Broadcast Algorithms

Guilherme Sousa Lopes - 535869
Lucas Rodrigues Aragão - 538390

Introdução

Motivação

- É comum em redes locais que, processadores compartilhem canais comuns de comunicação, como ethernet ou token ring. Todos os processadores possuem formas diretas de comunicação com os demais.
- Nesse sentido, é normal que um processador queira que sua mensagem seja transmitida para todos os outros.
- Para isso é utilizado o comando broadcast.

Comando broadcast

broadcast ch(m)

- Um processo Ti transmite uma mensagem m para todos os demais utilizando o comando acima.
- Equivalente a

```
co [i = 1 to n]

    send ch [i] (m);
```

- Broadcast não é atômico. Ou seja, duas mensagens transmitidas por dois processos A e B, podem ser recebidas em ordens diferentes pelos outros processos.

Relógios lógicos e ordenação de eventos

Motivação

- Imagine que uma série de mensagens são enviadas e recebidas por um processador, de modo que, a ordem cronológica dos eventos é essencial para o funcionamento correto do programa.

- Como podemos manter a ordem lógica no recebimento de mensagens?
  - A alternativa ideal seria utilizar relógios sincronizados em todos os processadores, dessa forma, poderíamos enviar as mensagens junto de um *timestamp*, que indicaria a ordem delas.
  - Infelizmente isso não é possível, tendo em vista que seria extremamente complexo e trabalhoso manter todos os relógios **perfeitamente** sincronizados em diversos processadores.

- Como contornamos essa situação?

Relógios lógicos

- A ideia que surge é **simular** um relógio físico, utilizando relógios lógicos. Dessa forma é possível associar um valor de tempo a cada mensagem enviada por um processador.

- Relógios lógicos são simplesmente contadores inteiros, incrementados seguindo determinadas regras.

- Cada processador manteria um como variável local, que seria inicializada em 0.

Regras de Atualização dos Relógios lógicos

- Existem certas regras para a “passagem de tempo” de um relógio lógico, elas são:
- Dado um processo **A** e um relógio lógico **lc**, **A** atualiza **lc** quando:
  - **(1) A** envia ou transmite (broadcast) uma mensagem, colocando o timestamp da mensagem como o valor atual de **lc**, e então incrementa **lc** em 1.
  - **(2) A** recebe uma mensagem com timestamp **ts,** então atribui à **lc** o maior valor entre o valor atual de **lc**, e o timestamp ts + 1, também incrementando após isso **lc** em 1.
- Como o valor de lc é incrementado a cada evento, cada mensagem passada por A terá um **lc** maior que o anterior. Dessa forma garantimos que, se um evento x ocorre antes de um evento y, seu valor do relógio será menor, ocasionando uma ordenação parcial do conjunto de eventos “causally related”, que pode virar uma ordenação total ao utilizar o id do processo como critério de “desempate” caso tenha mensagens com timestamps iguais.

Caso prático - Distributed Semaphores

Motivação

- Semáforos normalmente são implementados como variáveis compartilhadas ou utilizando um processo **server** para controlar o acesso, esse segundo com menos frequência.

- A ideia que surge é criar um semáforo sem variáveis compartilhadas ou processo centralizador. Implementá-los de forma completamente distribuída.

- Semáforos basicamente são contadores inteiros que sempre precisam ser não negativos. Ou seja, precisamos que os processos utilizem os sem que violem a invariante básica.

Semáforos distribuídos

- Podemos fazer isso utilizando broadcast de mensagens.
- Cada processador terá sua fila para armazenar as mensagens e um relógio lógico.
- Para simular as mensagens de **P(sem)** e **V(sem)**, são feitos broadcasts de mensagens com o id do processo, uma *tag* que indica o tipo de operação que está sendo feita e o *timestamp* dela, esse último com o valor do relógio lógico.
- Isso seria suficiente para ter a ordem correta de execução de p e v, se broadcast fosse atômico, o que não é o caso.

Semáforos distribuídos

- Tipos de mensagem:
  - ACK: Sinalizador enviado ao receber alguma mensagem.
  - POP: Operação P de um semáforo.
  - VOP: Operação V de um semáforo.
  - reqV: Requisição (de uma operação V) enviada por um processo usuário ao seu processo Ajudante.
  - reqP: Requisição (de uma operação P) enviada por um processo usuário ao seu processo Ajudante.

- Cada processo ajudante mantém o predicado:
  - ***DSEM: s >=0 and mq is ordered by timestamps in messages***

Solução

```
type kind = enum(reqP, reqV, VOP, POP, ACK);
chan semop[n](int sender; kind k; int timestamp);
chan go[n](int timestamp);

process User[i = 0 to n-1] {
  int lc = 0, ts;
  ...
  # ask my helper to do V(s)
  send semop[i](i, reqV, lc); lc = lc+1;
  ...
  # ask my helper to do P(s), then wait for permission
  send semop[i](i, reqP, lc); lc = lc+1;
  receive go[i](ts); lc = max(lc, ts+1); lc = lc+1;
}
```

Solução

```
process Helper[i = 0 to n-1] {
  queue mq = new queue(int, kind, int);    # message queue
  int lc = 0, s = 0;                       # logical clock and semaphore
  int sender, ts; kind k;                  # values in received messages
  while (true) {    # loop invariant DSEM
    receive semop[i](sender, k, ts);
    lc = max(lc, ts+1); lc = lc+1;
    if (k == reqP)
      { broadcast semop(i, POP, lc); lc = lc+1; }
    else if (k == reqV)
      { broadcast semop(i, VOP, lc); lc = lc+1; }
    else if (k == POP or k == VOP) {
      insert (sender, k, ts) at appropriate place in mq;
      broadcast semop(i, ACK, lc); lc = lc+1;
    }
    else {  # k == ACK
      record that another ACK has been seen;
      for (all fully acknowledged VOP messages in mq)
        { remove the message from mq; s = s+1; }
      for (all fully acknowledged POP messages in mq st s > 0) {
        remove the message from mq; s = s-1;
        if (sender == i)     # my user's P request
          { send go[i](lc); lc = lc+1; }
      }
    }
  }
}
```

Implementação

O problema

- Para testar os semáforos distribuídos, implementamos eles para um problema simples.

- Processos tentam entrar em seção crítica e para isso precisam utilizar os semáforos distribuídos. Após conseguir o acesso, o processo dorme por um tempo aleatório e depois libera os semáforos.

- A implementação foi feita em python.