CS/CE/TE 6378: Project I

Instructor: Ravi Prakash

Assigned on: February 2, 2012
Due date and time: February 16, 2012, 11:59 pm

This is an individual project and you are expected to demonstrate its operation to the instructor and/or the TA.

1 Requirements

1. Source code must be in the C /C++ /Java programming language.
2. The program must run on UTD lab machines (`net01, net02, ..., net50`).

2 Client-Server Model

In this project, you are expected to use *client-server* model of computing. You will need to know thread and/or socket programming and its APIs for the language you choose. It can be assumed that each process (server/client) is running on a single machine (`netXX`). Please get familiar with basic UNIX commands to run your program on *netXX*.

3 Description

Design and implement a distributed system which consists of *three* server processes and *five* client processes. Assume that each file is replicated on all the servers, and all replicas of a file are consistent in the beginning. A client can perform a READ or WRITE operation on the files. READ operation involves only one server that has the target file and the server is chosen randomly by a client. WRITE involves all servers that have a copy of the target file and all of them should be updated in order to keep the replicas consistent. READ/WRITE on a file can be performed by only one client at a time. However, different clients are allowed to concurrently perform a READ/WRITE on different files. In order to ensure this, implement Lamport's mutual exclusion (M.E) algorithm so that no READ/WRITE violation could occur. The operations on files (hosted by servers) can be seen as *critical section* executions.

To host files, create separate directory for each server. The servers must support following operations and reply to the clients with appropriate messages –

1. ENQUIRY : A request from a client for information about the list of hosted files.
2. READ: A request to read last line from a given file.
3. WRITE: A request to append a string to a given file.

The set of files does not change during your program's execution. Also, assume that there is no server failure during your program's execution.

The clients must do the following –

1. Gather information about the hosted files by querying the servers and keep the metadata for future.

2. Append a string $< client\_id, Timestamp >$ to a file during WRITE. Here $client\_id$ is the name of the client, and $Timestamp$ is the value of the client's local clock when the write request is generated. This must be done to all replicas.

3. Read last line of a file during READ.

Instead of submitting READ/WRITE requests to the servers through a command line interface, write an application that periodically generates READ/WRITE requests for some file(s).

Pertaining to socket programming, close all open sockets when your program exits/terminates. This may save you some valuable time during the test runs and more importantly, from a panic situation during the project demonstration.

Displaying appropriate log messages to the console or to a file is very important for testing, debugging and analyzing the correctness of your program.

4 Submission Information

The submission should be through WebCT in the form of an archive consisting of:

1. File(s) containing the source code.
2. The README file, which describes how to run your program.

NOTE: Do not submit unnecessary files.

CS/CE/TE 6378: Project II
Dynamic Replication

Instructor: Ravi Prakash

Assigned on: March 9, 2012
Due date and time: March 28, 2012, 11:59 pm

This is an individual project and you are expected to demonstrate its operation to the instructor and/or the TA.

1 Requirements

1. Source code must be in the C /C++ /Java programming language.
2. The program must run on UTD lab machines (`net01, net02, ..., net50`).
3. All processes (Server/Client) must run on different `netxx` machines.

2 Description

Design and implement a distributed replication system which consists of N servers with IDs $1, 2, 3, ..., N$. Server i is a predecessor of server $i+1$ and server $i+1$ is a successor of i. The server with maximum id is called *tail*. Server i has a logical clock (C_i) which is incremented by 1 on receiving a request. Initially, servers are hosting consistent copies of some objects (files). Also, there are M clients in the system that send READ or WRITE requests on some replicated objects. A client sends requests, one at a time, to a randomly selected server. For a particular request, a server that received the request first is called *head*.

You can assume that all communicating channels are FIFO and reliable, and there is no node failure.

On receiving a WRITE request from a client, a server (*head*) does the following –

1. It generates a locally unique *seq_num* for the request.
2. It forwards the request along with its *ID*, *seq_num* and C_i (for server i) to all servers but *tail*.
3. It also records the request into a local history. The request is treated as a pending request.

On receiving a WRITE request from another server, a server does the following –

1. It records the request into a local history as a pending request.
2. It augments the logical timestamp value with its current timestamp and forwards the request to *tail*.

On receiving a WRITE request r, the *tail* does the following –

1. If r is a new request to the *tail* then creates a vector timestamp, V_r.
2. Assigns the the received logical clock value to the corresponding entry of V_r. For example, if the request is received from server i then assigns $V_r[i] = C_i$.
3. If a vector timestamp for a request is complete (no more logical timestamp required) then select k^{th} pending request such that $\forall j \neq k, V_k < V_j$, where V_j are all other pending requests. For vector clock values that are mutually concurrent, the identity of the head is used as the tie-breaker. *Note that it is possible that none of the requests can be selected at this time.*

4. It commits WRITE for request k (if exists), sends ACK to the requesting client. Then sends a COMMIT notification to its predecessor.

On receiving a COMMIT notification from its successor, a server does the following –

1. Commits the request to local replica.
2. Deletes the corresponding entry from the local history.
3. Forwards the notification to its predecessor (if any).

