Distributed Algorithms 347

Coursework – Raft consensus

01 The aim of coursework is to implement and evaluate a simple replicated banking service that uses the Raft consensus algorithm described in the paper: ***In Search of an Understandable Consensus Algorithm (Extended Version)*** by Diego Ongaro and John Ousterhout. The shorter version was published at USENIX, June 2014, pages 305-319. The extended version inlcudes a better summary. Both versions can be downloaded from http://raft.github.io

02 You'll need to develop an Elixir implementation of the algorithm for Raft's *server* component plus any other components that you use. Various components will be provided (see below).

03 You'll need to submit your code and to submit a short report describing your work.

04 You can work and submit either individually or jointly with one classmate. Our recommendation is to do the coursework jointly with one classmate.

05 Parts marked OPTIONAL are UNASSESSED. You're welcome to work on them and report on them **but only work on them if you have time** and wish to.

06 Use Piazza if you have questions about the coursework or general questions. **Do not post your solutions or share them with others.** Email me directly if you have a specific question about your solution.

07 If there are any corrections or clarifications, the most up-to-date version of the coursework will be provided on the course webpage **not on CATE or MATERIALS.**

_____ The deadline for submission is Tuesday 25th February 2020 _____

Part 1 – System Structure

08 A directory with a Makefile and various Elixir modules can be downloaded from:

```
http://www.doc.ic.ac.uk/~nd/347/raft.tgz
```

Familiarise yourself with the supplied files, in particular:

- `Makefile` has commands for compiling and running a system. The Makefile creates 5 Server nodes and 5 Client nodes plus the top-level raft node. You can increase the number of nodes if you wish for evaluation.
- `raft.ex`, `client.ex`, `database.ex`, `monitor.ex`, `state.ex` and `dac.ex` are my implementations of various components - to help get you started. You can extend, delete, change these as you see fit.
- `Monitor` checks that each server database is executing the same sequence of requests. It periodically outputs pairs of lines like:

```
time = 2000  updates done = [{1, 657}, {2, 657}, {3, 657}]
time = 2000 requests seen = [{1, 218}, {2, 220}, {3, 219}]
```

  Here, after 2 seconds, servers 1 to 3 have each performed 657 database updates. Server 1 received 218 client requests, while servers 2 and 3 received 220 and 219 requests respectively. You can change `monitor` (and other components) to produce better checks or more informative diagnostics.
- `Monitor` can also be used as a component for debug functions. But again, you can use a different debugging component/style. Do not remove debugging code that you develop, just make not print out when things are working.

- `DAC` can be extended/changed to set your own configuration parameters, either directly, or via arguments from the Makefile.
- `State` can be extended/changed to define your own server data (held in a map). Server creates and initialises the map and passes it as parameter to subsequent calls. Setters are also defined. You can extend/change/ignore as required.
- `Client` sends requests to servers to move amounts from one account to another. The number and frequency of requests is controlled by configuration parameters set in `DAC`.
- `Database` receives requests to execute move commands from its local `Server` component.
- `Raft` is the top-level component. It creates 1 `Database` component and 1 `Server` component at each Server node, and 1 `Client` component at each Client node. It also creates a local `Monitor` component at its own node for use by any components.

09 `Server` is the main component that **you need to develop**. The `start` function for this is provided. Do not write a lengthy implementation in `server.ex`. Rather use small files for the various functionality and behaviours.

10 For example, the `next` function in my *server* implementation just switches between the various "roles", *follower*, *candidate*, *leader* according to the algorithm. Each role is written in its own Elixir module. I've also put the main message sending and message handling logic for votes and append entry request and replies in their own modules `vote` and `append`. Keep coding simple, do not spent time researching Elixir's functional programming libraries, comprehensions can handle most tasks, otherwise, it's often quicker to write a short recursive function.

11 If you wish, you can (but don't need to!) use Elixir processes for various tasks. For example, I implemented elections by spawning a process that sends vote requests to other servers and collects replies. I didn't implement *follower*, *candidate*, and *leader* as separate processes. It's possible to do so - spawning a new local process as roles are switched and passing the state as a parameter. You would then need to use the *server* process to forward incoming messages to the appropriate current process for the role, i.e. forward client requests to the current leader. Seems like overkill. So only use additional processes if you feel comfortable doing so.

Part 2 – Implementation

12 Before starting on the implementation. Review Montresor's slides on the course webpage and skim through the extended paper marking point of interest. Lots of other materials on the Raft web site that you can use.

13 Page 4 of the extended paper has an informal description of most of the functionality required. Appendix B of Ongaro's PhD has a formal specification. Be warned the notation is "antiquated". The main logic starts at line 186. Note lastEntry on line 209, should not have +1 in the second argument to function min().

