# Algoritmos e Sistemas Distribuídos – Trabalho Prático 2

João Reis Nº43914, Luís Correia Nº42832, Miguel Anciães Nº43367

Dezembro 18, 2017

## Abstrato

Esta fase do projecto consistiu em reutilizar o Serviço em Membership previamente implementado na 1ª fase, adicionando a cada Processo uma camada de Storage, que suporta duas operações distintas - **Read** e **Write** - feitas por um Cliente, e garantindo a sua Replicação. O serviço terá que assegurar que, em caso de falha, não se perde a informação contida na camada de Storage do Processo que falhou, disseminando-a para outro Processo que estará responsável de guardar essa informação, isto através de um Algoritmo de Consensus que suporta State Machine Replication, dado nas aulas.

A estrutura dos processos passará agora a ser constituída pelas anteriores 3 camadas e uma nova camada de Storage:

- Partial View membership
- Information dissemination
- Global membership
- Storage

Para a implementação do Algoritmo de Consensus, que garante que a informação de um Processo, caso falhe, não se perca, será feita através do **Paxos**. Por definição, um Processo tem três tarefas a executar neste algoritmo:

- **Proposer** - responsável por propor os valores/operações a executar
- **Acceptor** - recebe os valores propostos pelos Proposers e decide qual o valor/operação que tem maior valor, e portanto, qual o par a executar
- **Learner** - aprendem o valor sugerido pelos acceptors, e portanto aprendem o valor aceite pelo Sistema atingindo uma decisão consensual

# 1. Introdução

O objetivo deste trabalho consiste em desenvolver um serviço membership escalável, previamente implementado, que suporta Operações de Escrita e Leitura de valores da sua camada de Storage, e assegura que tal operações, uma vez efectuadas sobre um Processo, não são perdidas caso o mesmo Processo falhe, através da Replicação dessa mesma camada.

Para realizar este trabalho será preciso focarmo-nos nos algoritmos a implementar falados durante as aulas, de seguida iremos explicar quais usámos, a razão pela qual os escolhemos e como os implementámos. A principal questão na resolução do problema deste trabalho é como garantir a persistência das Operações que um Processo fez no Sistema no caso de ocorrer uma falha do processo, isto tudo de maneira eficiente, sem redundância e com robustez.

# 2. Implementação

O projeto foi desenvolvido em Scala com suporte do Akka, que nos ajudou a simular processos distribuídos e concorrentes num sistema.

Para a camada de **Storage**, usou-se um HashMap responsável por conter a informação do Sistema, TreeMaps para guardar as Réplicas de cada Processo e também as StateMachines dos Processos pelos quais cada Processo é responsável por assegurar a persistência da camada Storage, em caso de falha.

Para a implementação do Paxos, usámos o Algoritmo **Multi-Paxos**, em que só existe um Proposer, optimizando assim o tempo de execução e o número de mensagens trocadas no Sistema, visto que este algoritmo dispende a primeira fase de comunicação em que é decidido qual o valor/operação de maior número de série, ou seja, o mais actualizado, isto porque é sempre decidido o valor do Proposer.

Os nós/processos são representados em Scala utilizando Strings do tipo '***akka.tcp://AkkaSystem@127.0.0.1:2551***' onde o nome 'AkkaSystem' é o nome do sistema akka onde o nó corre, seguido do seu IP e porta.

## 2.1 Storage

A camada **Storage** é responsável por guardar todas as operações de Escrita e Leitura feitas por parte dos clientes do serviço. A sua implementação teve como base um HashMap, que associa um **ID** de uma dada operação de Escrita à **Data** da mesma. O ID resulta da aplicação de uma função de Hash ao título dado à operação de Escrita, estando este ID um número inteiro limitado entre 0 e 1000.

