UNIVERSIDADE EDUARDO MONDLANE

FACULDADE DE CIÊCIAS

DEPARTAMENTO DE MATEMÁTICA E INFORMÁTICA

Licenciatura em Informática

Regime Laboral

Sistemas Distribuídos

Tema: Algoritmos Distribuídos

Discente:

Chirindza, André Domingos

Mazive, Simão Gabriel

Docente:

Dr. Helder MC Muianga

Maputo, Abril de 2017

Índice

1. Introdução

Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus utilizadores como um sistema único e coerente (Tanenbaum e Steen).
Processos em um sistema distribuído geralmente buscam atingir cooperação e para tanto utilizam mecanismos de sincronização para que esta cooperação seja realizada de maneira correta. Pois, são várias as questões que se levantam quando o assunto é mesmo sincronização, entre elas:

Em Sistemas Operativos, os problemas de exclusão mútua e região crítica são solucionados através de semáforos ou monitores. Tais métodos utilizam memória compartilhada para implementar a solução, portanto, impossível de ser feito em um Sistema Distribuído. Entretanto essa sincronização pode ser garantida por meios de Algoritmos Distribuídos, que serão apresentados em diante.

2. Sincronização De Relógio

A sincronização em sistemas distribuídos é mais complicada do que me sistemas centralizados porque em sistemas distribuídos é necessário utilizar algoritmos distribuídos. Não é usualmente possível (ou desejável) colectar todas as informações sobre o sistema em um único lugar e depois deixar que alguns processos examinem-na e tomem uma decisão como é feito no caso centralizado.

Em geral, os algoritmos distribuídos possuem as seguintes propriedades:

a) A informação relevante está espalhada em múltiplas máquinas;
b) Processos tomam decisões baseadas somente nas informações locais;
c) Um único ponto de falha no sistema deve ser evitado;
d) Não existe um relógio em comum ou outro tipo preciso de tempo global.

Os primeiros três pontos significam que é inaceitável colecionar todas as informações em um único local para o processamento. Por exemplo para fazer a alocação de recursos, não é aceitável enviar todas as requisições para um único processo gerenciador, o qual examina as requisições e permite ou nega as requisições com base em informaçoes contidas em suas tabelas.

2.1. Relógio Lógico

Quase todos os computadors possuem um circuito para manter-se a par do tempo. O timer utilizado nos computadores é geralmente produzido a partir do cristal de quartzo. Quando mantido sob tensão o cristal do quartzo oscila em frequência bem definida que depende do tipo do cristal. Lamport (1978), afirma que a sincronização dos relógios é possível e demostrou o algoritmo para se conseguir isto. Também diz que: a sincronização dos clocks não precisa ser absoluta, pelos seguintes motivos:

- Se dois processos não interagem, não é necessário que seus clocks sejam sincronizados.
- Usualmente o que importa não é que todos os processos concordem com o exato tempo em que os eventos aconteceram, mas que concordem na ordem em que os eventos ocorreram.

Ou seja, relação ao relógio lógico a consistência interna é o que importa e não quanto eles estão próximos do tempo real.

2.2. Relógio Físico

Em alguns sistemas (por exemplo, sistema de tempo-real), o tempo real é importante. Para estes sistemas o relógios físicos externos são necessários. Por razões de eficiência e redundância, muitos relógios físicos são geralmente considerados desejáveis, os quais trazem dois problemas:

i) Como será feita a sincronização deles com os relógios do mundo real;

ii) Como sincronizar os relógios entre si.

Visto esse problema percebe-se que, existem vários algoritmos desenvolvidos para gestão dos relógios, entre eles:

a) Algoritmo de Berkeley;
b) Network Time Protocol;
c) Algoritmo de Cristian.

3. Deadlock

Um *deadlock* é causado pela situação onde um conjunto de processos está boqueado permanentemente, isto é, não consegui prosseguir a execução, esperando um evento que somente outro processo do conjunto pode causar.

Condições necessárias para a ocorrência do *deadlock* no sistema:

a) Exclusão mútua;
b) Segura e espera;
c) Não preempção;
d) Espera circular.

