Distributed Algorithms 60009

Coursework – Raft consensus

01 The goal of Distributed Algorithms 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.

02 A shorter version of paper was published at USENIX, June 2014, pages 305-319. The extended version includes a better summary. Both versions can be downloaded from http://raft.github.io/ Even better is Ongaro's PhD thesis at https://web.stanford.edu/~ouster/cgi-bin/papers/OngaroPhD.pdf

03 For the coursework you'll develop and evaluate an Elixir implementation of the core log replication algorithm. Various Elixir modules will be provided to help get you started. Note: the modules may have extraneous variables and functions in places, others are just placeholders. You're welcome to remove them.

04 You'll need to submit your code and a short report (max 5 A4 pages).

05 **You can work on the coursework either individually or jointly with one classmate.** It is recommended that you start on the coursework early. The coursework is not easy, having a classmate to discuss, develop, evaluate and write-up the report should help.

06 Use Ed if you have questions about the coursework or general questions. **Do not post your code or share it 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 my webpage for the module.

The deadline for submissions is Monday 26th February 2024

Part 1 – Supplied Files

08 A directory with a `Makefile` and various Elixir modules to help you get started can be downloaded from: https://www.doc.ic.ac.uk/~nd/dal_lab/cw.tgz

09 Create a new mix project (`mix new raft`) and add the `Makefile` and Elixir files to the generated `mix` directory.

10 **You're welcome to change the code in these files or write your own replacements.** Do highlight any interesting changes that you make in your submission.

11 **Familiarise yourself with the supplied files**. Below are brief summaries: The main files you'll be developing are shaded grey below but do extend/adapt others as desired. I've indicated the length of my files - my code is probably longer than needed since I implement additional functionality.

12 Your aim should be to develop code that you and others can understand. My coding style, formatting, etc may not be up to the modern standards that you've been taught so you're welcome to improve it.

`Makefile`	`Makefile` has commands for compiling and running a system. By default, it creates 5 `Server` nodes and 5 `Client` nodes plus the top-level `Raft` node. The `Makefile` variable `PARAMS` is the name of a set configuration parameters that will be used when running the system. Code in `configuration.ex` will invoke a `params` function corresponding to the `PARAMS` value in the `Makefile`. You should define additional `params` functions for different parameter setups (e.g. for tests and your evaluation experiments). You can adjust the `PARAMS` variable on the command line i.e. `make run PARAMS=params_function.` The `Makefile` also includes 2 variables `DEBUG_OPTIONS` and `DEBUG_LEVEL` that I used to control the information that was output during a run. You're welcome to adapt, extend or ignore these `Makefile` variables. You can also define and set parameters directly in `configuration.ex`
`appendentries.ex`	Add your implementation of Raft's *log replication* logic here. `AppendEntries` is the most complex part of Raft - I recommend that you only start on it once you have implemented an initial *leader election.* My code is ~150 lines.
`client.ex`	`Client` sends requests to move random amounts from one account to another. The goal is for each replicated server's database to execute the same sequence of requests from clients. After each request is sent, `Client` waits for a reply before sending another request. Replies include the process-id of the current Leader (if known) which is used in subsequent requests. Client will select a server round-robin when it doesn't know which server is the Leader. Configuration parameters can be used to control various aspects of the client's behaviour, *such as* the delay after receiving a reply and sending the next request, the maximum number of requests a client should send, and a time-limit defining the maximum time that a client should run. You're welcome to adapt these parameters and behaviour of clients for your evaluation or for debugging but do indicate what you changes were at the top of the file.