On receiving a READ request, a server (*head*) does the following –

1. If there is no pending request for the given object, reads locally and returns the desired value.
2. Else, send a query to *tail* inquiring for the latest value of the object.
3. On receiving the updated value of the object, returns the desired value to the client.

The COMMIT (to file f_1) procedure consists of – appending `<ID, seq_num, C>` followed by the *message* to be written to f_1. An ACK of a WRITE request could be SUCCESS or FAIL. However, ACK of READs (from file f_1) must contain last written value to f_1. The READ requesting clients then log all read values locally.

Please read the paper [1] for more detailed description about the Dynamic Replication System.

3 Notes

Pertaining to socket programming, close all open sockets when your program exits/terminates. This may save you some valuable time during the test runs and more importantly, from a panic situation during the project demonstration.

Displaying appropriate log messages to the console or to a file is very important for testing, debugging and analyzing the correctness of your program.

4 Submission Information

The submission should be through WebCT in the form of an archive consisting of:

1. File(s) containing the source code.
2. The README file, which describes how to run your program.

NOTE: Do not submit unnecessary files.

References

[1] Guy Laden, Roie Melamed, and Ymir Vigfusson, "Adaptive and dynamic funnel replication in clouds," in *SIGOPS Oper. Syst. Rev. 46, 1, pp. 40-46*, February 2012.

CS/CE/TE 6378: Project III
Dynamic Replication

Instructor: Ravi Prakash

Assigned on: April 17, 2012
Due date and time: May 1, 2012, 11:59 pm

This is an individual project and you are expected to demonstrate its operation to the instructor and/or the TA.

1 Requirements

1. Source code must be in the C /C++ /Java programming language.
2. The program must run on UTD lab machines (`net01, net02, ..., net50`).
3. All processes (Server/Client) must run on different `netXX` machines.

2 Description

This project is a continuation of Project II. You will be implementing scenarios of failure handling on top of usual file replication operations.

In addition to replica server nodes, there is a *master* server that keeps track of any failure in the system. If a node fails, *master* notifies all other servers and clients in the system. The objective here is to execute clients' requests exactly once even if a server node fails. In order to do that, your program is expected to do the following –

Upon receiving a notification of a failure of a replica server, S_i (non-tail), from *master*:

1. If server S_j has an uncommitted (pending) request from S_i (head) in its local history, then it waits for $j \times RTT$ time before S_j becomes *head* of pending requests and forwards pending requests to all intermediate nodes. The intermediate nodes eventually forward them to *tail* following the usual process.
2. If S_j comes to know that some other node became *head* of a pending request within the waiting time, it discards the pending request.
3. If S_j receives a COMMIT notification within the waiting time, it deletes the corresponding pending request from its local history.
4. *Tail* discards any pending requests previously received from S_i.

Average RTT (round trip time) can be calculated at the beginning by taking average of RTTs of all neighbors [1]. All duplicate information should be discarded (if any).

Upon receiving a notification of a failure of *tail* from *master*:

1. The server with next highest *node-id* is automatically chosen to be new *tail*.
2. All other replica servers send their pending write request(s) directly to the new *tail*. The servers must send same information that it sent to old *tail*. It must send all pending requests.
3. Once new *tail* receives from all servers, then only it proceeds to COMMIT or handle any READ/WRITE request.

Please read Section 3.3 of the paper [1] for detailed description about fault tolerance in the Replication System.

[1] A sample program for calculating RTT will be posted on elearning.

3 Test Case

You are also required to implement a test case in order to verify the correctness of your program. The test case is described below –
Tail generates a COMMIT sequence (CSEQ) number to the write requests in the order in which they are committed. CSEQ is propagated along with commit-notification from *tail* to all other servers. While writing (committing) to local copy of the replica, each server appends CSEQ to the tuple `<ID, seq_num, C>`. So, new tuple is `<ID, seq_num, C, CSEQ>`. In case of failure of *tail* it should start CSEQ value with last known value and incrementing it by one. To test the correctness of the program you should check following –

1. All files must be consistent if there is no WRITE pending in the system.
2. A client should read lines from a file with non-decreasing CSEQ.

You can make a client just to send READ requests continuously and filter the output (REPLYs) for a particular file to verify this.

4 Notes

Pertaining to socket programming, close all open sockets when your program exits/terminates. This may save you some valuable time during the test runs and more importantly, from a panic situation during the project demonstration. Test your program on lab machines before submission.

Displaying appropriate log messages to the console or to a file is very important for testing, debugging and analyzing the correctness of your program.

5 Submission Information

The submission should be through WebCT in the form of an archive consisting of:

1. File(s) containing the source code.
2. The README file, which describes how to run your program.

NOTE: Do not submit unnecessary files.

References

[1] Guy Laden, Roie Melamed, and Ymir Vigfusson, "Adaptive and dynamic funnel replication in clouds," in *SIGOPS Oper. Syst. Rev. 46, 1, pp. 40-46*, February 2012.