Para o Sistema garantir o conceito de Replicação, atribuímos a cada Processo um ID, resultante da aplicação de uma função de Hash ao seu endereço (qualquer coisa do tipo 'akka.tcp://AkkaSystem@127.0.0.1:2551'), ordenando-os por este ID. Ainda nesta camada, criámos TreeMaps responsáveis por guardar tanto as Réplicas pelas quais cada Processo é responsável por replicar a Storage (2 réplicas com ID imediatamente maior que o seu), assim como guardar as Réplicas onde a Storage do próprio Processo está replicada (2 réplicas com ID imediatamente menor que o seu), sendo que cada um destes mapas associa o ID do Processo ao seu endereço. Implementámos também um TreeMap para as State Machines dos processos que cada Processo irá replicar, associando por isso o ID de cada um desses processos à State Machine do mesmo.

## 2.2 Paxos com State Machine Replication

Para a implementação do Paxos, seguimos o algoritmo fornecido nos slides, e adaptámo-lo ao Multi-Paxos, em que o Proposer que recebe a operação é o líder desse Paxos, e é ele o responsável por informar o Cliente da actualização desse valor.

Temos então 3 tipos de réplicas:

- Proposer
- Acceptor
- Learner

O Proposer envia Prepare para todos os Acceptors da sua lista de réplicas assim recebe uma operação pela primeira vez - se o mesmo Proposer receber outra operação já não precisará de fazer este Prepare, evitando assim uma fase de envio de mensagens a todas as réplicas, tornando todo o algoritmo mais eficiente. Os Acceptors, por sua vez, ao receberem o Prepare, vão verificar se o sequence

number do Prepare é o maior que alguma vez receberam, se sim atualizam o seu número de sequência e devolvem ao Proposer um Prepare_OK. A fase 1 (Prepare) está assim concluída, e o Proposer poderá começar a enviar Accepts para os Acceptors. Recordando, se um dado Proposer já tiver feito esta fase de Prepare, ao receber outra operação salta esta fase, começando logo na fase seguinte.

O Proposer, já com maioria de Prepare_OK, envia para todos os Acceptors das suas réplicas um Accept, que vão verificar se o sequence number desse Accept é o maior que alguma vez receberam (mais uma vez). Se sim, atualizam o valor, devolvem um Accept_OK (Accept_OK_P) para o Proposer, e enviam para cada Learner um Accept_OK (Accept_OK_L).

Os Learners ao receberem mensagem de um Acceptor, atualizam o seu sequence number e o valor da operação, caso o da mensagem seja maior que o seu. Se receber uma maioria de Accept_OK por parte dos Acceptors, o valor é decidido e envia-se à Storage para que esta atualize o seu estado de acordo com a operação decidida pelo Paxos.

Já o Proposer, ao receber o Accept_OK por parte de uma maioria de Acceptors, informa o Cliente (Camada Aplicacional) da realização da operação.

**Nota:** A Global View é agora também a responsável por iniciar a camada Storage - encaminha as operações dos clientes para esta - e por dar início à Replicação da mesma.

## 3. Camada Aplicacional

Foi implementada uma pequena aplicação com o objectivo de conseguir saber e interagir com os processo a correr no sistem. Esta aplicação permite obter informação corrente sobre o Sistema, através dos seguintes comandos:

- "***gv*** *<endereço do processo>*" : mostra a Global View do processo indicado
- "***pv*** *<endereço do processo>*" : mostra a Partial View do processo indicado
- "***ms*** *<endereço do processo>*" : mostra o número de mensagens de vários tipos do processo indicado
- "***write*** *<ID> <Data>*" : escreve na Storage um par em que um dado ID está associado a uma Data
- "***read*** *<ID>*" : lê da Storage o ID, e devolve a Data associada a esse mesmo ID
- "***buckets*** *<endereço do processo>*" : mostra as StateMachines do bucket onde o processo está inserido
- "***replicas*** *<endereço do processo>*" : mostra as réplicas do processo indicado (Front Replicas) e os processos responsáveis pela replicação da camada Storage do processo indicado (Back Replicas)
- "***test1***" : testa o sistema para 0 Writes e 1000 Reads, e retorna o tempo de execução do mesmo
- "***test2***" : o mesmo que o test1, só que para 100 Writes e 900 Reads
- "***test3***" : o mesmo que os dois pontos anteriores, só que para 500 Writes e 500 Reads

