Universidade Federal
do Espírito Santo

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Processamento Paralelo e Distribuído – Turmas 01 e 02 (EARTE) – 2022/1

Prof. Rodolfo da Silva Villaça – rodolfo.villaca@ufes.br

Laboratório IV – Comunicação Indireta, Eleição e Coordenação Distribuída

1. Objetivos

- Experimentar a implementação de sistemas de comunicação indireta por meio de *middleware Publish/Subscribe* (Pub/Sub) com Filas de Mensagens;
- Realizar eleição em sistemas distribuídos por meio da troca de mensagens entre os participantes do sistema;
- Realizar votação sobre o estado de transações distribuídas por meio da troca de mensagens entre os participantes do sistema.

2. Conceitos Básicos

Publish/Subscribe é um padrão arquitetural onde existem os *Publishers* (Publicadores) que enviam as mensagens e os *Subscribers* (Assinantes) que recebem as mensagens. Em uma maneira mais prática, sempre que houver algum evento, o publicador vai enviar uma mensagem para que os assinantes sejam notificados. O padrão arquitetural *Publish/Subscribe* (Pub/Sub) se resume na imagem a seguir: existe um publicador, que vai enviar uma mensagem em um canal (*Channel*) que redistribui uma cópia da mensagem para cada assinante. Em nosso caso, o canal é representado por um conjunto de filas gerenciados pelo *broker*.

Nenhuma das partes se conhece, muito menos sabem os detalhes de suas implementações. Apenas conhecem o endereço do *broker*. O modelo de comunicação Pub/Sub pode trazer baixo acoplamento e facilidade de escalabilidade em um ecossistema de software. Por ser assíncrono não haverá problemas com *timeouts* ou quebras de processamento por erros encadeados.

Mosquitto vs RabbitMQ

Mosquito é um *broker* de mensagens Pub/Sub, com licença de software livre e que implementa o protocolo MQTT (*Message Queue Telemetry Transport*). É leve e adequado para uso em todos os dispositivos, desde computadores de placa única de baixo consumo (Raspberry Pi, Arduino, ESP32,

etc) até servidores completos. O protocolo MQTT fornece um método leve para implementação do modelo de comunicação indireta usando Pub/Sub.

O RabbitMQ também é um *broker* de mensagens Pub/Sub que oferece aos seus aplicativos uma plataforma comum para enviar e receber mensagens, e armazenamento persistente destas mensagens até serem recebidas pelos assinantes (comunicação assíncrona). *Advanced Messaging Queuing Protocol* (AMQP) é o protocolo base no qual foi construído o RabbitMQ. Mosquitto e RabbitMQ também pertencem à categoria "Filas de Mensagens" (*Message Queue*) da pilha de tecnologias para implementação de Sistemas Distribuídos baseados em Comunicação Indireta.

3. Funcionamento do Sistema

Você precisará construir um protótipo similar a um minerador de criptomoedas do tipo *bitcoin*, que consiste na solução de uma prova de trabalho baseada em *hashing*, usando modelo de comunicação indireta Pub/Sub e implementação em Python. Nesta tarefa você poderá escolher o *broker* a ser utilizado: Mosquitto OU RabbitMQ. Neste laboratório a arquitetura ainda seguirá o modelo descentralizado, ou seja, sem a existência de um servidor para coordenar as ações do sistema.

Todos os nós deverão manter, localmente, enquanto estiver em execução, uma tabela constantemente atualizada com os seguintes registros:

TransactionID	*Challenge*	*Seed*	*Winner*
int	int	str	int

- *TransactionID*: Identificador da transação, representada por um valor inteiro (32 bits). Use valores incrementais começando em 0 (zero);
- *Challenge*: Valor do desafio criptográfico associado à transação, representado por um número [1..128], onde 1 é o desafio mais fácil. Gere desafios aleatórios, ou sequenciais, experimente as diferentes abordagens;
- *Seed:* String que, se aplicada a função de hashing SHA-1, solucionará o desafio criptográfico proposto;
- *Winner*: Identificador do usuário que solucionou o desafio criptográfico para a referida *TransactionID* (mesma linha da tabela). Enquanto o desafio não foi solucionado, considere que esse identificador é igual a -1;

A tabela deverá ser impressa na tela sempre que houver alguma atualização. Na inicialização do sistema, todos os nós participantes deverão limpar essa tabela e iniciar com *TransactionID*=0. A definição das filas de mensagens usadas no sistema, com seus respectivos nomes, assinantes, publicadores e formatação e parâmetros passados no conteúdo de cada mensagem ficará sob responsabilidade de cada grupo. **É necessário documentar essa etapa no relatório final da atividade!**

A Figura 1 mostra os 3 (três) estados iniciais do sistema, que deverá ser implementado em cada participante.

