Universidade Federal da Bahia - UFBA

Linsmar da Silva Vital
Pedro Hugo Passos da Silva Carlos

DESCRIÇÃO TÉCNICA DA IMPLEMENTAÇÃO DO ALGORITMO PAXOS

Salvador - Bahia
2024

Contents

1 Visão Geral

O algoritmo Paxos é um protocolo fundamental para alcançar consenso em sistemas distribuídos, especialmente em cenários onde os nós podem falhar ou as mensagens podem ser perdidas. A implementação apresentada simula o Paxos em um ambiente local, usando processos paralelos e comunicação por sockets para ilustrar como os Proposers e Acceptors interagem para alcançar o consenso, mesmo com a presença de falhas simuladas. Abaixo está uma descrição detalhada da implementação, cobrindo a arquitetura do sistema, o funcionamento do algoritmo, as justificativas de projeto, e as instruções para compilar e executar o código.

2 Arquitetura do Sistema

O sistema é implementado usando o algoritmo Paxos distribuído para atingir consenso entre vários nós, mesmo na presença de falhas. A arquitetura do sistema inclui as seguintes partes:

- **Processos Paxos (Nós)**: Cada processo representa um nó Paxos que pode atuar como propositor ou aceitador durante a execução do algoritmo. Cada nó é iniciado como um processo separado utilizando a biblioteca `multiprocessing` do Python.
- **Comunicação entre Nós**: A comunicação entre os nós é gerenciada através de sockets ZeroMQ (`zmq`), onde cada nó cria um socket PULL para receber mensagens e vários sockets PUSH para enviar mensagens a outros nós.
- **Módulos do Sistema**:
  - `main.py`: Controla a inicialização dos processos (nós) e gerencia o fluxo principal do sistema.
  - `paxos_node.py`: Define o comportamento de cada nó Paxos, incluindo as fases do algoritmo Paxos.
  - `communication.py`: Gerencia o envio e EnvioEmMassa de mensagens entre os nós, com a opção de simular falhas de comunicação.
  - `config.py`: Contém a configuração básica do sistema, como a porta base para comunicação entre os nós.
- **Sincronização**: Utiliza a barreira (`Barrier`) para sincronizar as fases entre os processos/nós.

3 Explicação Passo a Passo do Algoritmo

3.1 Inicialização do Sistema (`main.py`)

A execução do programa começa na função `main`, que espera três argumentos de linha de comando:

- `num_proc`: Número de nós (processos) que participarão do protocolo.
- `prob_falha`: Probabilidade de falha na comunicação entre os nós, simulando um ambiente não confiável.

- `num_rodadas`: Número de rodadas que o protocolo será executado, cada rodada representando uma tentativa de chegar a um consenso.

A função valida os argumentos recebidos e cria uma barreira de sincronização (`Barrier`) que será usada para coordenar os nós, garantindo que todos os processos avancem de forma sincronizada nas diferentes fases do protocolo.

A função `criar_processos` é chamada para criar uma lista de processos do Python (`multiprocessing.Process`), onde cada processo representa um nó Paxos com um valor inicial aleatório. Cada processo é configurado para executar a função `PaxosNos` com os parâmetros necessários.

Os processos são iniciados pela função `iniciar_processos`, e o programa aguarda a conclusão de todos os processos com `aguardar_termino`.

3.2 Execução dos Nós Paxos (`paxos_node.py`)

Cada nó Paxos executa a função `PaxosNos`, que implementa as fases do protocolo Paxos. Um nó pode atuar como Proposer (proponente) ou Acceptor (aceitador), dependendo da fase e da rodada atual.

Configuração de Comunicação

- Os nós usam o ZeroMQ para comunicação. Cada nó cria um socket `PULL` para receber mensagens e múltiplos sockets `PUSH` para enviar mensagens para outros nós.
- O socket `PULL` é configurado com `bind` para escutar na porta específica do nó (baseada em seu ID), enquanto os sockets `PUSH` são configurados com `connect` para enviar mensagens aos outros nós.

Execução das Rodadas O protocolo é executado em múltiplas rodadas, onde em cada rodada, um nó é designado como Proposer (líder), responsável por propor um valor para consenso.

- A seleção do Proposer é baseada no índice da rodada (`r % numProc == id_no`). Isso garante que o papel de Proposer seja rotativo entre os nós.

