SDIS - Raft

Segundo projeto - Relatório

Afonso Bernardino da Silva Pinto - up201503316
Leonardo Gomes Capozzi - up201503708
Ricardo José Santos Pereira - up201503716
Tomás Sousa Oliveira - up201504746

Índice

---

1. Introdução

Este projeto foi realizado no âmbito da unidade curricular Sistemas Distribuídos, do 3º ano do Mestrado Integrado em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto. O nosso objetivo para este projeto foi criar uma livraria da linguagem de programação Java que implemente o Raft, um algoritmo de consenso, para replicação de logs em servidores. O presente relatório servirá para explorar as particularidades deste projeto e fornecer detalhes sobre a sua implementação.

2. Arquitetura

2.1. Raft

O Raft é um algoritmo que, a fim de aumentar a compreensibilidade, separa os elementos principais de um consenso e reforça um grau de coerência mais forte para reduzir o número de estados que devem ser considerados. Basicamente existem três estados que cada servidor pode ser (*Candidate*, *Follower* e *Leader*)

```java
public Raft(Integer port, InetSocketAddress cluster) {
    this.port = port;
    log.add(new RaftLog<T>( entry: null,  term: 0));

    // Connect to known cluster
    {
        SSLChannel channel = new SSLChannel(cluster);
        if (channel.connect()) {
            this.pool.execute(new RaftDiscover( raft: this, channel));
        } else {
            System.out.println("Connection failed!"); // DEBUG
            return; // Show better error message
        }
    }

    // Listen for new connections
    this.pool.execute(() -> {
        while (serverState.get() != ServerState.TERMINATING) {
            SSLChannel channel = new SSLChannel(port);
            if (channel.accept()) {
                pool.execute(new RaftServer<T>( raft: this, channel));
            }
        }
    });
}
```

Figura 1: Construtor do Raft

2.2. Estados

O *Leader* é escolhido através do processo *Leader Election*, eligido pelos outros servidores. Estes servidores começam como *Followers* quando são inicializados, e continuam a ser até pararem de receber *heartbeats* (*AppendEntries* RPC[1] sem logs) do *Leader*, durante um período de tempo aleatório, denonimado de *election timeout*. Caso isso aconteça, o *Follower* começa uma eleição, e transita para o estado *Candidate*. Logo depois, vota em si mesmo e envia um *RequestVote* RPC aos restantes servidores.

Um *Candidate* continua neste estado até que este ou outro servidor ganhe a eleição, ou um período de tempo passa sem vencedor. O *Candidate* ganha a eleição caso receba a maior parte dos votos dos restantes servidores.

Para o projeto, decidimos utilizar uma ENUM para representar os estados.

```java
public enum RaftState {
    FOLLOWER,
    CANDIDATE,
    LEADER
}
```

[1] Chamada remota de procedimento (*Remote Procedure Call)*

2.3. Reader & Writer

Foram criadas duas classes para ler (*RaftReader*) e escrever (*RaftWriter)* para os nós. O *RaftReader* está sempre à espera de de mensagens e quando recebe alguma, é lida a primeira palavra da mensagem. Essa palavra indica quais dos tipos de mensagem o programa pode ter recebido (algum servidor realizou uma das funções *callAppeandEntries* ou *callRequestVote* ou respondeu a uma dessas funções). Dependendo do tipo de mensagem, são aplicadas determinadas alterações ao estado do raft, para corresponder ao que o algoritmo necessita.

O *RaftWriter* escreve e envia as mensagens para os outros nós.

2.4. RPC

Depois de um líder ser eleito, este fornece solicitações com comandos a serem executadas pelas máquinas de estados replicadas.

O nosso programa utiliza chamadas remotas de procedimento para enviar mensagens entre processos. A função call*AppendEntries* é invocada pelo *Leader* para replicar entradas de logs e para enviar *heartbeats*.

```
static String callAppendEntries(Raft server) {
    StringBuilder message = new StringBuilder(callAppendEntriesRPC).append("\n")
            .append(server.currentTerm.get()).append("\n")
            .append(server.ID.toString()).append("\n");

            if(server.log.size() == 0) {
                message.append(0).append("\n")
                        .append(0).append("\n");
            }
            else {
                message.append(server.log.size() - 1).append("\n")
                        .append(((RaftLog) server.log.get(server.log.size() - 1)).term).append("\n");
            }

            message.append(server.commitIndex.toString()).append("\n");

    System.out.println("Append Entry: " + message.toString());

    return message.toString();
}
```

A função *callRequestVote* é invocada pelos *Candidates* para recolher votos dos outros servidores.

```
static String callRequestVote(Raft server) {
    StringBuilder message = new StringBuilder(callRequestVoteRPC).append("\n")
            .append(server.currentTerm.get()).append("\n")
            .append(server.ID.toString()).append("\n")
            .append(server.log.size()-1).append("\n")
            .append(((RaftLog) server.log.get(server.log.size()-1)).term).append("\n");

    System.out.println("Request Vote: " + message.toString());

    return message.toString();
}
```

3. Conclusão

Com a realização deste trabalho pudemos familiarizar-mos-nos com sistemas de processamento distribuídos. Apesar de não termos tido tempo para realizarmos o trabalho como desejado, consideramos que fizemos os possíveis para ter o algoritmo devidamente implementado.

4. Melhoramentos

Por falta de tempo, não implementámos certas funcionalidades que podiam ter sido inseridas no projeto:

- Utilizar HTTP;
- Implementar o algoritmo no primeiro trabalho (Distributed Backup Service)

5. Bibliografia

"The Raft Consensus Algorithm." Raft Consensus Algorithm. Acessado a 28 de Maio, 2018. https://raft.github.io/.