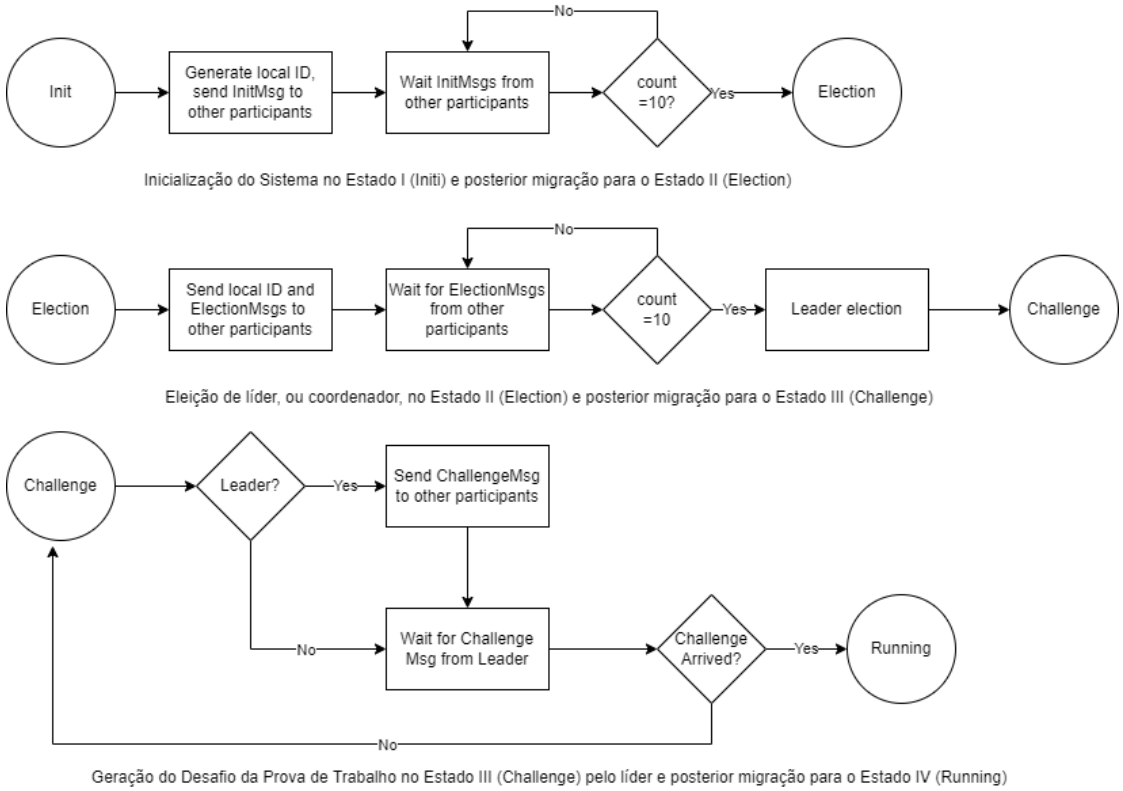


Figura 1: Estados de Inicialização (Init), Eleição (Election) e definição do Desafio (Challenge) de hashing criptográfico.