3.1. Estratégias para o tratamento do *deadlock*:

- **Algoritmo do Avestruz** (ignorar o problema);
- **Detenção** (permite que o deadlock ocorra, detecta e tenta recuperar);
- **Prevenção** (estaticamente faz com que o deadlock não ocorra);

- **Espera circular** (evita deadlock alocando os recursos cuidadosamente).

4. Detector de falha

Segundo Coulouris (2012), qualquer que seja o tipo de falha, um processo correto é aquele que não apresenta falhas em qualquer ponto da execução em consideração. Observe que a correção se aplica a toda a execução, não apenas a uma parte dela. Assim, um processo que sofre uma falha de falha é 'não-falhado' antes desse ponto, não 'correto' antes desse ponto.

Um dos problemas no projeto de algoritmos que podem superar falhas de processo é o de decidir quando um processo caiu. Um detector de falhas [Chandra e Toueg 1996, Stelling et al. 1998] é um serviço que processa consultas sobre se um determinado processo falhou. É muitas vezes implementado por um objeto local para cada processo (no mesmo computador) que executa um algoritmo de detecção de falhas em conjunto com suas contrapartes em outros processos. O objeto local para cada processo é chamado de detector de falha local.

4.1. Propriedades dos detectores de falha

Um "detector" de falha não é necessariamente preciso. A maioria cai na categoria de detectores de falhas não confiáveis. Um detector de falhas não confiável pode produzir um de dois valores quando for dada a identidade de um processo: Insuspeitado ou suspeito. Ambos resultados são sugestões, que podem ou não refletir com precisão se o processo realmente falhou. Um resultado de *Unsuspected* significa que o detector recebeu recentemente evidências sugerindo que o processo não falhou; Por exemplo, uma mensagem foi recentemente recebida a partir dele. Mas é claro que o processo pode ter falhado desde então. Um resultado de *Suspected* significa que o detector de falhas tem alguma indicação de que o processo pode ter falhado. Por exemplo, pode ser que nenhuma mensagem do processo tenha sido recebida por mais do que um comprimento nominal máximo de silêncio (mesmo em um sistema assíncrono, limites superiores práticos podem ser usados como dicas). A suspeita pode estar mal colocada: por exemplo, o processo poderia estar funcionando corretamente, mas estar do outro lado de uma partição de rede, ou poderia estar funcionando mais lentamente do que o esperado.

Um detector de falhas confiável é aquele que é sempre preciso na detecção de falha de um processo. Responde a consultas de processos com uma resposta de *Unsuspected* - que, como antes, só pode ser uma dica - ou Falha. Um resultado de Falha significa que o detector determinou que o processo caiu. Lembre-se que um processo que caiu permanece dessa forma, já que por definição um processo nunca dá mais um passo depois que ele caiu.

É importante perceber que, embora falemos de um detector de falhas atuando para uma coleção de processos, a resposta que o detector de falhas dá a um processo é tão boa quanto a informação disponível nesse processo. Um detector de falhas pode às vezes dar respostas diferentes a diferentes processos, uma vez que as condições de comunicação variam de processo para processo.

Os detectores de falha nos ajudam a pensar sobre a natureza das falhas em um sistema distribuído. Em qualquer sistema prático que é projetado para lidar com falhas deve detectá-las - ainda que imperfeitamente. Mas verifica-se que mesmo os detectores de falhas não confiáveis mas com certas propriedades bem definidas podem nos ajudar a fornecer soluções práticas para o problema de coordenação de processos na presença de falhas.

5. Exclusão mútua distribuída

Os processos distribuídos muitas vezes precisam coordenar suas atividades. Se uma coleção de processos partilha um recurso ou colecção de recursos, muitas vezes a exclusão mútua é necessária para evitar interferências e garantir a coerência ao aceder os recursos. Esse é o problema da seção crítica, familiar no domínio dos sistemas operacionais. Em um Sistema distribuído, no entanto, nem as variáveis compartilhadas nem as facilidades fornecidas por um único *kernel* local podem ser usadas para resolvê-lo, em geral. Exigimos uma solução para a exclusão mútua distribuída: uma que se baseie unicamente na passagem de mensagens.

