Assignment 1

Due date: June 3rd at 11:59pm Waterloo time

ECE 454 / 750: Distributed Computing

Instructor: Dr. Wojciech Golab wgolab@uwaterloo.ca

Assignment TA: Hua Fan h27fan@uwaterloo.ca

Learning objectives

To gain hands-on experience with Apache Thrift:

- defining and implementing RPC interfaces
- selecting Thrift protocols and server implementations

To develop a basic understanding of scalability:

- request level parallelism (RLP)
- load balancing

To develop a basic understanding of fault tolerance:

- detecting failures
- maintaining service availability despite server crashes

A few house rules

Collaboration:

- groups of 1 or 2 for ECE 454 students
- groups of 1 for ECE 750 students

Managing source code:

- do keep a backup copy of your code on a different storage device
- do <u>not</u> post your code in a public repository (e.g., GitHub free tier)

Software environment:

- build your solution on eceubuntu using the software provided: JDK (cross-compile to Java 1.6), Apache Thrift 0.9.1, and ant
- at your own risk you may use Scala 2.11.X, also provided
- test your solution on ecelinux using JRE 1.6

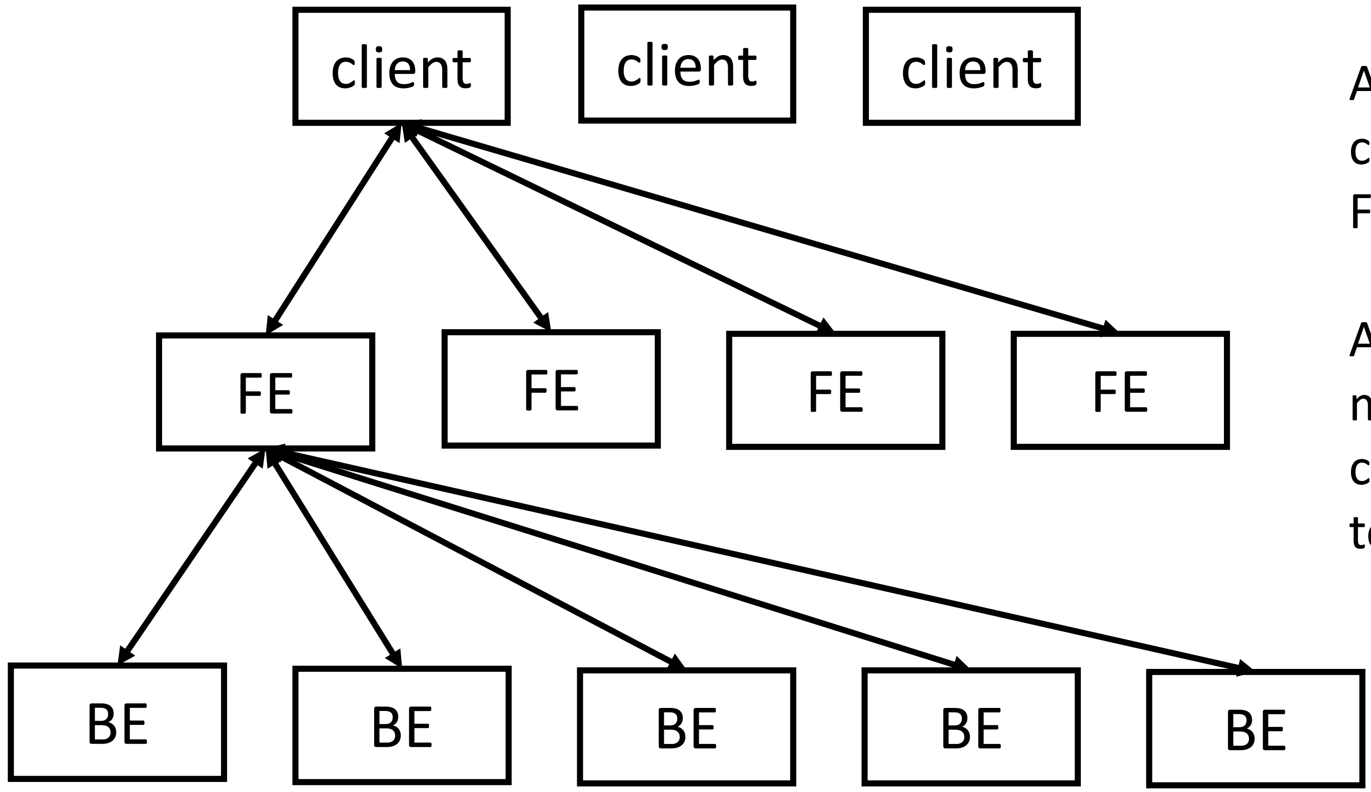
Overview

In this assignment you will build a scalable distributed system for computing the bcrypt key derivation function, which is a popular technique for securing passwords in web applications.