Fase 1 - Proposta de Valor e Respostas (JOIN)

1. **Proposer Inicia a Fase**:
   - O Proposer envia uma mensagem "INICIAR" a todos os Acceptors, sinalizando o início da fase de proposta.
   - A mensagem é enviada usando a função `EnvioEmMassaComFalha`, que simula a possibilidade de falhas na transmissão com base na probabilidade especificada (`prob_falha`).
2. **Acceptor Recebe a Solicitação**:
   - Cada Acceptor ao receber "INICIAR", responde com uma mensagem "JOIN", indicando que está participando da votação, ou "FALHA" se uma falha de comunicação for simulada.

- A resposta é enviada ao Proposer usando a função `enviarComFalha`, que decide aleatoriamente, com base na probabilidade, se a mensagem chegará corretamente ou se será substituída por uma mensagem de falha.

3. **Proposer Avalia as Respostas**:
   - O Proposer coleta as respostas dos Acceptors. Se a maioria dos Acceptors (mais da metade) responde com "JOIN", o Proposer decide se irá propor um valor.
   - O valor a ser proposto pode ser o valor inicial do Proposer ou um valor que tenha sido anteriormente votado pela maioria dos Acceptors.
4. **Decisão de Propor ou Mudar de Rodada**:
   - Se o Proposer decide propor, ele envia uma mensagem "PROPOSE" a todos os Acceptors. Caso contrário, envia uma mensagem "MUDARODADA", indicando que a rodada será reiniciada.

Fase 2 - Proposição e Votação (PROPOSE e VOTO)

1. **Proposer Envia Proposição**:
   - Se o Proposer opta por propor um valor, ele usa `EnvioEmMassaComFalha` para enviar a mensagem "PROPOSE" com o valor proposto.
   - Caso a rodada deva ser reiniciada, o Proposer envia "MUDARODADA" sem falha usando `EnvioEmMassaSemFalha`, pois esta mensagem não deve ser perdida.
2. **Acceptor Recebe a Proposição**:
   - Ao receber "PROPOSE", cada Acceptor decide se aceita o valor proposto enviando um "VOTO" de volta ao Proposer.
   - Se o Acceptor receber "MUDARODADA", ele se prepara para a próxima rodada sem votar.
3. **Contagem de Votos**:
   - O Proposer coleta os votos dos Acceptors. Se a maioria votar a favor do valor proposto, o Proposer considera o valor como o consenso e declara sua decisão.
   - A decisão é então impressa no console.
4. **Sincronização**:
   - Ao final de cada fase (proposta e votação), todos os nós se sincronizam usando a barreira (`barreira.wait()`). Isso garante que todos os nós avancem para a próxima fase ou rodada de forma coordenada.

Finalização das Rodadas Esse processo se repete para o número de rodadas especificado. Em cada rodada, um novo Proposer é selecionado e o processo de proposta e votação é reiniciado.

3.3 Simulação de Falhas na Comunicação (`communication.py`)

A comunicação entre os nós simula um ambiente não confiável, onde mensagens podem falhar com base em uma probabilidade configurada.

Funções principais:

- `enviarComFalha`: Envia uma mensagem com apossibilidade de falha. Dependendo da probabilidade de falha especificada (`prob_falha`), a função decide se envia a mensagem original ou uma mensagem de erro.
- `enviarSemFalha`: Envia a mensagem sem introduzir falhas, garantindo que a comunicação ocorra de forma confiável. Utilizado principalmente para mensagens críticas que não devem ser perdidas, como comandos de sincronização.
- `EnvioEmMassaComFalha` e `EnvioEmMassaSemFalha`: Funções que enviam mensagens a todos os nós (EnvioEmMassa), simulando um ambiente de comunicação com ou sem falhas, respectivamente.