## 4. Resultados Práticos

Para avaliar a performance do nosso sistema definimos três situações de teste com escritas e leituras sob o sistema, os quais foram executados quatro vezes e calculados os seus valor médios com intenção de reduzir variações do ambiente de execução. Os testes foram realizados apenas numa máquina com apenas um cliente a realizar as operações. Nestes testes foram escritas mensagens de um array de caracteres de tamanho entre 58 e 60 caracteres.

| Dimensão de teste | Tempo de execução |
|---|---|
| 0 - Writes 1000 - Reads | 0.062 seg |
| 100 - Writes 900 - Reads | 0.068 seg |
| 500 - Writes 500 - Reads | 0.087 seg |

A execução do sistema foi testada com cinco nós, ordenados pelas suas hashs como se vê na Fig.1.

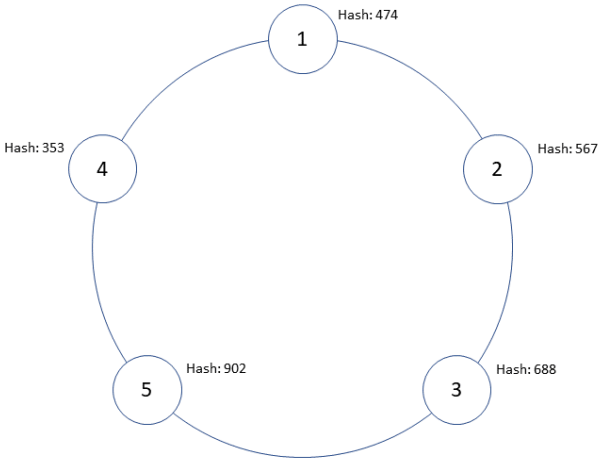


Fig.1 - Esquema dos processos usados na realização dos testes

No seguimento da ordenação dos nós, numa operação write é feita a replicação do novo estado após a escrita através de Paxos. Como explicado anteriormente, nesta operação a escrita actualiza as state machines das réplicas

com hashs imediatamente anteriores mantendo assim actualizados os buckets destas réplicas (Fig.2).

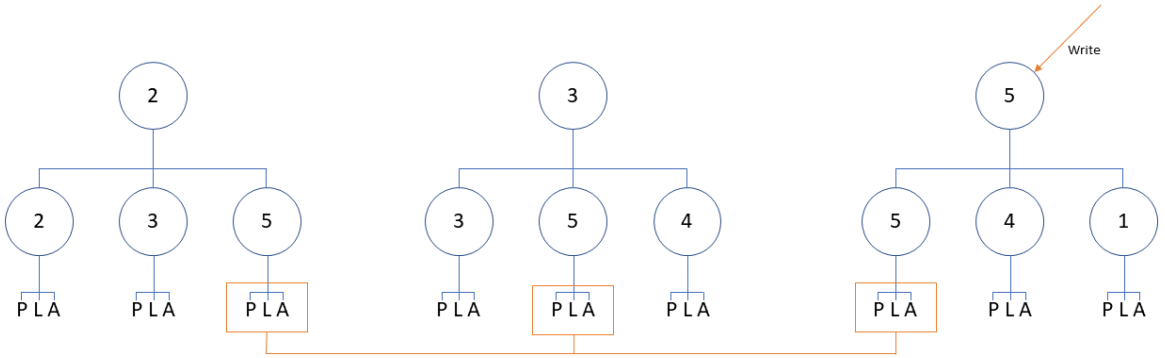


Fig.2 - Exemplo de uma escrita no processo 5 e a actualização das State Machines de cada réplica através de Paxos