The system will comprise a front end (FE) layer and a back end (BE) layer. The FE layer will accept connections from clients and forward requests to the BE layer in a manner that balances load. Both layers will be distributed horizontally for scalability.

Furthermore, the system will support elastic scalability and rudimentary fault tolerance.

System architecture



Any client may connect to any FE node.

Any FE node may forward the client's request to any BE node.

Functional requirements

The system will support two fundamental computations on passwords:

- **Hash password:** hash a given password.
- **Check password:** check that a given password matches a previously computed hash.

In addition, each process will implement a management interface for querying **performance counters**, such as the total number of requests received.

To facilitate grading you will also implement a third interface that enumerates the members of your group.

Non-functional requirements

Scalability: The system must provide scalable performance with respect to both the number of hosts and the number of cores allocated to each process. To maximize parallelism each layer must be distributed horizontally and each component must be multi-threaded.

Elasticity: The system must allow addition of FE and BE nodes "on the fly". To simplify coordination, every process will be configured with the coordinates of a subset of designated FE nodes called "seeds".

Fault tolerance: For high availability the FE nodes must mask failures of BE nodes from clients. A request forwarded to a BE node that crashes before returning a result must be rerouted.

Interfaces: password hashing

Hash password: Computes bcrypt over a given password using a given number of rounds. The cryptographic salt is generated internally by the service. The return value is a single string encoding the number of rounds, salt, and hash.

```
exception ServiceUnavailableException { 1:string msg }
```

```
string hashPassword (1:string password, 2:i16 logRounds)
throws (1: ServiceUnavailableException e)
```

Check password: Checks the given plain text password against the given password hash obtained from hashPassword. Returns true if the passwords match, and false otherwise.

```
bool checkPassword (1:string password, 2:string hash)
```

Interfaces: performance counters

Get performance counters: returns a structure with various performance counters.

```
struct PerfCounters {
        // number of seconds since service startup
        1: i32 numSecondsUp,
        // total number of requests received by service handler
        2: i32 numRequestsReceived,
        // total number of requests completed by service handler
        3: i32 numRequestsCompleted
}
PerfCounters getPerfCounters ()
```

Note: 2/3 refer to hashPassword and checkPassword requests only.

Interfaces: group membership

Query group members: returns a list of strings representing the Nexus IDs of the group members.

```
List<string> getGroupMembers ()
```

Note: The Nexus IDs (e.g., “h5simpson”) are case-sensitive but their order in the returned list does not matter.

Interfaces: other

You may need to implement additional interfaces, which you will design yourself, to support load balancing and elastic scale-out.

Example 1: On startup, an FE or BE must contact one of the FE seed nodes to join the cluster.

Example 2: If a BE node joins the cluster then every FE node must eventually learn the coordinates of the new BE node.

Packaging

Java package name / Thrift namespace: ece454750s15a1

Thrift service names: Create one service called A1Password for the password hashing interface, and bundle all the other interfaces into a service called A1Management.

Class names: The main classes for the BE process and FE process should be BEServer and FEServer, respectively.

BCrypt implementation: Use the open-source Java library http://www.mindrot.org/projects/jBCrypt/.

Classpath: We will execute your FE and BE processes with libthrift.jar (v0.9.1) and jbcrypt.jar (v0.4) in the classpath.

Configuration

FE and BE processes will receive the following command line arguments:

- -host: name of host on which this process will run
- -pport: port number for password service
- -mport: port number for management service
- -ncores: number of cores available to the process
- -seeds: comma-separated list of host:port pairs corresponding to FE seed nodes

As an example, an FE node might be launched as follows:

```
java -host ecelinux1 -pport 8123 -mport 9123 -ncores 2
-seeds ecelinux1:10123,ecelinux2:10123,ecelinux3:10123
ece454750s15a1.FEServer
```

Note: To avoid conflicts please use port numbers that end with the last four digits of your Waterloo student number.

Testing

The following is a non-exhaustive list of desirable behaviors for your system:

- The FE layer should distribute forwarded requests across the entire BE layer even if all the clients connect to the same FE node.
- Load balancing should take into account the processing capabilities of different BE nodes, which may receive different numbers of cores. (We will assign cores to processes using the **taskset** command.)
- A BE node that joins the cluster should be eligible to receive forwarded requests from the FE layer shortly after (e.g., within 1s of) contacting one of the seed FE nodes.
- If a BE node crashes or is shut down using the kill command then the FE nodes should detect the failure and re-balance load accordingly.

Test your implementation thoroughly!

Assessment

50% for functional requirements:

- password hashes will be validated against a reference implementation implemented using the same Java BCrypt library
- performance counters returned by FE and BE nodes will be validated against statistics collected by the test driver

50% for non-functional requirements:

- the response of the system to the addition and removal of FE and BE nodes, including simulated crashes, will be assessed
- peak throughput and average latency will be measured to assess scalability and load balancing
- the solution will be tested in a heterogeneous hardware environment (e.g., mixture of old and new servers, different core counts)