Trabalho 2 - Raft
Marcelo Paulon / 2112515

Trabalho executado com Lua 5.3.6, em um macOS Big Sur (11.3.1) - shell: zsh.

A eleição de líder foi implementada no arquivo "server.lua", usando a arquitetura sugerida em aula (uma fila de tarefas (onde as requisições feitas são armazenadas para serem posteriormente respondidas), e um loop usando a função luarpc.wait para executar as tarefas pendentes).

As configurações de porta, compartilhadas entre os nós, devem estar localizadas em um arquivo "ports.txt" contendo uma lista de portas separadas por vírgulas.

Ao inicializar, cada nó imprime as portas lidas do arquivo, a sua porta (escolhida através de um parâmetro obrigatório, um número de 1..n onde n é o número de nós), e quantos nós são necessários para formar uma maioria para eleição do líder.

Inicialmente, cada servidor encontra-se no estado parado, e para que comece a funcionar depende de receber uma chamada ao método RPC InitializeNode, que só deve ser chamado uma vez.

O arquivo "test.lua" chama o método InitializeNode de um nó, e deve ser chamado para cada nó que deseja-se executar, também passando para ele o índice do nó (como feito para o arquivo "server.lua"; ex.: "lua server.lua 1" para ligar o nó 1, "lua test.lua 1" para chamar seu método InitializeNode e começar sua execução/main loop).

Para imprimir configurações do nó (estado, termo) e habilitar o modo debug / verbose (em que mais estados são impressos, assim como o modo verbose da biblioteca luarpc é ativado), deve-se chamar o método RPC Debug; notas: i) a informação é impressa no próprio nó executando "server.lua", não retornada pelo método RPC; ii) para habilitar totalmente o modo verbose da biblioteca luarpc, deve-se alterar a variável global "debug" no arquivo "server.lua", pois os proxies luarpc são criados logo ao carregar o script.

- O arquivo "debug.lua" recebe o índice do nó e chama o método RPC Debug

Para parar um nó deve-se chamar o método RPC StopNode (arquivo "stop.lua" recebe o índice do nó e chama StopNode). Um nó parado continua recebendo requisições e respondendo com ack, no entanto, as mensagens são ignoradas (são descartadas da fila de processamento).

Para fazer com que o nó volte a executar deve-se chamar o método ResumeNode (arquivo "resume.lua" recebe o índice do nó e chama ResumeNode).

A comunicação entre nós utiliza a biblioteca luarpc com co-rotinas, conforme fornecido na descrição do trabalho, com uma alteração para que as chamadas às implementações utilizem a

função Lua pcall; essa alteração facilitou debugging quando a implementação continha algum erro. O formato escolhido para os dados foi json, visto que a biblioteca não possuía suporte a múltiplos structs.

Testes podem ser executados usando o bash script "test.sh". Esse script recebe como parâmetro obrigatório o número de nós que se deseja simular (mínimo 3 nós, para que haja uma eleição). Exemplo: "./test.sh 5" para testar com 5 nós. O output é consolidado de forma ordenada (por timestamp em segundos, impresso a cada linha junto com o estado e porta do nó) em um arquivo txt na pasta "results", contendo data/hora do teste.
O script inicia todos os servers e tests (que chamam o InitializeNode), aguarda 20 segundos, e depois, para cada nó, aguarda 20 segundos, e retoma a execução do nó. Isto faz com que o tempo de execução do script seja de ~20+20*n segundos onde n é o número de nós.

Cada eleição é iniciada com timeouts aleatórios de 80ms a 300ms, e cada heartbeat do líder é enviado com timeouts aleatórios de 20ms a 40ms; o fator de velocidade utilizado na simulação foi de 60x, ou seja, na prática, cada eleição demora de 4s a 18s e cada heartbeat de 1s a 2s.

Inicialmente, os testes foram feitos com a seed como os.time(), no entanto, muitos nós ficavam com a mesma seed ao serem executados pelo arquivo "test.sh", e, portanto, foi necessária outra solução para que a eleição pudesse ser alcançada. Foi usada então como seed: os.time() - index*10, e isto resolveu o problema para os testes realizados.

Foram realizados testes manuais, parando e voltando a executar nós candidatos, líderes e seguidores. Inclusive, foram testados casos em que nós tentavam se eleger ao mesmo tempo, e recebiam votos de nós diferentes. Inicialmente, nas primeiras versões, tais testes manuais foram extremamente úteis para identificar bugs e avançar na implementação. Por exemplo, em um caso de teste, foi verificado que um nó candidato não se converteu para seguidor, pois outro candidato foi eleito no mesmo termo; imprimir o estado e termo de cada nó foi bastante útil para identificar esse caso e corrigir o bug.

