacting with one of the RMs. However, in the case of a TM crash it may be preferable, as

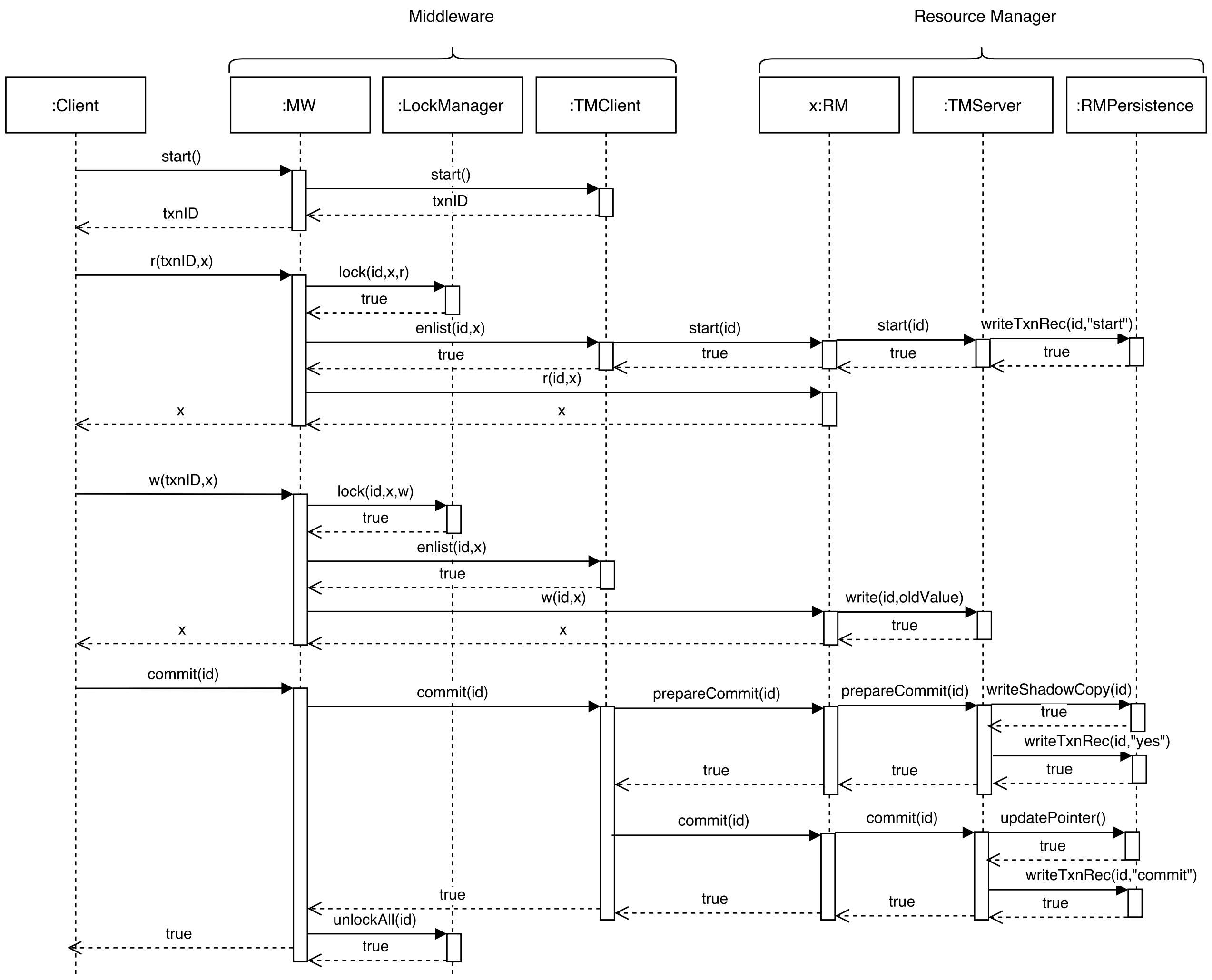


Figure 3: A sequence diagram depicting a successful transaction with read and write operation.

was suggested in the lectures and project specification, to abort the transaction after a time-out at the RM. A more realistic design would have to consider how to ideally calibrate the two time-out periods.

