Raft consensus algorithm

Author: Joan Manuel Marquès

Distributed Systems course

Universitat Oberta de Catalunya

Autumn 2013

Índex de continguts

1. Assignment Outline

The aim of this practical assignment is to implement and evaluate a consensus algorithm for managing a replicated log.

The project consist on:

- Implementing Raft consensus algorithm [1] into an application that stores cooking recipes in a set of replicated servers.
- Evaluate how Raft behaves under different conditions.

1.1 Groups

- **Phase 1**: **individually**.
- **Phases 2 to 4:** you are strongly advised to do the **phases 2 to 4 in groups of 2 students**, even though it is also possible to do it individually.

Use dslab web (http://sd.uoc.edu:8080/dslab) to automatically assess the practical assignments. After registering, you will be able to create your group.

1.2 Evaluation

Phase 1: up to a grade of C-

Phase 2: up to a grade of C+

Phase 3: up to a grade of B

Phase 4: up to a grade of A

1.3 To Deliver

Two deliverable (more details in section 5):

1. phase 1
2. phases 2 to 4

Deadlines are in the course schedule.

2. Overview

2.1 Application

The practical assignment implements the core mechanisms of a replicated application that stores cooking recipes. The application is formed by N servers, all of them computing the same sequence of command (the same state and sequence of outputs).

A recipes has three fields:

- Title
- Recipe
- Author

Two operations on recipes:

- Add recipe
- Remove recipe

Add and remove operations can be applied to any server.

Servers will converge to a common state (all servers having all recipes) by means of a consensus algorithm: Raft consensus algorithm. Raft manages a replicated log and guarantees that all servers will apply all operations in the same order.

2.2 Raft consensus algorithm

Raft is a consensus algorithm for managing a replicated log. [1] explains the details of the algorithm. (**https://ramcloud.stanford.edu/wiki/download/attachments/11370504/raft.pdf**)

A **video and a presentation** (together with other relevant information) is available at **http://raftuserstudy.s3-website-us-west-1.amazonaws.com/study**/

2.5 Your tasks

Phase 1: Basics of Raft consensus protocol.

Phase 2: Implementation of the leader election.

Phase 3: Implementation of the log replication.

Phase 4: Evaluation of Raft consensus algorithm and client interaction.

3. Phases of the practical assignment

3.1 Phase 1: Basics of Raft consensus algorithm

Phase 1.1 Questions

Answer briefly (two or three sentences per question) the following questions about the RAFT algorithm.

1. The RAFT algorithm is an algorithm to keep a log replicated and a state machine in multiple servers. What does this log contain and how it is related to the state machine.
2. Which module job is to keep logs replicated and consistent? When commands are processed by state machines?
3. At any given time each RAFT server is in one of three states. Which are these three states? In normal operation, how many servers there are at each state? What is the role of each server for each state?
4. What is a "*term*"*?* How are terms numbered? When does a new term start? What is the purpose of terms? What do they detect?
5. a) At any given time, do all servers have the same term? b) Is it possible to any server increment its term number? If it is possible, when does a server increment its own term number? c) Is it possible to any server decrement its term number? If it is possible, when does a server decrement its own term number? d) When does a server update term and to which value?
6. How many RPC messages are there in the RAFT algorithm? Which are they? In which state are they invoked and with which purpose (maximum 2 lines each). In which state can RPC be received and why?
7. Which is the initial state of a server? For how long does it remain in this state? Which mechanism is used to trigger a leader election? When does a Follower become a Candidate (start an election to choose a new candidate)? Can a Follower become Leader directly?

8. When can a Candidate become a Leader? When does it win an election? How many votes can a server emit (answer) for a given term, in which basis?
9. Is it possible to have more than one server leader at any given time? If it is possible, can server leaders have the same term? If it is not possible, how RAFT ensures no duplicated server leaders?
10. Each server has its own log. What information is stored at each log entry? At any given time, are all logs the same? How does a RAFT leader handle inconsistencies between logs?
11. When sending an AppendEntries RPC, the leader includes the index and term that immediately precedes the new entries. When does a Follower append new entries to its log and when does a Follower refuse a new entries?
12. Leader has a nextIndex for each Follower, which is the index of the next log entry the leader will send to that follower. Which value it is initialized with? When does the leader decrement the nextIndex of a Follower? When does the leader increment a nextIndex?
13. What is a committed entry log, how is it related to the state machine? In the simple case (the log entry has the same term) when a log entry is committed? Can an entry with a lower term be stored before an entry from its current term?
14. Clients send requests to servers requesting commands. Which server sends all requests to clients? What happens if a client contacts to another server? If a command appears to be lost from the client perspective, it can retry. Which solution can a client use to avoid executing the same command multiple times?