Em alguns casos, os recursos compartilhados são geridos por servidores que também fornecem mecanismos de exclusão mas, em alguns casos práticos, é necessário um mecanismo separado para a exclusão mútua.

Consideremos, um sistema de monitoramento do número de vagas em um parque de estacionamento com um processo em cada entrada e saída que rastreia o número de veículos

entrando e saindo. Cada processo mantém uma contagem do número total de veículos dentro do parque de estacionamento e mostra se está cheio ou não. Os processos devem atualizar a contagem compartilhada do número de veículos de forma consistente. Existem várias maneiras de conseguir isso, mas seria conveniente que esses processos fossem capazes de obter exclusão mútua unicamente através da comunicação entre si, eliminando a necessidade de um servidor separado.

É útil ter um mecanismo genérico para a exclusão mútua distribuída na nossa que é independente do regime específico de gestão de questão.

5.1. Algoritmos de Exclusão Mútua

Suponhamos N processos, pi, i=1,2,…,N que não partilham variáveis, mas que partilham algum recurso partilhado ao qual devem aceder numa secção critica (s.c.). Assumimos que o sistema é assíncrono, que os processos não falham e que a comunicação é fiável.

O protocolo ao nível da aplicação é o seguinte:

enter()*: entrar na secção crítica, bloquear se necessário*

resourceAccesses()*: aceder aos recursos partilhados na s.c.*

exit()*: sair da secção crítica, outro processo pode entrar*

Os requisitos para a exclusão mútua são os seguintes:

- EM1: (**segurança**)
  Apenas um processo de cada vez pode executar a s.c.
- EM2: (**nem impasse nem asfixia = *liveness***)
  Pedidos para entrar ou sair da secção crítica deverão ter sucesso num período razoável de tempo.

Por vezes é importante que:

- EM3: (**ordenação**)
  Se um processo pede o acesso à s.c. antes de outro, o acesso deve ser-lhe garantido primeiro.

Para avaliar a performance de cada algoritmo, vamos analisar:

- **Bandwidth,** a largura de banda consumida, proporcional ao número de mensagens enviadas em cada operação de entrada e saída da s.c.
- **Client** delay, que é o tempo necessário para o processo entrar na s.c.
- **Synchronization delay,** efeito do algoritmo sobre o throughput do sistema. Tempo entre um processo sair da s.c. e outro processo entrar.

5.1.1. Algoritmo Centralizado

A forma mais simples de garantir a exclusão mútua é ter um processo que garante a permissão para entrar na secção crítica; Para entrar na s.c. um processo envia ao servidor um pedido de entrada, a resposta do servidor constitui um "*token*" que significa permissão para entrar na secção crítica. Se nenhum processo detém o *token*, o servidor responde imediatamente, garantindo o acesso, se o *token* está na posse de algum outro processo o servidor não responde, colocando o processo em espera, quando um processo sai da s.c. envia uma mensagem ao servidor, devolvendo o *token*. Se a fila de processos em espera não está vazia, o servidor escolhe a entrada mais antiga na fila e envia o *token* a esse processo.

5.1.2. Token Ring Algorithm

Os processos são organizados em anel, segundo o seu IP ou outra numeração. Cada processo Pi tem de ter um canal de comunicação com o processo que se segue no anel P(i+1) mod N. O token que dá acesso à s.c. é passado de processo em processo numa única direcção.

Se um processo não pretende aceder à s.c. simplesmente passa o token ao seguinte. Um processo que queira aceder à s.c. espera pelo token, fica com ele enquanto acede à s.c. e no final passa-o ao processo seguinte.

Nota: este algoritmo não verifica a regra EM3.

- **Bandwidth**

  O algoritmo consome continuamente largura de banda (excepto quando está a aceder à s.c.)

- **Client delay**

O delay de um processo que quer entrar na secção crítica pode ir de **1 a N** mensagens.