3.2 Persistence

There are several actions that must be taken in order to ensure persistence when a RM node crashes. First, the persistence layer (PL) copies the shadow copy over from the current safe version to the new location. Next, the changes made by the transaction being prepared are applied to the new shadow copy. Finally, the log is upated. Our implementation is inefficient (even for shadow copying!) because we decided to overwrite data for all transactions in main memory for the second milestone. As a result, the copy in main memory contains changes that should not necessarily be persisted. As a result, we read the entire shadow copy into main memory and then write it back out, and then apply only the changes for the prepared transaction. The TM already has a list of all items written by a transaction because rollback information is being stored. The keys are retrieved from the rollback list and then the values for these keys are retrieved from the table in main memory (only the the rollback keys and *not* the values are used).

3.3 RM Recovery

We implemented recovery for the RM according to Algorithm 1. The state for a transaction is restored if the log shows that it has voted but not received a reply. Otherwise, any transaction that has started is aborted. The RM then sends a no vote when the TM asks to prepare an aborted transaction. This implementation performs a bit of extra "house-keeping" at startup to ensure that all other actions performed by the RM need not be aware of whether the system is in a crashed/recovered state.

```
Upon startup do
    Load shadow table into memory
    Check the transaction log
    if transaction voted but did not commit or abort then
        Restore undo information for abort
        Restore uncommitted changes to table
    end
    else if transaction started but did not vote then
        Write abort to transaction log
    end
end
```

Algorithm 1: Recovery procedure for RM.

Unfortunately, the TM blocks (i.e. stays in the critical commit section) until the RM is back up after a crash. Ideally, the commit could be moved to a background operation and other transactions which do not require the downed RM could still execute.

3.4 Implementation Crash Simulations

The web-services interface makes it difficult to implement automated unit testing with JUnit and to debug across the MW/RM interface. For this reason, we devoted considerable energy in the final milestone to implementing an efficient testbench to explore our test scenarios. For this reason, we consider our implementation to be a bit simplistic but considerably more robust. Crash locations were assigned identifiers in both the classes `TMClient` in the MW layer and `ResourceManagerImpl` on the RM side. Crashes are created by notifying the TM or RM of a crash location prior to executing the command which should crash. The following crash scenarios are defined for the TM:

1. crash before sending vote reply
2. crash after sending vote reply to 1 RM
3. crash after processing all commits
4. crash after sending one vote request

and for the RM:

1. crash before aborting
2. crash before committing
3. crash before voting

The class structure for the crash testbench is shown in Figure 4. Different implementations are provided for the actual crash event. For our testbench, we wanted crashes to simply throw an exception during execution without exiting the process. We wanted to only *simulate* the crash to observe behaviour of the nodes that do *not* crash in our initial testing for correctness. The `TestBench` class can therefore instantiate a test version of the MW/RM interface (`MWTestClient`) which directly references `ResourceManagerImpl` without using web services. Most bugs can be solved more efficiently when bypassing web services. Only once all tests pass do we start up the full system and run the same scenarios with `MWClient`, which accesses the RM using web-services through the inherited `WSClient` class (also depicted).

Consider an example where the RM crashes and the TM must take appropriate action. We would like to instantiate the TM and RM in the same process in a testbench. A simulated crash (`TestRMCrash`) will cause the RM to only throw a `CrashException` which propagates back to `TMClient`. `TMClient` will catch the exception and take appropriate action. When using web-services, the RM uses `WSCrash` which exits the program. Meanwhile, the crash manifests itself in the same way for `TMClient` because `MWClient` catches the socket exception and throws a `CrashException`.

The efficiency of this testing infrastructure is shown in Figure 5. A test of each crash scenario can be carried out in 13 seconds. By contrast, running a single scenario with web-services can take several minutes (as seen in the demo). Also, the testbench can check assertions at various points in each scenario for all objects in the system (i.e. state of objects in MW and RM) to directly ensure correct execution. Furthermore, the `Client`, `Middleware`, and `LockManager` classes, can be bypassed as they have no real impact on the sub-system being tested.