4 Resultado de Exemplo

NÚMERO DE NÓS: 4, PROBABILIDADE DE FALHA: 0.0, NÚMERO DE RODADAS: 3

RODADA 0 INICIADA COM VALOR INICIAL: 65
ACCEPTOR 2 RECEBEU NA FASE DE JOIN: INICIAR
LÍDER 0 RECEBEU NA FASE DE JOIN: INICIAR
ACCEPTOR 1 RECEBEU NA FASE DE JOIN: INICIAR
ACCEPTOR 3 RECEBEU NA FASE DE JOIN: INICIAR
LÍDER 0 RECEBEU NA FASE DE JOIN: JOIN -1 None
LÍDER 0 RECEBEU NA FASE DE JOIN: JOIN -1 None
LÍDER 0 RECEBEU NA FASE DE JOIN: JOIN -1 None
LÍDER 0 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
ACCEPTOR 2 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
ACCEPTOR 1 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
ACCEPTOR 3 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
LÍDER 0 RECEBEU NA FASE DE VOTAÇÃO: VOTO
LÍDER 0 RECEBEU NA FASE DE VOTAÇÃO: VOTO
LÍDER 0 RECEBEU NA FASE DE VOTAÇÃO: VOTO
LÍDER 0 DECIDIU PELO VALOR: 65
RODADA 1 INICIADA COM VALOR INICIAL: 54
ACCEPTOR 0 RECEBEU NA FASE DE JOIN: INICIAR
ACCEPTOR 3 RECEBEU NA FASE DE JOIN: INICIAR
ACCEPTOR 2 RECEBEU NA FASE DE JOIN: INICIAR
LÍDER 1 RECEBEU NA FASE DE JOIN: INICIAR
LÍDER 1 RECEBEU NA FASE DE JOIN: JOIN 0 65
LÍDER 1 RECEBEU NA FASE DE JOIN: JOIN 0 65
LÍDER 1 RECEBEU NA FASE DE JOIN: JOIN 0 65
LÍDER 1 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
ACCEPTOR 0 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
ACCEPTOR 2 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
ACCEPTOR 3 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65

LÍDER 1 RECEBEU NA FASE DE VOTAÇÃO: VOTO
LÍDER 1 RECEBEU NA FASE DE VOTAÇÃO: VOTO
LÍDER 1 RECEBEU NA FASE DE VOTAÇÃO: VOTO
LÍDER 1 DECIDIU PELO VALOR: 65
RODADA 2 INICIADA COM VALOR INICIAL: 83
ACCEPTOR 0 RECEBEU NA FASE DE JOIN: INICIAR
ACCEPTOR 3 RECEBEU NA FASE DE JOIN: INICIAR
ACCEPTOR 1 RECEBEU NA FASE DE JOIN: INICIAR
LÍDER 2 RECEBEU NA FASE DE JOIN: INICIAR
LÍDER 2 RECEBEU NA FASE DE JOIN: JOIN 1 65
LÍDER 2 RECEBEU NA FASE DE JOIN: JOIN 1 65
LÍDER 2 RECEBEU NA FASE DE JOIN: JOIN 1 65
LÍDER 2 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
ACCEPTOR 0 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
ACCEPTOR 1 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
ACCEPTOR 3 RECEBEU NA FASE DE VOTAÇÃO: PROPOSE 65
LÍDER 2 RECEBEU NA FASE DE VOTAÇÃO: VOTO
LÍDER 2 RECEBEU NA FASE DE VOTAÇÃO: VOTO
LÍDER 2 RECEBEU NA FASE DE VOTAÇÃO: VOTO
LÍDER 2 DECIDIU PELO VALOR: 65

5 Justificativa das Escolhas de Projeto

- **Uso do Multiprocessing**: Cada nó é representado como um processo separado para simular um ambiente distribuído, garantindo a independência e a paralelização de cada nó.
- **ZeroMQ para Comunicação**: `zmq` é utilizado para comunicação eficiente entre processos com suporte para padrões de comunicação PULL/PUSH, ideal para o modelo de mensagens usado pelo algoritmo Paxos.
- **Simulação de Falhas**: As funções `enviarComFalha` e `EnvioEmMassaComFalha` permitem simular falhas de comunicação de acordo com a probabilidade de falha definida, replicando cenários reais em sistemas distribuídos.
- **Barreiras de Sincronização**: Uso de barreiras (`Barrier`) para garantir que todos os processos/nós estejam sincronizados na mesma fase antes de continuar, essencial para a execução correta do algoritmo Paxos.

6 Instruções de Compilação e Execução do Código

6.1 Pré-requisitos

- Python 3.x instalado.
- Biblioteca ZeroMQ (`pyzmq`) instalada (`pip install pyzmq`).
- Biblioteca NumPy instalada (`pip install numpy`).

6.2 Execução

1. Salve todos os arquivos do código em um diretório.
2. Abra o terminal na pasta onde os arquivos estão salvos.
3. Execute o comando para rodar o script principal:

```
python main.py <num_proc> <prob_falha> <num_rodadas>
```

7 Conclusão

Esta implementação do Paxos oferece uma simulação realista e detalhada de como o consenso é alcançado em sistemas distribuídos com falhas. Utilizando processos paralelos, comunicação por sockets e simulação de falhas, o código demonstra as complexidades e a resiliência do algoritmo Paxos, sendo uma excelente ferramenta de aprendizado e teste para entender os desafios de alcançar consenso em sistemas distribuídos.