**COMP 512 project report (final)**

General architecture

Our architecture is as follows:



The following points are of interest.

*RM customer*

When centralizing the management of customers for Part 1, we had to decide whether to put it in a separate RM or at the middleware. We decided to use a separate RM as the RM interface already supported customers and it made the middleware simpler (no synchronization was required). We only had to change the handling of some commands related to customers. When a customer is created, we only create it at the customer RM. Then, for any command that references a customer, we execute it 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 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.

The deletecustomer command is always executed on all RMs.

*TM/LM*

We decided to have the transaction and lock managers at the middleware. For the lock manager, we decided that dealing with distributed deadlocks would be fairly complex. Furthermore, we had a lot of logic in the RM at that point, while the middleware was little more than a relay. Therefore, it appeared easier to plug the TM and LM in the middleware. This has the advantage of reducing the message overhead in the system, at the cost of having a slightly less flexible architecture.

Components

*Automated client*

In addition to the interactive client, we have an automated client that is designed to run stress tests on the system. Given a file of transactions, it executes a certain number of them, by either picking randomly or sequentially (with looping), and by respecting a specific rate of transactions per second. The transaction file may contain transactions like the following:

%1,start

%2,queryflight,%1,767

abort,%1,%2,==,0

reserveflight,%1,1,767

commit,%1

This example shows a transaction that checks if a flight has available seats, and if yes, books one. The key points of the syntax here are the use of registers (%1, %2, etc.) to save command results which can be reused in subsequent operations in the same transaction. Note the condition on the abort: this is tested purely on the client side. If the condition is true, a “normal” abort command will be sent to the middleware.

*Message handlers*

We used a web-service implementation for parts 2 and 3 (which was horribly misguided, but we had to stick with it). One thing that we noticed was that almost all methods had some pieces of code used to check that the transaction ID was valid and to measure performance. This code was moved into a SOAP message handler which was then installed at the MW and at the RM. Specifically, there are two separate handlers, one for performance and the other for transaction validation.

The transaction validation is only installed at the MW (the RMs assume that transaction IDs sent to them are valid). It performs a superficial parse of the message contents to determine if a transaction ID is included, and if yes, whether that transaction ID is valid. If it is not valid, then the handler cancels the message delivery and returns an appropriate error to the client. If the transaction ID is valid, the handler sends a heartbeat message to the involved RMs so that they know the transaction is still active (see the TM section below).

The performance handler logs the amount of time taken for each method call, in microseconds. Note that as the handler operates on the SOAP data, the performance figures take into account the time need to (de)serialize the data.

*Middleware*

The middleware is responsible for routing the messages between the RMs and the clients. To avoid creating a whole new proxy object for every request, it uses pools of proxies (one pool per RM) which are constructed at initialization time. Each pool maintains its available proxies in a queue that it synchronizes access to, providing ways of checking proxies in and out. A semaphore (counting the number of objects in the queue) is used to make threads wait for their proxy as necessary. We considered managing the pools from the handler, for the sake of code correctness (it wouldn’t be possible to forget to check in a proxy) and beauty, but decided against it. Doing so would have made the handler more complex and more like an actual message executor, and would have prevented fine-grained management of proxies in the methods. For instance, methods that call several RMs only check out a proxy for the duration of the call to its RM, which reduces the pressure on the pools.

*Transaction manager*

The transaction manager is a singleton living on the middleware, created at middleware start-up. It is responsible for tracking the results of transactions, their time-outs and which RMs are participating in which transactions, depending on which commands are received. It also allocates new transaction IDs as new start commands are received. The access to the internal structures is restricted by means of synchronized methods.

The transaction manager is also in charge of aborting transactions that do not receive any activity in a sufficient amount of time. This is simply achieved by using a timer that is reset (as mentioned above, by the handler) at every new operation of the transaction.

*Lock manager*

The lock manager has not been modified extensively, only to add lock conversion. As explained before, it resides on the middleware. Our locking algorithm is simply 2PL.

*RMs*

The RMs manage data in a pessimistic way. That is, they do not write to the master table (even the in-memory one) until a commit actually occurs. Instead, they write new versions of the data in a temporary queue (on a per-transaction basis). Any operation always checks this queue first to be sure to operate on the latest data visible to the transaction. It is only at commit time that the new information is written to the master table. If a transaction aborts, the temporary queue is simply discarded.

The RMs also implement a timeout mechanism, in case the middleware fails to carry out its part of the protocol in a timely manner. This timeout is slightly higher than the one for the transaction time to live at the TM, but not by much. Indeed, for any operation on a transaction, the TM sends a heartbeat to every participating RM. There is thus no risk of a transaction expiring only on one RM because it was not used much in the transaction.