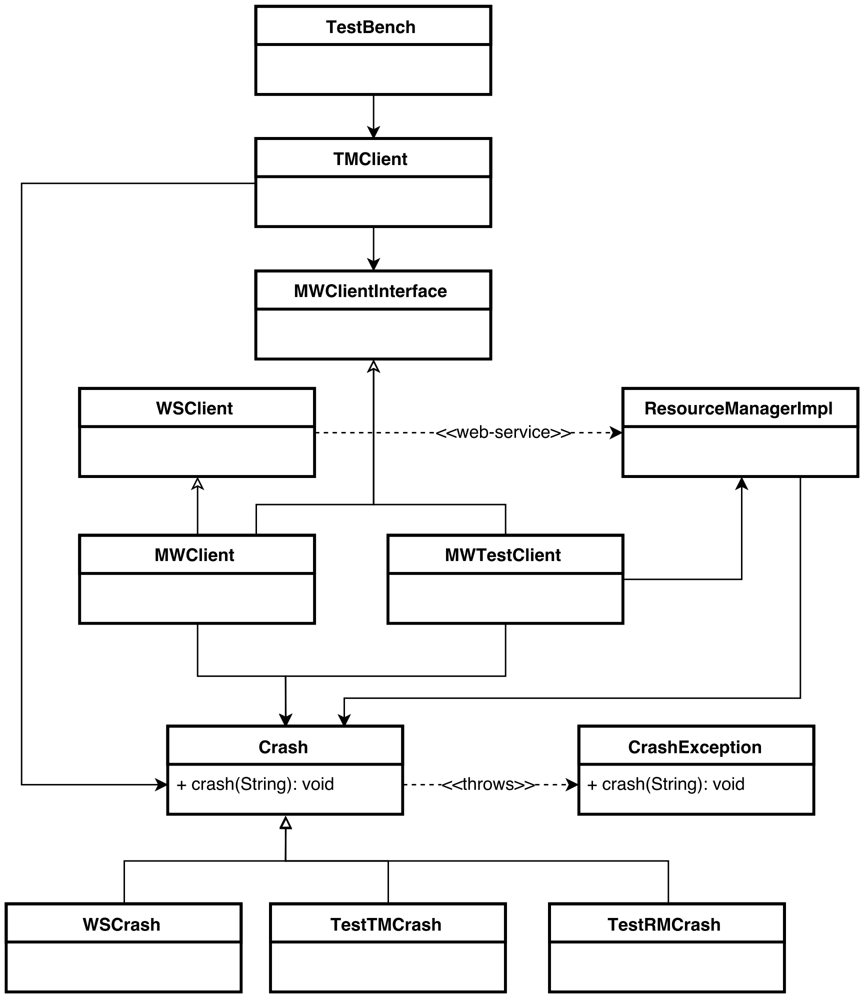
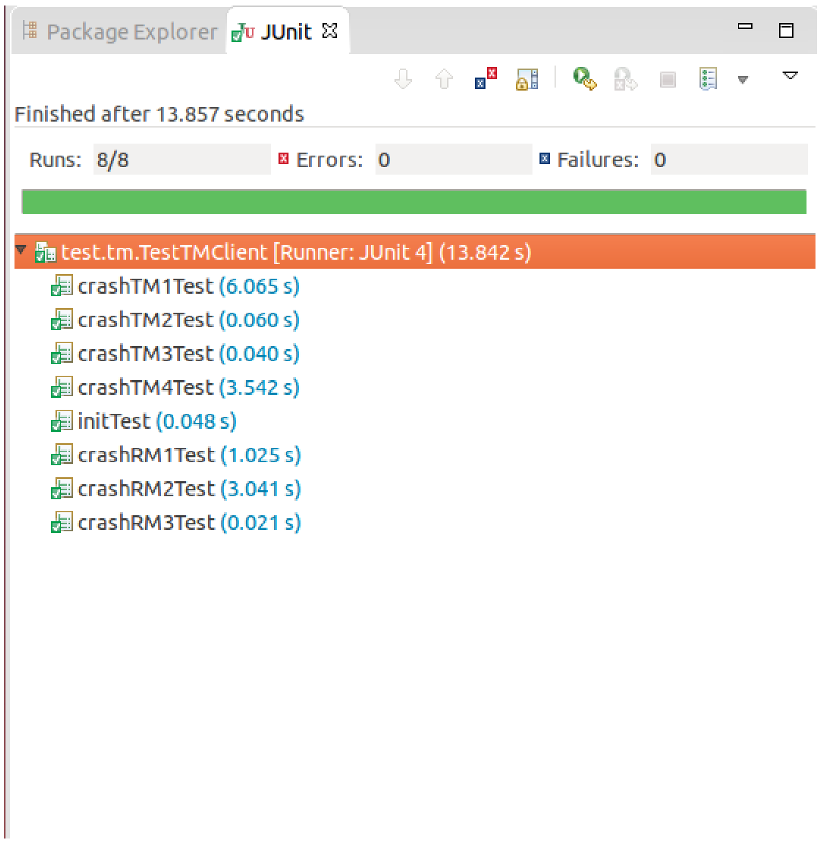


