School of Computer Science, McGill University

COMP-512 Distributed Systems, Fall 2015

Project Part III: Reliability and Fault Tolerance

Due: December 4, 2015; 17:00
Demos: November 27, December 3/4

You can present this project ANYTIME before December-4. I suggest a week before but leave the flexibility for everybody to accommodate other deadlines. There are two choices.

1 2PC (choice 1)

Implement the two-phase commit (2PC) protocol. The TM is the coordinator of the 2-phase commit. You can assume that communication is reliable (messages never get lost) since you use reliable web services. Since the client only connects to the middleware server and is not aware of any distribution, its available interfaces do not change.

1.1 Tasks

1. Make your data persistent by using shadowing. Each resource manager should have two files for the two versions of its data and a master record which should all be located in a specific directory for this RM. At startup of the RM, if the master record file exists, it automatically recovers, otherwise it creates new files and starts as a new RM.

2. Code the basics of 2-phase commit. That is, implement commit and abort under the assumption that sites do not fail and messages do not get lost or excessively delayed. Generate appropriate log information at appropriate places. In particular, you have to decide how you write information to stable storage such that in the *prepared* state, a transaction can still either abort or commit.

3. Now consider site failures.

   a.) Enhance the system so that you can simulate the failure of any relevant component (middleware, TM (if standalone), RM).

   b.) All errors and exceptions occurring at other sites due to the failure of some site should be handled correctly (at client, at middleware, at TM (if standalone), at RM). You might have to detect failures via timeout. Implement this accordingly. The only case where a transaction should block indefinitely is when a participant has voted YES and does not receive any message from the coordinator. In all other cases such transactions should eventually abort and resources should be freed. You do NOT need to implement a termination protocol where sites asks around when they are in doubt.

   c.) Note that your implementation should also be able to handle incorrect timeouts. For instance, a participant times out while waiting for the VOTE-REQUEST and hence aborts the transaction, but then receives the VOTE-REQUEST from the coordinator. That is, each site should be able to handle requests it does not expect.

   d.) Implement the recovery procedure for either the middleware or an RM.

1.2 Interface Requirements

1. At middleware server:

   - `public void crash(String which);`

     which indicates which server should destroy itself (e.g., FLIGHT, CAR, TM, etc.). The middleware then calls the `selfDestruct` method of this server which forces the server to exit.

2. Wherever needed and appropriate:

   - ```
     public boolean prepare(int transactionId)
         throws TransactionAbortedException, InvalidTransactionException;
     ```

     which initiates the first phase of 2PC.

   - `public void selfDestruct();`

     which forces the server to exit.

3. You should have a version of your system that shows the correctness of the 2PC implementation. In order to test, one has to artificially force the crash of sites. In your demonstration at the end of the project, you should be able to show the behaviour of some of the following cases:

   - At the TM (coordinator):
     - Crash before sending vote request
     - Crash after sending vote request and before receiving any replies
     - Crash after receiving some replies but not all
     - Crash after receiving all replies but before deciding
     - Crash after deciding but before sending decision
     - Crash after sending some but not all decisions
     - Crash after having sent all decisions
     - Recovery of the coordinator (if you have decided to implement coordinator recovery)
   - At the RMs (participants)
     - Crash after receive vote request but before sending answer
     - Which answer to send (commit/abort)
     - Crash after sending answer
     - Crash after receiving decision but before committing/aborting
     - Recovery of RM; especially if it voted yes for a transaction but has not yet committed it.

   You somehow must be able to enforce such cases. For instance, run the 2PC in a cycle requesting (yes/no) input from the client to any of the situations above. With this, you can enforce the failure cases through corresponding input from the client. Alternatively, one can have a config file that contains the crash alternatives. The managers read the file before starting the 2PC and crash/behave accordingly. Or, alternatively, one can enforce specific behaviour through additional methods that are called after the transaction started but before the commit. For instance, the method

   ```
   public void setDieAfterPrepare(String which);
   ```

   helps to kill the server indicated in `which` after the vote but before the commit/abort decision. This can be implemented as an asynchronous method of the TM. When executing this method the TM simply sets a local variable, e.g., `serverToKillAfterPrepare`, to the value of `which`. Then, at the appropriate time during the execution of the 2-phase commit, the TM checks whether the variable is set, and if yes, to which server it is set. If it is set, the TM calls the `selfDestruct` method of the corresponding server.

4. In your description, indicate how you extended the shadowing or integrated logging so that you have enough information on stable storage for recovery.
   ```

2 Replication (choice 2)

Use replication to make your system fault-tolerant. The crash of no single component (middleware, individual RM) should make the system unavailable. Data consistency should be maintained at all times. You could use an existing group communication system (JGroups is a group communication system that is easy to work with and used widely), or try to find a Paxos library. But you can also use your own mechanism or do it via web service interfaces. Ideally, this should also provide some scalability for read operations.

Your system needs to handle all kinds of crashes at different time points. For that you should be able to artificially inject crashes at certain time points in the execution. Read under 2PC some failure scenarios that are relevant for distributed commit. Think about similar scenarios for replication. You only have to consider one crash at a time.

For this choice you have to think about what correctness really means. Be thorough in your crash cases that you consider, such as the middleware crashes or any of the RMs crashes, the crash occurs in the middle of a transaction (before the commit is submitted), after the commit confirmation has been returned to the client, or in the middle of the commit execution (request was just at the middleware, has been forwarded to some/all resource managers but not yet back at the middleware, etc.).

3 Deliveries

The final delivery is as follows:

- An online demonstration of your system. This demo is at the same time something like a project presentation of the ENTIRE project although the focus will be on the last delivery. It will take around 30 minutes. The last project has more weight than the others due to the higher expectations for this project demo and the project report. Start your presentation with a description of the different components of the system and how you implemented each of the tasks. Describe the architecture and some major design decisions. Some slides/figures would be very useful and are HIGHLY recommended (i.e., counts towards your grade). Your system should be running when the demo starts. You should prepare some typical example scenarios. The example scenarios should show the general execution of transactions, distributed transactions, locking, failures, the socket implementation, etc. The TA will ask questions throughout the demo so it probably will be an interactive presentation. Include enough informative output (to standard output or to a log file), so that during the demo it is convincing that your system does what it is supposed to do.
- The source code in form of a tar/jar zipped file.
- A project report containing:
  - The general design and architecture of your system
  - Performance results from second deliverable
  - Data design
  - How each of the individual features of the system is implemented
  - Special features of your implementation
  - Algorithm in case of optimistic concurrency control
  - Problems that you encountered
  - How you tested the system for correctness
  - You can include the project descriptions of the previous milestones in your final report. But the final report should be one concise paper.