CE 6378 Project 2 Design

# Introduction:

This document describes the design and implementation details for the CE6378 project 2. It covers the major components and design decisions in this project. It also discusses our assumptions and design tradeoffs made on this project.

# Overview:

The goal of this project is to implement a network based key/value storage solution of N servers. The objective is to guarantee that each object is stored on 3 servers when all the server nodes are working normally. When the system suffers network outages or server crashes, a write request should be accommodated by at least 2 storage servers before it is committed in the system.

Our project design consists following aspects:

1. Important data structures
2. The communication protocol design.
3. The server logic design.
4. The client and control utility design.

The various supporting data structures used through out the implementation are also described in the appendix.

# Important data structures:

### VectorTimestamp:

This is the data structure that implements vector timestamp. It also implements the total ordering of the storage time stamp for an object based its primary, secondary, tertiary server’s logic timestamp.

ContentObject:

This is the data structure used to represent the object in storage. It uses an integer value as key, and has integer and string data types as values. It also contains a 3-value timestamp to represent the primary, secondary and tertiary server’s logic timestamp when this object is stored.

### MessageObject:

This is the message data structure that was sent between clients and servers and among serves. The message object contains the type of messages, the sending server’s Id, the sending server’s logic time clock, the content object being sent etc.

# Communication protocol:

The core component of this project is a well-defined communication protocol. We categorize the messages being used in this system into following classes:

1. Server and client communication messages. These messages include the client’s request to read/write an object onto a server, and server’s response to the client.
2. Servers’ coordination messages. These messages include one server’s requests for its backup servers to store the same object, and the responses the backup servers will reply back.
3. Server’s control/status message. To partition servers into different groups, or to ask the server to simulate crash failure, we will use a command line tool to send messages to a server to cut itself off from other servers, or to simulate failure and stop processing other clients and server’s requests.

|  |  |  |
| --- | --- | --- |
| Message Type | Sender/Receiver | Message Body |
| CLIENT\_PUT\_OBJECT | Client/Server | Content Object to be stored |
| CLIENT\_GET\_OBJECT | Client/Server | Key of the Object to be retrieved. |
| SERVER\_PUT\_OBJECT | Server/Server | Content Object to be stored and its 3 tuple timestamp from the initiating server. |
| SERVER\_CONTROL\_FAIL | Control/Server | Control utility ask server to Simulate Fail. |
| SERVER\_CONTROL\_ISOLATE | Control/Server | Control utility ask server to cut itself from the initiating node. |
| SERVER\_TO\_CLIENT\_PUT\_OK | Server/Client | Client’s put request finished successfully, as there is at least one other server agreed to write. |
| SERVER\_TO\_CLIENT\_PUT\_FAIL | Server/Client | Clients put request failed, as there is no other server available to backup this writing request. |
| SERVER\_TO\_CLIENT\_READ\_OK | Server/Client | Content Object as client requested. |
| SERVER\_TO\_SERVER\_PUT\_OK | Server/Server | The cohort server fulfilled the coordinating server’s request successfully. |
| SERVER\_TO\_SERVER\_PUT\_END | Server/Server | The Coordinating server signals this writing synchronization is done, and the communication channel can be destroyed. |
| SERVER\_CUTOFF\_OK | Server/Control | Server successfully cut itself off form the other node. |
| SERVER\_UNAVAILABLE | Server/Server, Server/Client | Server is simulating failure mode. |

The communication happens in following patterns:

### Client Read:

1. Client has an object key and wishes to read its value. The clients compose a message object of type CLIENT\_GET\_OBJECT and send it to the object’s primary storage server, or secondary storage server, or tertiary storage server.
2. Server upon receiving the request, parses the object key, looks it up in its storage, and sends the value back in a message of SERVER\_TO\_CLIENT\_READ\_OK type. If the value does not yet exist on the server, the object itself embedded in the message will be NULL value.

### Client Write:

1. Client has an object ready and wishes to commit it into the storage. The client composes a message of type CLIENT\_PUT\_OBJECT; with the object itself embedded in the message, and send the request to the object’s primary, secondary, or tertiary storage server.
2. Upon receiving the clients put request, the server prepares at SERVE\_PUT\_OBJECT message and stamps the object with its knowledge about the object’s primary, secondary, tertiary server’s logic time stamp. The server then sends the SERVER\_PUT\_OBJECT message to the other reachable (not cut off form this server) backup servers of this object, thus initiating the write coordination process.
3. Upon receiving the SERVER\_PUT\_OBJECT request from another server, the receiving server will compare the object in its storage against the object in the message. If the object in message has a later time stamp, then this object will be stored in this server’s storage. After finishing the message processing, the receiving server sends the initiating server SEVER\_TO\_SERVER\_PUT\_OK (even if the physical write did not happen, as in future implementations, the write will always happen as we intend to keep multiple versions of this object in storage). If the receiving server is simulating crash failure, the receiving server will send the initiating server SERVER\_UNAVAILABLE.
4. Upon receiving a reply from the other server, the initiating server will send the server SERVER\_TO\_SERVER\_PUT\_END message to indicate the socket connection can be severed, as no further communication is required for this round of write coordination, and close the socket. The receiving server will close its socket connection accordingly.
5. If the initiating server received at least one SERVER\_TO\_SERVER\_PUT\_OK message, it will try to write the same object in its storage. The object will be stored if it has a later timestamp than the value currently stored. The initiating server then sends SERVER\_TO\_CLIENT\_PUT\_OK to the client.
6. If the initiating server can not get a consent from any of the other reachable backup servers, it will not write the object in its storage either, and it will send the SERVER\_TO\_CLIENT\_PUT\_FAIL message to the client.