Figure 4: Class structure for crash testbench



```
Finished after 13.857 seconds

Runs: 8/8    Errors: 0    Failures: 0

test.tm.TestTMClient [Runner: JUnit 4] (13.842 s)
    crashTM1Test (6.065 s)
    crashTM2Test (0.060 s)
    crashTM3Test (0.040 s)
    crashTM4Test (3.542 s)
    initTest (0.048 s)
    crashRM1Test (1.025 s)
    crashRM2Test (3.041 s)
    crashRM3Test (0.021 s)
```

Figure 5: Test results

4 Performance Results

For all experiments, the clients are located on one computer, the middleware on a second computer, and all the RMs are located on a third computer. Experiments were conducted without persistence and 2PC.

4.1 Single client response time

The transaction for a single RM consisted of issuing the *newcar* and *deletecar* commands three times each. The transaction for all three RMs consisted of creating and deleting each type of resource on each RM. We measured 5000 transactions and found that the single RM transaction took an average of 42ms while the multiple RM transaction took 48ms.

The computations on all resources are fairly lightweight. We did not attempt to measure them explicitly for this assignment. However, since the test consists of writing values to a hash table it seems fairly probable that all that all the computations and thread switching overhead accounts for at most a dozen milliseconds. It would seem like the overhead is largely due to the network. However, there is one overhead that is not neglected when calculating response time which is not trivial: printing log information directly to the terminal can cause create sizable execution overhead. Unfortunately, the code we have been provided does not have adequate logging facilities to quickly change the `PrintStream` destination to a file or to completely disable logging altogether. The inclusion of logging (essentially a high overhead debug mode) while doing profiling may distort the results.

4.2 Multiple client response time

Each client in the experiment commits a transaction at regular intervals t according to Eqn. 1 where N is the number of clients and $r \in [-1, 1]$ is a random number. Any time spent in retrying an aborting transactions due to deadlock is counted towards the response time of that transaction. All clients are competing for the same resources in this experiment.

$$\left(\frac{1000}{tps} \cdot N\right) \cdot (1 + 0.3r) \tag{1}$$

Figure 6 shows the performance results for the multi-client system. The system gradually approaches saturation when number of clients is less than 22. The saturation point appears at roughly 30 tps. For 22 clients or more (we tested for more but have omitted the results from the graph), the system is immediately saturated regardless of the number of clients. As expected, the response time generally increases as the number of clients in the system increases. Once the system is fully overloaded, then the saturation point stops increasing. In fact, the 22 client system appears to outperform the 18 client system (probably due to some strange timing in the 18 client trial).

Figure 7 shows the same data but now with the number of transactions per second held constant and the number of clients increasing. We see that once 25 clients are reached the number of transactions per second is no longer relevant, implying that the system is completely overloaded.

There are two main explanations for these performance figures. First, the middleware eventually becomes a bottleneck for the system once too many transactions arrive at the same time. Second, since the clients are competing for the same resources it forces a largely serial execution at the system level. Both performance figures show that the system is saturated in the 25-30 transaction range (either 30 clients submitting 1 transaction or 1 client submitting 30 transactions). However, that the response time for 1 client submitting 30 transactions is so much lower while the overall network traffic remains constant, implies that the overhead of context switching in the middleware is considerable.

Another minor point to note in both figures is that a single client performs *worse* on average at 1tps compared to 4tps. One explanation could be that there is some implementation detail of the underlying web services that creates an overhead after a certain short amount of time has passed (e.g. releasing a thread of more than a second elapses). It is difficult to say without knowing more about the web services.