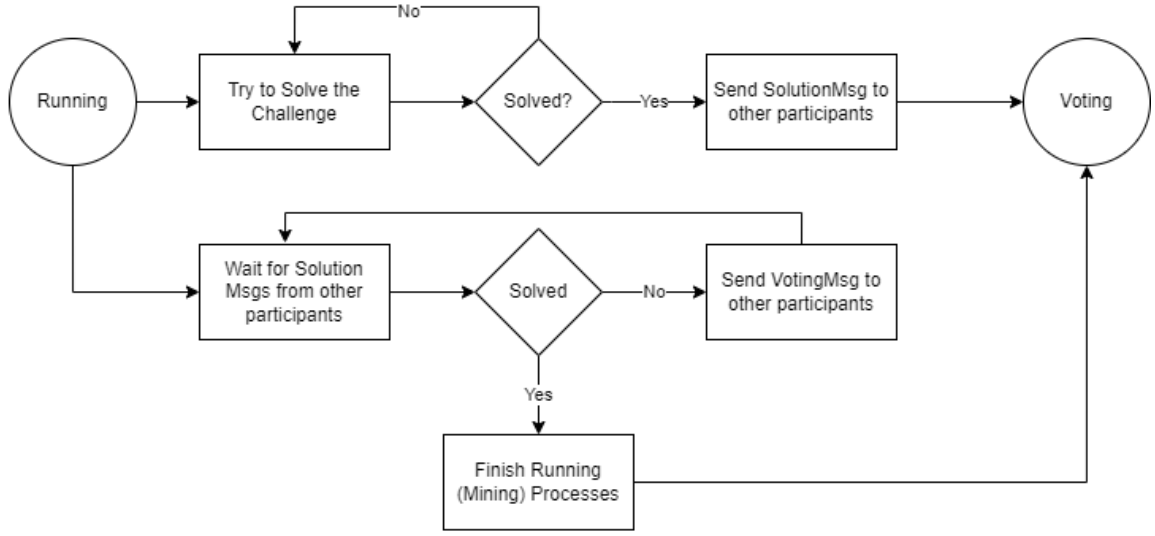
De acordo com a Figura 1, o sistema deverá inicializar sempre no estado Init, onde cada participante terá o seu identificador próprio na rede (*NodeID, LocalID, ClientID*, ou outro nome, defina e documente). Esse valor deverá ser gerado aleatoriamente. **O tamanho em bits do ID é de livre escolha pelos grupos, porém a especificação deve ser a mesma em todos os nós e lembre-se de documentar isso no relatório final da atividade!** Após gerar o seu ID, o nó participante deverá enviar seu ID para os demais por meio da publicação de uma mensagem (InitMsg na figura, mas pode ser outro nome – documentar) no *broker* de comunicação. Considere a existência de um *broker* único para todos os participantes.

Ainda de acordo com a Figura 1, após o envio da sua própria mensagem de inicialização, o nó deverá aguardar a chegada de um total de *n* (*n*=10 neste laboratório) mensagens de inicialização distintas, provenientes dos demais participantes. Ao receber um total de 10 (ou 9, dependendo da implementação, documente!) encerra-se a fase de inicialização. O nó poderá reenviar sua própria mensagem de inicialização enquanto as *n* demais mensagens não forem recebidas. Documente suas escolhas!

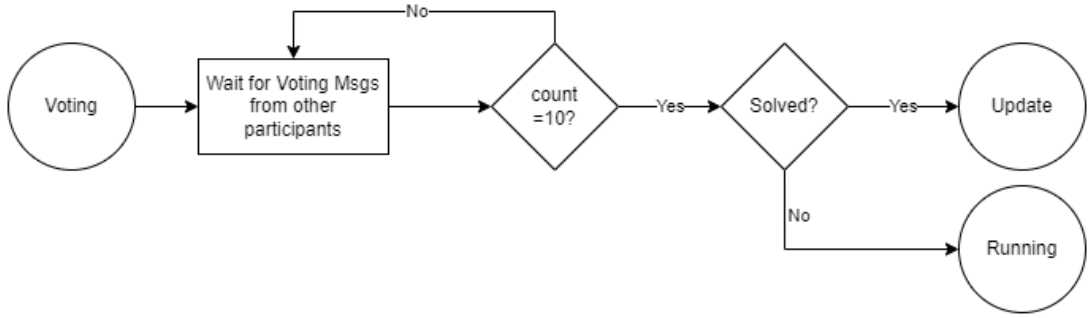
Após concluir a fase de inicialização (*Init*), tem-se inicio a fase de eleição do coordenador (ou líder) do grupo (*Election*). Nesta fase casa nó deverá gerar um número aleatório (documentar) e enviar ao

grupo por meio de uma mensagem de eleição, contendo seu ID e seu "voto" na eleição (número aleatório). Após enviar seu "voto", cada nó deverá aguardar o recebimento dos votos dos demais participantes do sistema (*count*=10 na Figura 1). Após receber todos os votos, deve-se escolher o participante com maior voto como líder do grupo e encerrar essa fase. Use uma combinação de ID + voto para decidir sobre eventuais desempates entre os participantes, sendo importante documentar todo esse processo de escolha do líder, inclusive e principalmente caso o processo seja feito de outra forma.