## 5. Pseudo-Código (State Machine Replication)

### Storage

**State**:

- **storage** //HashMap que guarda um DataID associado a uma Data
- **pending** //Fila de espera para as operações do Cliente
- **replicasFront** //TreeMap que contém as réplicas que o nó é responsável por replicar
- **replicasBack** //TreeMap que contém as réplicas responsáveis por replicar o nó
- **stateMachines** //TreeMap que contém as StateMachines das réplicas que o nó é responsável por replicar
- **myself** //Endereço do nó
- **myselfHashed** //ID do nó, de 0 a 1000

**Upon** InitReplication (`replicasFront, replicasBack, selfAddress, myselfHashed, node, nodeHashed`) **do**:

```
    myself <- selfAddress
    myselfHashed <- myselfHashed
    newProcess <- node
    newProcessHash <- nodeHashed
    replicasFront <- replicasFront
    replicasBack <- replicasBack


    //quando um outro nó vem acima e passa a ser responsável por uma certa data, transfere a data para esse novo nó
```

```
      foreach st <- stateMachines do
            if !replicasFront.contains(st._1) then
                  transferData(st, newProcessHashed, newProcess)
      //adiciona os nós que é responsável por replicar ao mapa de
stateMachines

      foreach r <- replicasFront do
            if !stateMachines.contains(r._1) then
                  stateMachines.put(r._1, new StateMachine(myself,
r._1, replicasBack, context.system))


      //assim que um nó é criado, pede às suas réplicas que terá
que replicar as suas respectivas State Machines

      if myself.equals(newProcess) then
            foreach r <- replicasFront do
                  if r._1 != myselfHashed then
                        Trigger GetStateMachine

      //actualiza as réplicas que replicam a sua Storage
      foreach st <- stateMachines do
            Trigger setNewReplicas(replicasBack)
```

**Upon** ForwardWrite (`dataId, data`) **do**:

```
//seleciona o lider, que vai ser o Proposer do processo que é
responsável pela operação, e começa o Paxos

op <- Operation("write", write.hashedDataId, write.data)
pending.enqueue(op)
leader <- stateMachines.get(myselfHashed).get
Trigger initPaxos(op, myselfHashed)
```

**Upon** ForwardRead (`dataId`) **do**:

```
//se o dataId estiver na sua storage, devolve a data

if storage.contains(read.hashedDataId.toString) then
 //Send back to Application
      Trigger ReplyStoreAction("Read", myself,
storage.get(read.hashedDataId.toString).get)

else
 //Send back to Application
```

```
      Trigger ReplyStoreAction("Read", myself, "Read not found in
the System!")
```

**Upon** ExecuteOP(`opType, opCounter, hashDataId, data, leaderHash`) **do**:

```
if myselfHashed == leaderHash then
 try
   if opType.equals("delete") then
     storage <- storage / hashDataId.toString
   if opType.equals("write") then
     storage.put(hashDataId.toString,data)
   pending.dequeue()
  catch {
   case ioe: NoSuchElementException => //
   case e: Exception => //
  }
hashToMatch <- hashDataId
if opType.equals("delete") then
  hashToMatch <- leaderHash

//procura o nó responsavel
stateHash <- matchKeys(hashToMatch, stateMachines)
stateMachines.get(stateHash).get.writeOp(opType, opCounter,
hashDataId, data)
```

## Proposer

**State**:

- **n** //numero de sequencia da operação
- **biggestNseen** //maior numero de sequencia que o Proposer viu
- **prepared** //boolean que verifica se esse nó já fez a fase Prepare com as suas réplicas, de forma a que não volte a fazê-la enquanto for líder
- **majority** //boolean que verifica se houve maioria de replies, para haver Consensus
- **nPreparedOk** //conta o numero de prepareOk que recebeu dos Acceptors
- **replicas** //TreeMap que guarda as replicas do processo
- **op** //Operação dada para o Paxos
- **myselfHashed** //ID do processo
- **smCounter** //contador para indice das operações na state machine
- **nAcceptOk** //numero de AcceptOk que recebeu dos Acceptors das suas replicas