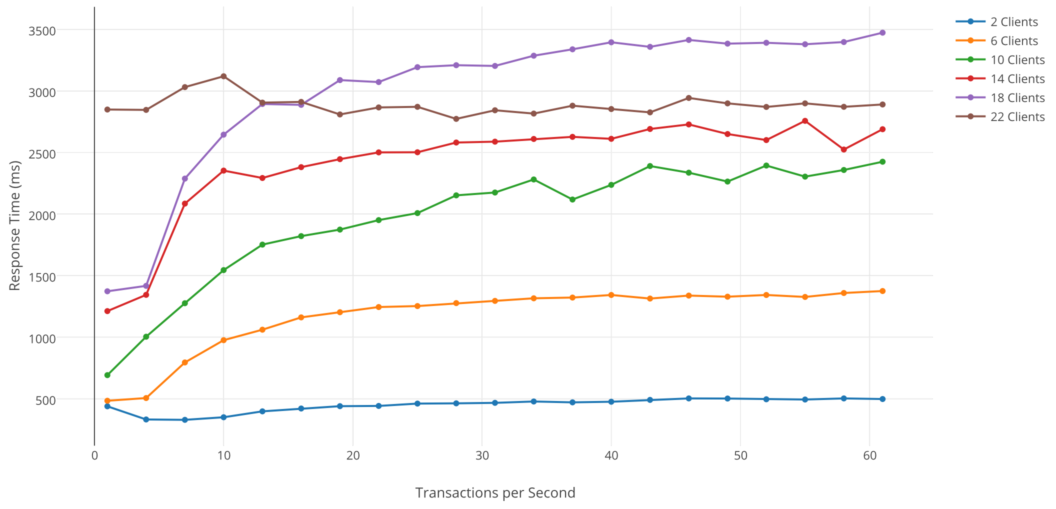


Figure 6: Average response time as transactions per second increases while holding the number of clients constant.

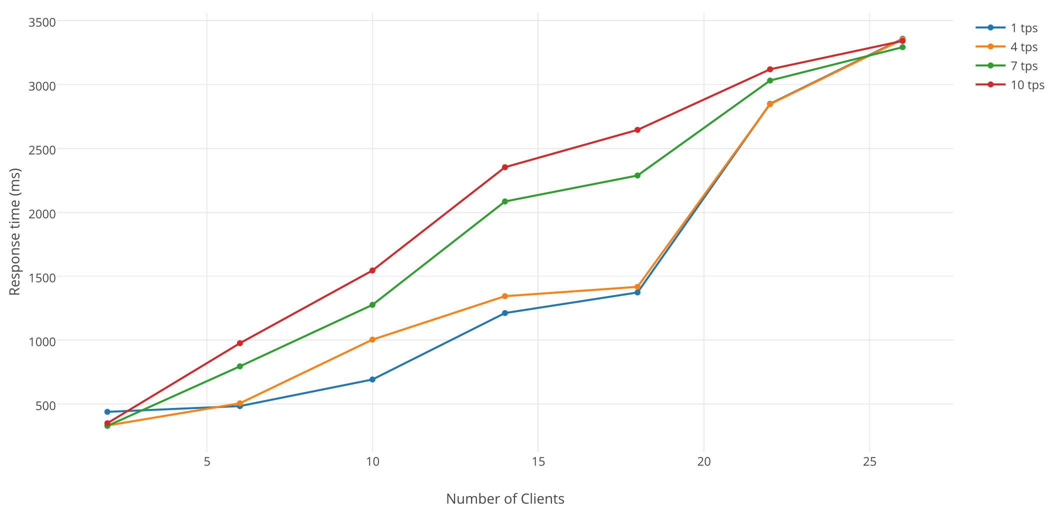


Figure 7: Average response time as number of clients increase while holding the number of transactions per second constant.

5 Conclusion

We have implemented a simple distributed reservation system that demonstrates key concepts in distributed system design including transactions, data persistence, and node recovery. Given more time, we would have liked to implement the system with TCP as this would have enabled two way communication at the RM/MW interface as well as simplified the testbench development for the final milestone. The problem with TCP is that the initial overhead of recreating a robust protocol for the pre-existing interfaces was simply too high given the time constraint for the second milestone. In general, we have been made aware of the complexities that can arise when designing a distributed system and the inefficient performance of easy-to-implement solutions.