ECE454 Distributed Systems
Assignment 1 Report

Fasih Awan 20424848

Victor Lai 20426719

June 5, 2015

1 Implementation Details

1.1 Servers and Protocols

We chose to use different types of servers depending on some presumed possible loads from the front and back end. We used the normal, non-async interfaces because we thought understanding the load balancing strategies on the servers would be easier with synchronous interfaces.

1.1.1 Front End Servers

FE nodes launch a **TThreadPoolServer** for the A1Password service. We use 5 worker threads.

The TThreadPoolServer creates a new worker thread to handle each client request. If we set a sufficiently high worker/thread count for the server, we can handle many requests concurrently because each worker thread can also handle network I/O. This means we can process as many requests as possible concurrently and efficiently accept a large number of client requests without being bottle-necked and blocked by new client connections.

FE nodes use a **TThreadedSelectorServer** for the A1Management service. We use 2 selector threads and 2 worker threads.

The TThreadedSelectorServer has two separate pools of network and worker threads. This means that we can have good control over the network and worker load. The main management workload for FE nodes is executing the gossip protocol, where every FE node exchanges server information with another random FE node every 100ms. Having multiple worker and network threads means we will rarely be blocked on network I/O or processing work, ensuring the exchanges are quick. We do not expect the management service to service many client connections at a time, so we do not need TThreadPoolServer's capability to manage network I/O on every worker thread.

1.1.2 Back End Servers

BE nodes use a **THsHaServer** for the A1Password service. We scale the number of worker threads to the number of cores on the node.

THsHaServer works well with handling many client requests concurrently while being busy performing work. We expect the password service to be blocked on the worker threads often, so we should scale the number of worker threads to the number of cores available. We also expect the network processing to be relatively quick relative to the work processing, so we don't anticipate being bottle-necked by the network processing. This fits the client-server architecture that THsHaServers work well for.

BE nodes use a **TNonBlockingServer** for the A1Management service.

TNonBlockingServer only uses one thread, so we minimize the resource usage, and because we expect requests to the BE management services to execute very quickly, we do not anticipate any problems with blocking incoming client connections.

1.1.3 Transfer Protocol

We chose to use the TCompactProtocol over TBinaryProtocol because sources online appeared to say the TCompactProtocol was more efficient and quicker than TBinaryProtocol.

1.2 Gossip Protocol

We gossip between FE nodes every 100ms. The selection process for gossipping is random. The data exchanged with gossipping is simply the list of online nodes that the server has. We merge the online list received from the other gossipping node with our own list. This gives us all the online nodes to target.

1.3 Failure Detection

How do you detect and deal with crash failures?

To detect failures, we use a loop in our handlers that listen for any exceptions that might get thrown by our request to a BE server. This allows us to handle exceptions thrown and take appropriate action. This is all surrounded by a retry loop that keeps looping until the result object is filled in or a set timeout of $30ms$ has been reached. If the result object is still not filled with the server response after the timeout, then we throw an exception up to the method that called the front end node. This exception is caught, and this signals that a server that we tried to connect to is down. We then schedule tasks to tell all the other servers that are online that a server has gone down, so that they can remove that server from their online list. Once we have notified other servers of this failure, we call the scheduler to give us another online server and we try again with the request.

1.4 Load Balancing

How do you achieve load balancing?

Load balancing was done using a weighted random approach. In order to associate a weight to each server, we use the number of cores passed to each server. The weighted random algorithm would sum up all the weights of the servers, and then generate a random number from [0, sumOfAllWeights). We then iterate through all the online servers, subtracting their weight from the random number until the random number is less than the i^{th} online server's weight. That server is then selected.

Using weighted random means we will generally prefer servers that can handle more load efficiently. With the random strategy we select good distribution of servers, though we may run into cases where one server will not be chosen to run for too long.

2 Constraints

2.1 Joining and Updating the Cluster

How do you ensure that a BE node that joins the cluster can receive requests from the FE layer within 1 second of startup?

As soon as a new BE node enters the cluster, it contacts all the seed nodes it has in its parameters. It contacts them letting them know that it exists with its host name, password and management ports, number of cores, and server type (BE, FE). When it contacts the seed nodes, the seed node adds that server information to its list of servers, which is gossiped to other FE servers every 100ms as part of the gossip protocol put in place for FE servers. This happens as soon as a new node is registered, and at this point, at least one FE server knows that it exists because the registration of the new BE node waits for at least one seed node to return a response, so the scheduler can now contact the new BE server with requests. This means that unless all the seed nodes are down or completely throttled, the new BE server should be available for requests from the FE layer within 1 second, after 10 rounds of gossipping.