A execução dos testes manuais foi feita inicialmente usando vários terminais abertos, e observando os resultados conforme a simulação avançava (Figura 1). Após automatizar a execução e fazer a consolidação dos logs em um arquivo só (ordenado), tornou-se mais fácil de executar testes com número de nós diferentes, no entanto, mais difícil de customizar os testes (por exemplo, para parar um líder e um seguidor por alguns segundos, e depois voltar com eles em uma ordem específica).

Alguns dos logs avaliados estão disponíveis na pasta results:

- nodes5.txt: simulação com 5 nós;
- nodes10.txt: simulação com 10 nós;
- nodes5-2: outra simulação com 5 nós;
- nodes5-3-leaderStopping.txt: simulação com 5 nós em que o líder é parado (linha 68), e podemos ver outros nós se candidatando e se elegendo em um novo termo (linha 89) alguns segundos depois. Quando o líder volta (linha 105) ele tenta enviar heartbeats do

termo em que estava (termo 1), mas rapidamente recebe um AppendEntries do líder atual e se converte em seguidor (já que o líder atual está com um termo maior, termo=2);

- nodes5-4-timeOnlySeed.txt: simulação com 5 nós, em que a seed só considerava os.time() (o arquivo server.lua foi modificado para isso) - podemos ver que a eleição demora bastante a acontecer, pois por muitos termos os nós ficam com o mesmo timeout; eventualmente um nó é eleito pois os timeouts passam a ser diferentes;
- nodes5-5-simulataneousElection.txt: simulação com 5 nós em que a seed também só considerava os.time(), mas o script test.sh foi alterado para que apenas dois nós tivessem a mesma seed. Os dois nós iniciaram a eleição ao mesmo tempo, enquanto os outros não, e nesse caso um recebeu 2 votos, mas outro teve o timeout de eleição e progrediu para o próximo termo; sendo assim, o primeiro candidato acabou se tornando follower (linha 58) e o outro candidato foi eleito no 2o termo.

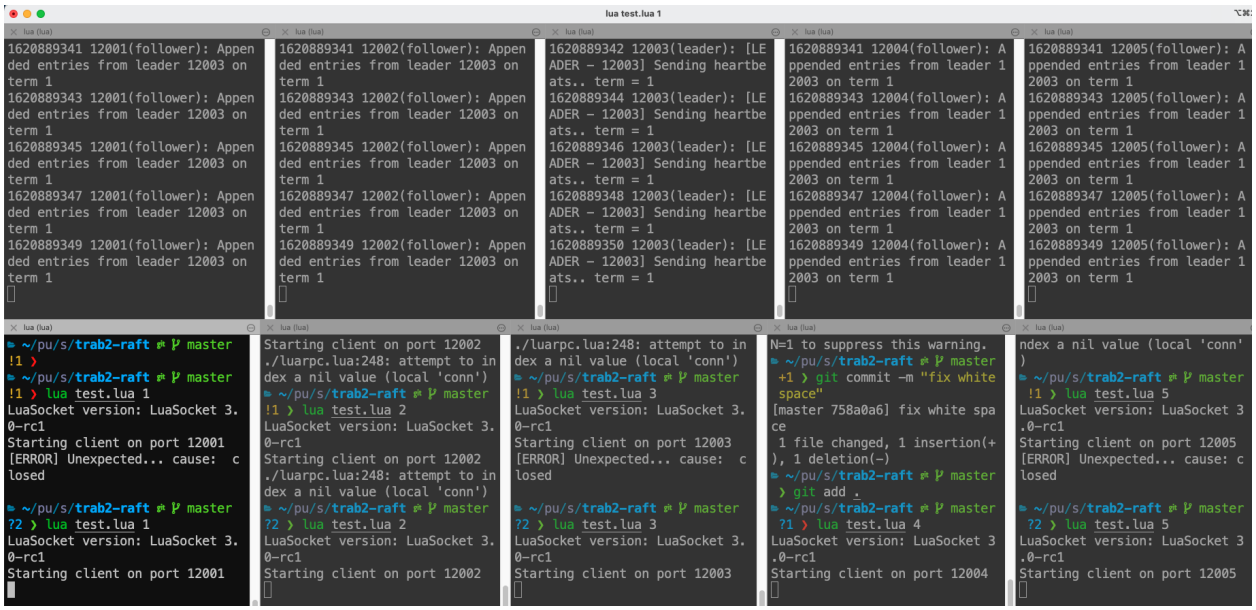

Figura 1: Screenshot de um testes manual realizado com várias janelas do Terminal