14 Montresor's full slides have a pseudo-code description in the style of Cachin for most of the functionality – some aspects are handled differently, e.g. updating commitIndex.

15 Before coding draw one or more diagrams that show the structure and connectivity of the various components, with the types of Elixir messages that pass between them. Update as you revise.

16 Don't try to write all the code in one go. Rather develop the code systematically, step-by-step.

17 **Important**. **Develop an approach to debugging and testing your code from the beginning.** Sometimes you'll want to print a lot of debugging information, other times little. Make it easy to selectively switch debugging on/off. *If you do not instrument your code for debugging the coursework will take longer.*

18 Put your debugging printing functions into a separate module and/or into the Monitor component to keep other code clean. Use assertions.

19 Redirect your output into a file for easier review (e.g. `make > output`).

20 Erlang (Elixir's underlying platform) includes a visual debugger and other debugging tools that can be used. We've never used them, so can't advise.

21 Debug and test with the minimum number of servers/clients and the minimum number of client requests.

22 Start on leadership elections first. Get one *server* to become the *leader* and other servers to become *follower*s. Ignore client requests for now. Test your handling of vote requests and vote replies thoroughly. Experiment with election timeouts. Use a dummy heartbeat to avoid re-elections or some other mechanism to prevent elections. Note: You can use guards on receive patterns to handle cases to write clearer code, e.g.
```
{ :MSG_TYPE, term, ... } when term > curr_term -> stepdown
```

23 The handling of append entries is complex and will take longer to get working.

24 Rig your system to handle one 1 client request and get it committed to all logs. If necessary, get a newly elected *leader* to invent a client request and process it (you can disable the client or not create them in Raft).

25 This involves the *leader* adding the request to its log and sending out append requests to other servers. Once a majority of replies are received, committing to its log and sending to their own local *database* for execution. *Follower*s need to process the append entries message updating their own log and database according to the algorithm and sending a reply reflecting success or failure.

26 Recall that the leader needs to handle replies from each follower individually (with individual follower timeouts) resending to a follower on a timeout or if an induction step is needed.

27 Now change your system to handle multiple client requests (and client replies). Rig the system if necessary, to prevent leadership changes.

28 Change your system to handle leadership changes. Experiment with long delays first. Sleep too long before an append entries is sent. Then node failures – make the leader node exit.

29 Experiment with delays of other messages and with arbitrary server failures.

30 Experiment with more servers and more clients (change the Makefile). high timeouts, low timeouts.

Part 3 – Submission

31 Submit the Mix directory for your implementation (do a **make clean** first), with a pdf of your report (**report.pdf**) included as a *single zip file* on CATE called **raft.zip.**

32 Ensure that your report and your source files with your name(s) and login(s) e.g.,
```
# Mary Jones (mj16) and Peter Smith (ps16)
```
Source files and reports without your name(s) and login(s) will not be marked.

33 Include a README.txt (or README.md) with any additional instructions on how to run your system, particularly for interesting experiments. **It must be possible for us to run your code** using your instructions on CSG Linux machines or on a Mac computer.

34 Include in your report:

35 One or more diagrams that show the structure and connectivity of the various components. Indicate the normal flow messages for a typical client request. Use some notation to indicate replicated and spawned components and some notation for any messages that are broadcast. DON'T SPEND TOO LONG ON THIS!! It's okay to use a scan or a photograph of a clear hand-drawn version of your diagram(s) – you will not lose marks for doing so. Suggested maximum length: 1 page.

36 A brief description of your design/implementation choices. Focus on aspects that someone who is familiar with the algorithm but not your design/implementation will find useful. Suggested maximum length: 1 page.

37 A brief description of your debugging and testing methodology. Again, focus on aspects that someone who is interested in updating/extending your implementation will find useful. Comment on what's hard to debug/test. Suggested maximum length: 0.5 page.

38 Evidence that your system works. Under normal situations, under failures, under load (e.g. no more than 2 pages). Summarise your findings. Don't try to do too many experiments or be comprehensive for the report. We're interested in comments that demonstrate your understanding and critical thinking, including your rationale when conducting experiments, not in being exhaustive. If your code does not work or only partially works you must explain what's not working. Suggested maximum length: 2.5 pages.

39 Put any outputs of your system in subdirectory called **output** with helpful files names, for example, **5_servers_2_ clients, server1_crash, low_election_timeout,** and then refer to files in your report.

40 The maximum report length is 5 A4 pages so summarise your findings. Use graphs if they help. The page limits above are just suggestions.

41 Marking. 10 for the Elixir implementation. 10 for the report.

Congratulations, on completion you'll know quite a lot about Raft so update your CV!