Phase 1.2 Exercises

1. Each figure below shows a possible log configuration for a Raft server (the contents of log entries are not shown; just their indexes and terms). Considering each log in isolation, could that log configuration occur in a proper implementation of Raft? If the answer is "no," explain why not.

a)

index:	1	2	3	4	5	6
term:	1		1	2	2	2

b)

index:	1	2	3	4	5	6
term:	1	2	3	2	3	3

c)

index:	1	2	3	4	5	6
term:	1	4	4	7	7	8

d)

index:	1	2	3	4	5	6
term:	1	1	1	1	1	2

2. The figure below shows the state of the logs in a cluster of 5 servers (the contents of the entries are not shown). Which log entries may safely be applied to state machines? Explain

your answer.
Row 1 is the log index, rest of rows are server logs. Server in row 2 is leader in term 4.

1	2	3	4	5
1	1	3	3	4
1	1	3	3	
1	1	2		
1				
1	1	3		

3. The figure below shows the state of the logs in a cluster of 5 servers (the contents of the entries are not shown). Row 1 is the log index, rest of rows are server logs indicating the term. Which servers can be elected leader if in term 4 server 1 fails? Explain your answer.

1	2	3	4	5	6	7
1	1	2	2	2	2	4
1	1	1	3			
1	1	2	2	2		
1						
1	1	2	2	2	2	

Phase 1.3 Your tasks

- Do the exercise from section *Phase 1.1* and *Phase 1.2.*

3.2 Phase 2: Implementation of the leader election

Implement the leader election part of the Raft algorithm in the `RaftConsensus` class (Package: `recipesService.raft`).

Package `recipesService.raftDataStructures` contains the implementation of required data structures (`PersistentState` and `LogEntry`). Also contains the implementation of another useful class: `Index`.

Use the following RPCs methods (both from `RMIsd` class in package `communication.rmi`) to invoke RPCs in other servers:

- `requestVote`
- `appendEntries`

Phase 2.1 Your Tasks

Implement the election of a leader.

To test if your solution works properly use the provided test environment. Section 4 contains more details about it.

3.3 Phase 3: Implementation of the log replication

Implement the log replication part of the Raft algorithm in the `RaftConsensus` class.

Phase 3.1 Your Tasks

Extend raft consensus algorithm from previous section implementing the log replication.

To test if your solution works properly use the provided test environment. Section 4 contains more details about it.

NOTE: be **careful with concurrent access to data structures**. Two actions issued by different threads may interleave. Whether necessary, **Use** some **synchronization mechanism to avoid interferences**. Nevertheless, remember that `synchronized` statements are costly. In the final report **detail and justify** where and why do you **used `synchronized`** statements.

3.4 Phase 4: Evaluation of Raft consensus algorithm and client interaction

Phase 4.1 implement log compaction and client interaction

Extend the implementation of raft consensus adding the client interaction described in section 8 of the reference paper [1]. Design and implement a solution and test it in the running environment.

Phase 4.2 Evaluation of Raft consensus algorithm

Run the practical assignment:

- in different environments:
  - local: all instances of the service in a single computer
  - distributed (realistic environment): instances of the service running in different computers distributed in Internet (and, therefore, connected by a real network).
- under different conditions:
  - scale: number of servers and number of clients
  - dynamicity: different degrees of connection and disconnection
  - level of activity generation (add and remove operations)

Evaluate the impact of parameters on the behavior of the raft consensus algorithm. Also compare local and distributed settings. You should detail the experiments done and the obtained conclusions. Current implementation includes a basic modeling of parameters. You are encouraged to improve this modeling to get a more realistic one. Changes must be justified.

Phase 4.3 Your tasks

Implement the *log compaction and client interaction* sections and test them in the testing environment.

Do a rapport describing how parameters and running environment influence on raft consensus algorithm performance.

4. Implementation and testing

4.1 Environment

Requires Java 7.

We recommend you to use eclipse as IDE. We will provide you an Eclipse project that contains the implementation of the cooking recipes application except some parts related to raft consensus algorithm.

All scripts for running local tests are prepared for Ubuntu-linux but other OS can be used. In that case, you will be responsible of adapting the scripts to your OS.