`clientrequest.ex`	`ClientRequest` is a module for functions that perform server-side communication with a client. My code: ~30 lines.
`configuration.ex`	`Configuration` defines configuration/setup parameters to control the behaviour of the system. Some parameters are defined and passed to Raft nodes in the `Makefile` (and copied from `argv` in `node_init`). Other configuration parameters are set in `node_info`. `node_init` will also call a params function based on the value of the `PARAMS` variable (see above). You're encouraged to add your own params functions for additional tests and evaluation experiments. Give the functions understandable names.
`database.ex`	`Database` is an implementation of the replicated state machine. `Database` receives requests to execute `MOVE` commands once your Raft implementation decides it is safe to do so. `Database` reports each `MOVE` request to `Monitor` which checks that each server's database executes the same sequence of `MOVE` commands.
`debug.ex`	`Debug` includes various debugging functions that you may find useful. The debugging information generated by these functions is controlled by the `DEBUG_OPTIONS` and `DEBUG_LEVEL` variables in the `Makefile`. The debugging functions are somewhat obscure so do roll-out your own. You may also find it useful to send outputs into separate server files. You're also welcome to use more advanced testing/debugging techniques e.g. unit tests, property-based testing. Do not remove debugging code from your submission, just make it work silently when not needed. Debugging code indicates the care your have taken to produce your solution.
`helper.ex`	`Helper` is like the one used in the lab exercises.
`log.ex`	`Log` is an implementation of a server log. It uses a map indexed from 1. You're welcome to adapt or use a different implementation but do indicate what you did differently.

`monitor.ex`	`Monitor` checks that each server's database is executing the same sequence of client requests. It also 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 2000 milli-seconds, servers 1, 2 and 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're encouraged to extend `Monitor` with further checks and/or collect statistics for your report. Do indicate what your extensions are at the top of the file.

`raft.ex`	`Raft` is the top-level module. It creates (i) one `Database` process and one `Server` process at each server node, and (ii) one `Client` process at each client

	node. `Raft` also creates one `Monitor` process for collecting information from other Elixir processes.
`state.ex`	`State` defines state variables for a server that are implemented as a map, with setter functions for mutable variables. The configuration parameters are also saved in the map (in `config`). My implementation has extraneous variables that I used in experiments. You're welcome to adapt the code but do indicate what you adaptation(s) were at the top of file.
`server.ex`	`Server` is the message receiving process of the server. It receives messages from other servers, clients or timeouts and then calls functions in `Vote` and `AppendEntries` and `ServerLib` to handle the messages. My code is: ~50 lines.
`serverlib.ex`	Library of server-side functions used by `AppendEntries`, `Vote`, `Server`. My code is: ~100 lines.
`timer.ex`	`Timer` handles the creation and cancellation of timers for `ELECTION` timeouts and `APPEND_ENTRIES` timeouts. You're welcome to adapt the code but do indicate what you adaptation(s) were at the top of file.
`vote.ex`	`Vote` handles *leader election.* It is recommended that you implement an initial (simple version) of this and revise it with the more complicated tests for granting a vote as you develop the code fo log replication (`appendentries.ex`). My code is: ~100 lines.

Part 2 – Implementation

14 Before starting on the implementation, review Montresor's slides on the course webpage and skim through the extended paper highlighting points of interest. There are a lot of materials on the Raft website that you can review to understand the algorithm.

15 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 in TLA+. The main logic starts at line 186. Note: `lastEntry` on line 209, should not have +1 in the second argument to function `min`(). Montresor's full slides have a pseudo-code description in the style of Cachin for most of the functionality required – some aspects are handled differently, for example, updating `commitIndex`.

16 **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.* Use assertions to improve your understanding. Adapt `Debug` and `Monitor` and/or add your own debugging techniques. Debug and test with a fewer number of servers/clients and a fewer number of client requests until you are confident that the system is working. You may find it useful to mock your system first, e.g. use dummy messages to check that the flow of messages between processes is correct. Consider redirecting your output into a file for easier review (e.g. `make > output`). Elixir has various debugging tools that can be used - I'm not familiar with them am unable to help.

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

18 If you haven't already being doing so, do use *for comprehensions.* They can iterate over Lists, Maps, MapSets, Strings etc, as well *filter* elements and *reduce* results. They are much easier to read than equivalent functions in the Elixir standard library.