Considerando que todos os participantes estão sincronizados, muda-se para o estado de desafio (*Challenge*) e o líder pode definir um desafio e enviar por meio de mensagem (*ChallengeMsg*, na Figura 1) para todos demais participantes, que deverão apenas aguardar o recebimento desta mensagem. Ao receber a mensagem e certificar-se de que ela foi enviada pelo líder, procede-se a execução do processamento para solução do desafio, migrando para o estado *Running* da Figura 1. A Figura 2 tem o estado *Running* como estado inicial.



Execução do desafio no Estado IV (Running), que necessariamente requer paralelismo ou concorrência para: i) aguardar soluções provenientes de outros participantes eii) gerar sua próprias soluções. Dois estados possíveis de destino: Runnning e Voting.



No Estado V deve-se aguardar os votos dos demais participantes e verificar se a maioria foi atingida. Em caso positivo, seguir para o Estado VI (Update), caso contrário, retroceder para o Estado IV (Running) e continuar a resolver o desafio.

Uma vez que um desafio foi solucionado por algum participante, deve-se proceder a atualização do estado de cada nó (Update) e aguardar novo desafio (Challenge)

Figura 2: Estados de Execução/Processamento (Running), Votação (Voting) e Atualização (Update).

No estado *Running* (Figura 2) cada participante pode usar qualquer estratégia local para resolver o desafio, seja criar múltiplos processos, seja criar zumbis remotos por RPC que farão o processamento do *hashing* criptográfico correspondente ao desafio. **Importante: documentar esta etapa, o uso de multiprocessamento e distribuição da tarefa em outros nós "zumbis" vale ponto extra!!**

Observe que no estado *Running* cada nó deverá, em paralelo, aguardar mensagens de outros participantes que correspondam à solução do desafio (*SolutionMsg* na Figura 2). Ao receber uma mensagem de solução proveniente de algum outro participante, cada nó deverá verificar a solução proposta e votar pela aprovação ou reprovação da solução (documentar). O voto é obrigatório em casa situação.

Se um nó local encontrar uma solução deverá enviar sua proposta de solução para os demais participantes por meio de mensagem e sugere-se aguardar os votos dos demais participantes no estado *Voting*, conforme ilustrado na Figura 2. Neste estado (*Voting*) deve-se aguardar o voto dos demais participantes e, após atingir a maioria dos votos pela aprovação, encerra-se o estado *Running* e todos seus processos e atualiza-se a tabela de controle no estado *Update.*

Por fim, se aprovada a solução, o líder poderá definir um novo desafio, retornando ao estado *Challenge*, conforme Figura 2.

Como requisitos e premissas, em seu projeto assuma que:
a) O número de participantes é fixo e conhecido, no caso, n=10;
b) Não haverá entrada/saída dinâmica de nós no sistema (*churn*);
c) O *broker* Pub/Sub é único, infalível, e de conhecimento prévio por todos os participantes;
d) Os nós participantes do sistema são bem comportados, não havendo comportamento malicioso para atrapalhar o funcionamento;

4. Instruções Gerais

1. O trabalho pode ser feito em grupos de 2 ou 3 alunos: não serão aceitos trabalhos individuais ou em grupos de mais de 3 alunos;
2. Os grupos deverão implementar os trabalhos usando Python;
3. Data de Entrega: 15/07/2022, 23:59h;
4. A submissão deverá ser feita por meio de um *link* com a disponibilização dos códigos no Github, em modo de acesso público ou, minimamente, que meu e-mail rodolfo.villaca@ufes.br tenha direito de acesso ao código;
5. A documentação [1]deverá ser feita na própria página do Github através do arquivo README;
6. O grupo deverá gravar um vídeo de no máximo 5 min apresentando o funcionamento/teste do trabalho.

Bom trabalho!

1Atente-se que desta vez a documentação requer mais detalhes!