For its **formal evaluation**, **each phase** of the practical assignment has to be **uploaded in dslab web** (http://sd.uoc.edu:8080/dslab), **that will execute it using the distributed evaluation environment prepared for this purpose**. This environment will run your implementation of the practical assignment in a realistic environment. Some predefined tests will be executed. Next you will get feedback telling you whether the executed phase is correct or not.

Before sending your practical assignment to this evaluation environment, you can use the script *start.sh* included in `scripts` directory to test your practical assignment locally. We strongly recommend you to thoroughly test your practical assignment locally before sending it to the evaluation environment.

4.2 Phases 2

To test your implementation locally in your computer use the shell script `start.sh` (`scripts` folder).

This script has many parameters. Some useful examples:

(arguments are explained only on its first appearance)

`$ ./start.sh 2 -phase 2`

Runs 2 `Servers` and a `TestServer` locally.

`-phase`: phase to evaluate. In this case, phase 2

Dynamism (connections and disconnections) is automatically generated. (We **recommend** you to **start** testing your implementation with no disconnections or failures. To do it, set `probDisconnect` parameter from `config.properties` file (in `scripts` folder) to 0)

`$ ./start.sh 5 -phase 2 &>../results/F`

Runs 5 `Servers` and a `TestServer` locally.

`-phase`: phase to evaluate. In this case, phase 2

`&>../results/F`: standard and error out is stored in file `F` (in `results` folder). To see the content of file `F` while the application is running execute `tail` command wit `-f` option in another terminal (from `results` folder):

```
$ tail -f F
```

We recommend you to **start** testing your implementation locally with two servers and no disconnections or failures (set `probDisconnect` parameter from `config.properties` file to 0 in `scripts` folder).

Then execute your implementation locally with more servers and disconnections and failures. Implementation should work with 5 servers and default values of `config.properties` file.

Finally, once your application runs in local, upload it in dslab web for its formal evaluation. In dslab, your implementation will run in a real distributed environment interacting with instances (also distributed) of an implementation done by the professors of this course.

(REMEMBER: use dslab web for its formal evaluation)

4.2.1 Evaluation framework

We have created a system to transparently execute and assess your practical assignments in a real distributed environment. These runs include instances of your implementation and of professor's implementation.

First of all, you must create a group. Then, you are able to upload your solution of the practical assignment, run it and check the result of all your executions.

(dslab web site: http://sd.uoc.edu:8080/dslab)

Building projects

First you need to create a new project uploading the necessary Java classes from your local machine. For example, you can create a project for each phase.

Secondly, you are able to built the project in order to use it for the next step.

Launching experiments

A built project can have as many experiments associated as you need. Thus, selecting one built project, you can configure and launch an experiment into the remote platform transparently.

Checking results

Once finished the experiment execution, you can review your results in your experiment information.

4.3 Phases 3

To test phase 3 locally:

```
$ ./start.sh 5 -phase 3 &>../results/F
```

Runs 5 `Servers` and a `TestServer` locally.

```
$ ./start.sh 5 -phase 3 --logResults &>../results/F
```

Runs 5 `Servers` and a `TestServer` locally.

Results will be stored in a file named as the *groupId* from `config.properties` file (on current path).

`--logResults`: log results in the following two files:

- <*groupId*[2]>: log of all executions and the result for each of them.
- <*groupId*[3]>`.data`:
  - in the case that all solutions are equal, contains the final state of data structures.
  - in the case that NOT all solutions are equal, contains the first two found solutions that were different.

```
$ ./start.sh 5 -phase 3 --logResults -path ../results
  &>../results/F
```

Runs 5 `Servers` and a `TestServer` locally.

---

2 Name of the file will be the value of `groupId` property in `config.properties` file (`scripts` folder).

3 Value of `groupId` property in `config.properties` file (`scripts` folder) will be the first part of the file name. i.e. file name: <value of groupId property>`.data`.

`-path <path>`: result files are created in `../results` folder. If no `-path` is specified (as in previous example) files will be stored in current folder.

(REMEMBER: use dslab web for its formal evaluation)

4.4 Phases 4

Use the script `start.sh` explained in previous phases.

In *phase 4.1* (extend application adding *client interaction*), run `start.sh` script to test your implementation locally.

(REMEMBER: use dslab web for its formal evaluation)

In *phase 4.2* (Evaluation of Raft consensus algorithm), modify parameters of `config.properties` file (`scripts` folder) to evaluate Raft consensus algorithm under different conditions and environments.