19 Develop the implementation for leadership election 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.

20 Warning. The implementation of `AppendEntries` is more complex and will take longer to get working.

21 Adapt 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 comment out its creation in `Raft`). Log replication involves the *leader* appending client requests to its log and sending out *append entries* messages to other servers. Once the leader knows that a log entry is replicated (strictly stored in stable storage) on a majority of servers it sends it to its own *database* for execution and considers the entry *committed*[1] *Follower*s process *append entries* messages updating their own log and database according to the algorithm. Note that the *leader* needs to handle replies from each follower individually (with individual follower timeouts) resending the *append entries* message on a timeout or if an induction step is needed.

22 After you have the system handling 1 client, change your system to handle multiple clients. If necessary, adapt your code to prevent leadership changes and then change it to handle leadership changes. Make the leader sleep a long time before an *append entries* is sent causing followers to become candidates. Make the leader and/or arbitrary nodes fail based on configuration parameters.

23 Experiment with delays of other messages and with arbitrary server failures. Experiment with more servers and more clients (change the `Makefile`), high timeouts, low timeouts, etc.

[1] The test for committed has an extra constraint. See paper/slides.

Part 3 – Submission

24 Submit the directory for your Elixir implementation as single a *single zip file* called `raft.zip` (do a `make clean` first!!). Include your report (`report.pdf`) inside the directory. Update the `README.txt` with any additional/special instructions on how to run your system, particularly for interesting experiments.

25 **It must be possible for us to compile and run your code** on a CSG Linux workstation or on a Mac computer running (Sonoma or Ventura)

26 Ensure that your report and the Elixir files that you implement include your name(s) and login(s), for example: `# Jane Smith (js19) and Peter jones (pj19)`
If your report or these files do not have your name(s) and logins(s) they may not be marked.

27 Put any evaluation outputs of your system in a subdirectory called `outputs`. Use helpful files names. For example, `01_5_servers_5_clients`, `02_server1_crash`, `03_run_60_secs`. Prefix a 2 digit reference number at the beginning of file names and use the prefix to order related files. Refer to the full file name or just the file prefix in your report.

INCLUDE IN YOUR REPORT

28 ***Architecture***. One or more diagrams that show the structure and connectivity of the various modules and messages. Indicate the normal flow messages for a typical client request. Use some notation to indicate replicated processes and some notation to indicate messages that are broadcast. Label or add comments to improve clarity. 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.

29 ***Design and Implementation***. A description of the rationale for you design/implementation choices. Focus on aspects that someone who is familiar with the algorithm but not **your** design/implementation will find useful. Comment on any interesting debugging code. Don't describe straightforward aspects. Suggested length: 0.5 to 1 page.

30 ***Evaluation***. Describe (in 1-2 lines) the machine you used for your evaluation, e.g. OS, processor, cores, RAM. Then summarise your findings, *for example*, of the behaviour and performance of the system under normal situations, under high load, under failures, when run for 60 seconds. Explain unusual or unexpected results that demonstrate your understanding and critical thinking. Graphs can help summarise findings. Reference the full file name(s) or the short 2-digit file reference number of output files that provide evidence of findings. Suggested length: 3 to 3.5 pages.

31 **The maximum report length is 5 A4 pages (excluding the cover page),** so be concise. *Think of your report as an Appendix to the paper and supplied files/documentation, so don't repeat the contents of the paper.*

32 The coursework will be marked out of 30. 15 marks for the Elixir implementation, 15 marks for the report. We'll be looking for a working implementation, concise but readable code with evidence of debugging, plus a well-written (insightful) report.

33 If you don't manage to complete the implementation, clearly state what's incomplete or not working e.g. '*Our solution is not working for XX*'. If you have some ideas where the problem lies in your code, indicate it in your report, reference any output files that demonstrate the issue.

34 You can add Appendixes (extra pages) to the report if you wish to describe any additional parts or other extensions of your own. There are no extra marks for doing this and the contents of the Appendix may not be read, so consider them personal unassessed pages.

On completion you should know quite a lot about Raft Consensus, so update your CV !