2PC

Our system uses 2PC to guarantee data integrity. When a client wants to commit a transaction, the middleware begins the 2PC procedure with all the participating RMs. Note here that a participating RM is one that has pending write or delete operations. Those that were only read from do not participate, as they have nothing to do. The commit protocol takes place as follows:

* First phase: upon receiving a commit request from the client, the middleware sends preparation requests to the participating RMs. Each RM writes the modified objects to disk in a separate file, and returns a confirmation. Failure could occur because of I/O, or because the RM does not know about the transaction that it is asked to prepare. Note that because the RM is pessimistic, there is no need to additionally write the current version of the objects on disk (those in fact already written in the master file).
* Second phase: when all the results are received, the MW decides whether to abort or commit the transaction, as per the 2PC protocol. It sends the appropriate action to the RMs and returns the result to the client. The RM then follows the action. In case of an abort, it discards the queue of temporary operations. In case of a commit, it writes the temporary operations into the master table, and saves it on disk. In both cases, it then deletes the now unneeded preparation file. Saving the master table is done by first writing the updated table to a new file, deleting the old one, and then renaming the new one. This ensures that a crash in the middle of writing the data will not compromise the data integrity.

*Logs*

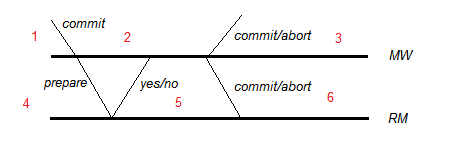
We log information at the following points (chronological order), including the transaction ID and when relevant, the decision:

* MW: RM joins a transaction as participant
* RM: write/delete operation
* MW: start of 2PC
* RM: decision taken (after data is on stable storage)
* MW: decision taken (after all votes have been received)
* RM: commit completed
* MW: commit completed

These logs are essential for us to perform the recovery protocol described below

*Recovery protocol*

Both the RM and MW implement a recovery protocol in case they crash during execution. It involves looking at the logs and deciding what to do based on the information available. For crashes on only one “side” (either the MW crashes, or some/all the RMs crash, but not both), we consider the following six cases:



1. *The middleware crashes before a commit/abort request is received from the client*. Here, we should be able to recover in the following way: verify that all the participants are still active (they did not timeout the transaction), re-acquire the locks on the objects of the transaction, send a heartbeat to the participants and resume normal execution. It is the second item that proves to require more work to implement, as we need to log the locks used on a per-transaction basis. Therefore, we did not implement this part of the recovery, so the middleware simply aborts the transaction in this case (we do log the participants to each transaction, which enables us to do the abort).
2. *The middleware crashes after receiving a commit, but before deciding whether to commit.* We do not know what the decision should have been, and maybe some RMs voted no and already discarded their changes. Therefore, we again abort the transaction. Here, too, we could potentially ask the RMs to re-send their decision, or ask them to prepare again and see whether it goes through. However, in the interest of keeping the recovery protocol rather simple, we do not do this.
3. *The middleware crashes after deciding on the transaction*. We logged the transaction result, so we can simply re-send this decision to the RMs. Here we do not care about the result of that operation, as some RMs could have already committed their operations. If the crash occurred after the end of the transaction, we have nothing to do.
4. *The RM crashes before sending its decision*. In this case, when restarting, it signals to the MW that it crashed, and to abort the transaction. Indeed, as long as there is no log entry saying that a decision was sent, we cannot know whether the data was prepared to disk properly. Therefore, we have to abort.
5. *The RM crashes after having decided*. The actions depend on the decision, which is obtained from the logs. If the decision was abort, then nothing needs to be done. If the decision was commit, then the RM contacts the MW and asks for the status of the transaction. The MW can then check the TM and send the appropriate result back, at which point the RM continues the protocol “normally”.
6. *The RM crashes after having committed*. Nothing to be done here!

Note that all these cases have a direct correspondence to the cases outlined in the project spec, as outlined in the following table:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Spec** | MW1 | MW2 | MW3 | MW4 | MW5 | MW6 | MW7 | MW8 | RM1 | RM2 | RM3 | RM4 |
| **Us** | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 5 | 5 |

If we are extremely unlucky and crash in the middle of the recovery, we should still be fine. We do not delete the old logs until the recovery is complete, so we can just redo everything when we restart.

We did not handle cases where both the MW and a RM would crash during 2PC, or cases where a crash combined with an extremely long message delay causes inconsistent state. For a discussion of two such examples and how they would have to be handled, check the appendix.

Performance measures

We collected performance measures with one client, multiple clients, and with the system on the same or different machines to measure the network impact. Time for each operation was observed at the client, MW and RMs. The logs and spreadsheets containing the data below can be found in the *analysis* folder.

Table 1 summarizes the results for one client on a *local* run (so there is no network overhead). We used the autoclient scenario files *onecli\_onerm.txt* and *onecli\_manyrms.txt* in the *autoruns* folder.