- Use *runN.sh* script to run N times *start.sh* script. Example:

  ```
  $ ./runN.sh 10 5 -phase 3 --logResults -path ../results
  ```

  runs 10 times *start.sh* script with the following parameters:

  ```
  $ ./start.sh 5 -phase 3 --logResults -path ../results
  ```

`Client` class (package `recipesService.test.client`) and `SimulationData` (package `recipesService.activitySimulation`) will help you to better understand the modeling of activity and dynamicity (respectively). You are free to improve its behavior for *phase 4.2* (provided you explain the changes and reasons for the changes). In any case, do it only for *phase 4.2*. Other phases should use predefined behavior.

5. Things to deliver

Two separated deliverables. One for phase 1 and another for phases 2 to 4.

In both deliverables, you should send a zip file. **Name** the **file** according to the following convention:

Year-groupId-FamilyName1_Name1-FamilyName2_Name2.zip

This file should **include**:

- For Phase 1 and Phase 4: required docs (name files as *phase1.pdf* and *phase4.pdf* respectively)
- A (short) report detailing and explaining all decisions taken. A proposal of report template is included in the practical assignment distribution (*/doc* folder).

  In addition, you should detail the portions of your source code you consider are most important, an explanation about how it works and the tests you used to validate your assignment.

  Finally, scripts and a detailed example of how to run your practical assignment.
- Source code: your eclipse project and the portion of source code you have implemented.

The zip file should have a single directory named like the zip file with the following structure (use same structure and names):

Subdirectory	Content
`/doc`	Report in pdf Other docs (phase 1 and phase 4)
`/src`	Source code. Eclipse project with your implementation.
`/bin`	All required files to execute your practical assignment. Explained carefully in the report how to execute it.
`/sol`	Your solution. Only include the code that you have done, i.e. the classes you implemented or modified. i.e. the files from `/src` that you have implemented, without subdirectories. Only the .java.

Annex A. Source code and documentation

`Papers` folder contains referenced paper that explain Raft consensus algorithm.

`Raft` folder contains all packages and classes required to the practical assignment.

Important: **do not implement new classes**. **Modify only the classes indicated in each phase**.

(except if you modify the basic modeling of parameters in phase 4)

Classes that you should modify:

1. package `recipesService.raft`:
   - `RaftConsensus`: implements raft functionality. You can create inner classes to handle the sending of RPCs (or create them in a separate class)

Classes that you should use but NOT modify:

2. package `recipesService.raftDataStructures`:
   - `PersistentState`: persistent state of all servers
   - `LogEntry`: a log entry
   - `Index`: a generic index class used by `nextIndex` and `matchIndex.`
3. package `recipesService.data`:
   - `AddOperation`: an add operation (operations are logged in the `Log` and exchanged with other partners).
   - `RemoveOperation`: a remove operation (operations are logged in the `Log` and exchanged with other partners).
   - `Recipe`: a recipe.
   - `Recipes`: class that contains all recipes.
4. package `communication.rmi`:
   - `RMIsd`: class that encapsulates RPC communication (using java rmi). Simulates failures and disconnections. Use methods `requestVote` and `appendEntries` for RPCs.
5. package `recipesService.test.client`:
   - `Client`: models clients' generation activity. Modify it only in case that in *phase 4.2* you want to improve its behavior. In case you do it, do it only for *phase 4.2*.
6. package `recipesService.activitySimulation`:
   - `SimulationData`: models dynamicity. Modify it only in case that in *phase 4.2* you want to improve its behavior. In case you do it, do it only for *phase 4.2*.

Annex B. `Config.properties` configuration file

Config.properties file contains parameterizable values that would help you test your application under different conditions.

GroupId is the id of the user running the experiment.

If the value of `probDisconnect` is 0 then no disconnections or failure will occur during the execution.

`SimulationStop` sets the duration of the execution. Before sending your implementation for an evaluation test it with at least 300 seconds (5 minutes).

Try your implementation with different number of clients, different value for the period of activity and dynamism simulation, and other parameters.

Annex C. Simulation of dynamicity

Simulation of communication failures and disconnections

`DynamicitySimulation` class (package `recipesService.activitySimulation`) decides when to simulate the disconnection (or network failure) of a host and when to reconnect it.

To simulate communication failures, a disconnection unbinds a server api, what will results in an exception in other partners that wants to call RPCs of the disconnected server.

Annex D. Activity generation

Simulation of clients' activity

`Client` class (package `package recipesService.test.client`) simulates the generation of activity by clients.

References

[1] **Diego Ongaro and John Ousterhout** (2013). In Search of an Understandable Consensus Algorithm.