**Upon** InitPaxos (op, myselfHashed, replicas, smCounter) **do**:

```
replicas <- replicas
op <- op
myselfHashed <- myselfHashed
smCounter <- smCounter
n <- biggestNseen + 1
resetPaxos()

if !prepared then
 foreach r <- init.replicas do
   Trigger PrepareAccepter(n, op) //Para Acceptors
   Trigger InitPaxos             //Para Learners

else
 foreach r <- replicas do
     //Para Acceptors
   Trigger Accept(n, op, replicas, myselfHashed, smCounter)
```

**Upon** Prepare_OK (n, op) **do**:

```
nPreparedOk <- nPreparedOk + 1
if biggestNseen < n then
 biggestNseen <- n

if (nPreparedOk > replicas.size / 2) && !prepared then
 prepared <- true
 foreach r <- replicas do
     //Para Acceptors
   Trigger Accept(n,op, replicas, myselfHashed, smCounter)
```

**Upon** Accept_OK (n, op) **do**:

```
nAcceptOk <- nAcceptOk + 1
if nAcceptOk > replicas.size / 2 && !majority then
 majority <- true
     //Envia informaçao que foi feita a Operação
 Trigger ReplyStoreAction(op.op, myself, op.data)
```

## Acceptor

**State**:

- **np** //maior numero de sequencia proposto
- **na** //maior numero de sequencia aceite
- **va** //valor aceite

**Upon** PrepareAccepter (`n, op`) **do**:

```
if n >= np then
  np <- n
     //Reply para o Proposer
  Trigger Prepare_OK(np, op)
```

**Upon** Accept (`n, op, replicas, leaderHash, smCounter`) **do**:

```
if n >= np then
 na <- n
 va <- op

 Trigger Accept_OK_P(n, va) //Reply para Proposer

 foreach r <- replicas do
     //Envia para Learners
     Trigger Accept_OK_L(na, va, replicas, leaderHash, smCounter)
```

## Learner

**State**:

- **decision** // valor recebido do acceptor, portanto, aceite ?
- **na**
- **va**
- **nAcceptOK**
- **majority**

**Upon** InitPaxos **do**:
    resetPaxos()

**Upon** Accept_OK_L (`n, op, replicas, leaderHash, smCounter`) **do**:

```
decision <- Operation("", 0, "")
na <- 0
va <- Operation("", 0, "")
nAcceptOK <- 0
majority <- false

if n >= na then
   na <- n
   va <- op

   nAcceptOK <- nAcceptOK + 1
       if nAcceptOK > replicas.size / 2 && !majority then
         majority <- true
         decision <- va
          //Envia para a Storage
          Trigger ExecuteOP(op.op, smCounter, va.key, va.data,
    leaderHash)
```

## 6. Conclusão

Para a realização deste trabalho foi-nos proposto o uso de uma linguagem de programação até agora desconhecida da nossa parte, Scala, que promove uma implementação igualmente eficaz mas menos trabalhosa que Java, e que juntamente o Akka permitiu-nos implementar e simular este Sistema. Para esta implementação recorremos a vários novos algoritmos e a diferentes abordagens para criar um Sistema Distribuído com Storage Replicada em vários processos, de forma a manter a coerência entre valores devolvidos a todos os utilizadores do sistema, mesmo em caso de falha por parte de algum processo pertencente ao sistema, isto tudo sempre com vista a garantir que cada processo corra de maneira mais eficiente possível. Entre os algoritmos que usámos estão (já referidos em cima) o algoritmo Multi-Paxos com State Machine Replication, implementado recorrendo ao que foi dado durante as aulas teóricas e práticas.

Quanto aos objectivos traçados, ficou por implementar:

- Não demos uso à fila de espera de operações

## 6. Bibliografia


https://www.scala-lang.org

http://docs.scala-lang.org/?_ga=2.262766031.2111072999.1509405960-672480768.1509405942

https://akka.io

https://akka.io/docs/


### 6.1 Plataforma de Desenvolvimento colaborativo de código

**Repositório Público em:** https://github.com/aanciaes/ASD-Project1