### Server Cutoff:

1. Based on user input, the control utility will prepare cut off messages to be sent to servers. If a server A and server B was disconnected, the control utility will send server A cutoff message for server B, and send server B cut off message for server A.
2. Once a server receives a cut off message, it will add the “from server” to its list of cut off server nodes, and will exclude these servers from write coordination in future.

### Sever Simulation failure:

1. Based on use input, the control utility send the server command to simulate crash failure.
2. Upon receiving the simulating failure request, the server will switch to “crash” mode, and reply SERVER\_UNAVAILBLE to all its future requests.

# Server Side logic:

### Hashing:

Each object in this implementation is uniquely identified by an integer key, and we will use the simple modular function (key % 7) to decide which server to serve as the primary storage, and ((key +1) % 7) as secondary storage, and ((key + 2) % 7) as the tertiary storage.

## Server node:

The server node logic handles the request from clients and other servers. The server also simulates crash failure and isolation based on input from a control utility.

Each object will have a cluster of 3 servers that forms a write quorum, the client can contact any of the server in an object’s quorum to initiate write operation, the three servers will decide if the write can be performed.

For concurrent write request initiated from different servers in the same quorum, the primary servers write order will decide the order on the other servers as well. If the primary server is not available, then the secondary servers order will decide the order for the tertiary server.

Each object is stored with a vector timestamp that contains its primary storage server logic time stamp, secondary storage server timestamp, and tertiary storage server logic time. The vector timestamp is totally ordered.

On each server, only the last write of the object is stored. The later write always overwrite the previous writes.

The read from the server always gets the latest value stored on the server.

While the server is in status “Simulating Failure”, the server will reply every request with “SEVER\_UNAVAILBLE”, and does not processing any messages.

The server also maintains a list of cut off nodes. In isolation mode, the server will not be able to contact any servers in its cut off list to form a write quorum.

# Client Side logic:

### Automated Client:

The automated client is mainly used for testing and validating server logic. The automated client reads from the configuration file to find out the servers’ listening port, generates random read and write request and sends them randomly to the primary, secondary or tertiary servers.

### Interactive Client:

The interactive client is designed to read commands from the console, parses the commands, and sends the proper messages to server, interprets the responses and displays the results to user.

The command has 4 components:

|  |  |
| --- | --- |
| Filed Name | Description |
| Server Choice | Takes value from (A,B,C) to indicate if the ensuing object’s primary storage server, secondary storage server, or tertiary storage server should be used. |
| Command | GET/PUT, indicates if this is a read or write command. |
| Object Id, | Integer value to indicate the key of the object being operated on. |
| Value | Optional integer value used for PUT command, as value to be stored. |

### Read Operation:

For read operation, the client will compose a message of type “CLIENT\_GET\_OBJECT”, and put the object id of the target object in the included “ContentObject” inside the message, and send the request to the server identified by the user.

If the server is alive, then the return message will contain the latest value stored on that server for this object.

If the server crashed, then the return message will be of type “SERVER\_UNAVAILABLE”.

### Write Operation:

For write operation, the client will compose a message of type “CLIENT\_PUT\_OBJECT”, and put the object to be stored in the message and send to the designated server. If the server is able to get consent from at least another server, then the client will receive the SERVER\_TO\_CLIENT\_PUT\_OK. If the server cannot contact another server to reach a write consensus, the client will receive the SERVER\_TO\_CLIENT\_PUT\_FAIL message.

# Control Utility Logic:

The control utility reads from the console input and sends server control messages. The command line format is following:

|  |  |
| --- | --- |
| Input | Description |
| ISOLATE/FAIL | The first field is used to control server isolation, or simulation of failure. |
| Server ids separated by comma | The servers to be put in an isolation group, or the server that is to simulate fail. |

The ISOLATE command will put the server ids in the parameter into one cluster, and the other servers into another. Only servers in the same cluster will be able to communicate with each other. For example, if user types:

ISOLATE,0,1,2

It will partition the 7 servers into two groups (0,1,2) and (3,4,5,6).

If user input

FAIL,1

It will put the server node 1 into simulation failure mode.

# Assumptions and Design tradeoffs

### Eventual Consistency model

In implementing this project, we assume that for concurrent writes coordinated through the same server, the last write wins. For concurrent writes coordinated from the different servers, the primary storage server decision is the final result. The value that the primary storage server keeps should be the same value kept on other servers eventually as well.

The implementation follows the eventual consistency model, as when write happens, one server could have this value updated before the write operation is committed on the other two servers. However, when all write operations finishes, the values of the same object on all three servers should be identical.

### Minimum communication overhead:

The design aims to minimize the communication overhead but

# Appendix:

## Selected Data Structures:

**VectorTimestamp** {

Integer[] timeVector;

@Override

public int compareTo(VectorTimestamp ts);

}

**ContentObject**{

Integer objId;

String strValue;

Integer intValue;

VectorTimestamp timestamp;

}

**MessageObject**{

MessageType messageType;

ContentObject contentObject;

VectorTimestamp timestamp;

Integer fromServerId;

}

# References:

[1] Dynamo: Amazon’s highly available key value store <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CCgQFjAA&url=http%3A%2F%2Fwww.allthingsdistributed.com%2Ffiles%2Famazon-dynamo-sosp2007.pdf&ei=-cFVU_60O4n68QHN2oGADA&usg=AFQjCNHhJccl0_0I9x7tkWizMx6NjcuUkQ&bvm=bv.65177938,d.b2U>