Table 2 shows the per-transaction results on a *remote* run (client on one machine, MW on another machine and all RMs on another machine). The same scenario files were used.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Op. type** | Client | | MW | | RM | |
| One RM\* | All RMs | One RM | All RMs | One RM | All RMs |
| Full txn† | 60.18 | 95.96 | 51.88 | 83.57 | 14.46 | 28.53 |
| Start | 1.4 | 2.38 | 0.54 | 0.71 | - | - |
| Commit | 17.67 | 39.01 | 16.61 | 37.25 | 1.05 | 1.4 |
| Prepare | - | - | - | - | 1.4 | 1.71 |
| Heartbeat | - | - | - | - | 0.6 | 0.83 |
| Others | 38.46 (tot.) | 51.94 (tot.) | 34.71 (tot.) | 45.61 (tot.) | 0.93 (each) | 1.18 (each) |

*Table 1. Average times of execution over 1000 transactions in milliseconds, with one client and all components on the same machine.*  
\*One RM + customer †6 operations / transaction (incl. start/commit)

One obvious data point we can see is the large amount of time taken at the MW, where there is significant overhead compared to the RM. This could be explained by the following:

* Extra SOAP handler to run (transaction validation).
* More shared data structures to synchronize against (transaction manager, lock manager, proxy pools).
* More messages (querying customer existence, sending hearbeats)

In particular, for the last point, a commit from the point of view of the MW will involve sending several messages to the RMs in order to perform the coordination. Notably, we do not proceed in parallel but rather execute the commands in sequence. That could explain in large part the performance hit (this also applies to normal commands, since we send heartbeats in sequence as well). One way to alleviate this would be to replace the proxy pools with thread pools that would allow to start multiple requests in parallel and then to collect their results at the end of the execution. The use of a pool would still be mandated to avoid thread creation overhead.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Client | MW | RM |
| One RM | 358,92 | 289,97 | 141,48 |
| All RMs | TODO | TODO | TODO |

*Table 2. Average times of execution over 1000 transactions (6 ops/txn) in milliseconds, with one client and components on a LAN (RMs all on the same machine).*

EXPLAIN

SECTION ABOUT MULTIPLE CLIENTS

Correctness tests

For the first two milestones, we did manual tests to make sure our code was behaving properly.

For the third milestone, correctness tests were done in two ways. The first way was “integration testing”, where we devised autoclient scenarios to test mainly the crash scenarios during 2PC. We used our crash-on-command facility, implemented similarly to the spec (see the technical section at the end for the details). We then ran the scenarios, manually restarted the crashed components and checked the output and data to ascertain that the proper behavior had occurred. The second way was to check the logs and behavior of the system during stress testing and data collection for the performance section. These tests indicated that the command execution was behaving properly (except for the irregular problems mentioned below).

Problems

Using web services was unambiguously the worst decision we made of the entire project: we had to battle with them to do things that would have been much simpler with TCP. Nearly all the problems we faced can be traced to web services in one way or another:

* Common treatment for all messages (logging, validation); we had to figure out how to install a SOAP handler, which was arduous as the documentation for that was sparse and very toy-like (so difficult to apply to our project). Getting this to work was just a matter of googling a lot and trying things out until it worked.
* RM -> MW communication; this was required for the RM recovery protocol. The problem is of course that to communicate with the MW, the RM needs the MW’s wsdl, while the MW needs the RM’s wsdl: both must be compiled before the other. We did not actually find a proper solution to this and resorted to the hack of constructing and sending the SOAP XML ourselves over HTTP (which was okay since the requests and responses were simple).
* Large amount of memory used; near the end of the project, we realized that all the components were using a very large amount of memory (over 300mb at startup, and climbing steadily with the number of transactions). Despite doing some memory profiling with Eclipse MAT and VisualVM, we could not figure out where that came from.

We strongly encourage the instructor and TAs to warn future students of how maladapted JAX-WS is to this project, so that they can avoid the pain we had to go through.

LOCKS?

Implementation details, technicalities and systems tools

TODO

Appendix: weird crashes we didn’t handle

*Double-crash at RM and MW*

During a 2PC, the RM crashes after sending a yes vote and the MW after taking the decision to commit but before sending this decision. The MW recovers first. According to its protocol, it resends the decision to the RMs, but the crashed RM is still down and the message is lost. Then the RM goes back up, sees an uncertain transaction, and asks the MW for the status. Since the MW crashed earlier, the TM has no information for that transaction. In that case, the default is to return abort. Therefore, the data at the RM is lost.

**Possible solution**: When the MW starts up, it keeps track of a set the transactions that could not be completed (we can reliably determine that the crashed RM did not receive our decision resent) and their decision. The TM is instructed to not use those transaction IDs. When a RM asks for the status of a transaction, we check that set first and return the appropriate status.

*Extreme delay at MW combined with RM crash*

During a 2PC, the RM crashes after sending a yes vote. When it restarts, it asks the MW for status. However, the MW is still awaiting the reply from another RM. Therefore, the TM has no information for that transaction, so abort is returned. But then the slow RM finally votes yes, and the MW decides to commit after all. However, the data at the crashed RM is already lost.

**Possible solution**: The TM should know that a transaction is in an “undetermined” state. Checking for the status of an undetermined transaction would block until a decision is made.