- **Sinchronization delay**

O tempo de sincronização entre a saída de um processo da s.c. e a entrada de um novo pode ir de **1 a N** mensagens.

5.1.3. Ricart and Agrawala

Os processos, p1,p2, … pN têm identificadores distintos. Todos os processos podem comunicar com todos e cada processo possui um relógio Lógico de Lamport, que mudam de acordo com o relacionamento de "acontece antes" entre os eventos.

Mensagens de pedido para entrar na s.c. são da forma <T,pi>> onde T é o timestamp do emissor e pi o seu identificador.

Cada processo regista o seu estado como, RELEASED, fora da secção crítica, WANTED, quer aceder à secção crítica e HELD, está na secção crítica.

Se um processo pretende entrar na secção crítica envia o pedido a todos os processos (multicast). Se o estado de todos os processos for RELEASED todos respondem imediatamente e o processo acede à s.c. Se algum processo está no estado HELD, então esse processo não responde ao pedido até terminar a s.c. Se dois ou mais processos requerem o acesso à s.c. ao mesmo tempo, então o processo que tem o menor timestamp irá ser o primeiro a obter as N-1 respostas que lhe garantem o acesso à s.c. Se dois pedidos tiverem o mesmo timestamp, então os pedidos são ordenados pelo identificador do processo.

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
Multicast *request* to all processes; request processing deferred here
    T := request's timestamp;
Wait until (number of replies received = (*N* – 1));
    state := HELD;
On receipt of a request <*Ti, pi*> *at* p_j *(*$i \neq j$*)*
    if (*state* = HELD or (*state* = WANTED *and* (*T*, p_j) < (*Ti*, *pi*)))
    then
        queue *request* from *pi*

```
            without replying;
        else
            reply immediately to pi;
        end if
To exit the critical section
        state := RELEASED;
        reply to any queued requests;
```

- **Bandwidth**

Obter o acesso à s.c. necessita **2(N-1)** mensagens. N-1 para o pedido e N-1 respostas. (Se existir suporte em hardware para o Multicast, o pedido necessita de 1 mensagem + N-1 respostas =

total de N mensagens.

- **Client delay**

O atraso do cliente para obter o acesso é um round trip time.

- **Synchronization delay**

O tempo para um processo sair da s.c. e outro entrar é o tempo de envio de apenas uma mensagem.

6. Eleição de um líder (coordenador)

Um algoritmo para escolher um processo único para desempenhar uma determinada função, designa-se por eleição (election algorithm) é essencial que:

- Todos os processos concordem na escolha, geralmente é escolhido o processo com o identificador mais elevado. Os algoritmos tentam localizar o processo com maior identificador e designá-lo como coordenador.

Assumimos que todos os processos sabem o identificador dos outros processos, mas não sabem quais os processos que estão activos e quais não estão.

6.1. Bully algorithm

Desenvolvido por Garcia-Molina em 1982, assume que um processo sabe a prioridade de todos outros processos no sistema e a entrega de mensagens é fiável e funciona mesmo no caso em que um processo falhe durante e eleição.

Quando um processo não tem resposta de um coordenador, inicia uma eleição

- Quando P inicia uma eleição,
    - P envia a mensagem "ELECTION" para todos os processos com identificador (ID) superior ao seu.
    - Se nenhum processo responde, P vence a eleição e torna-se o coordenador. Envia a mensagem "Coordinator" para todos os outros
    - Se algum processo com maior ID responde, o papel de P na eleição termina.
- Quando um processo recebe a mensagem (ELECTION) de processos com ID mais baixo,
    - envia a mensagem "OK" para o remetente
- Quando um processo recebe a mensagem "COORDINATOR" regista o identificador do processo emissor como o novo coordenador.
- Se um processo é reinicializado para ocupar o lugar de um processo que avariou, vai iniciar uma eleição.
- Se esse processo tiver o maior dos identificadores, assume-se como o novo coordenador. Envia a mensagem "COORDINATOR" aos outros.

6.2. Ring Algorithm

Os processos são organizados num anel lógico e as mensagens são enviadas no sentido dos ponteiros do relógio.

- Se um processo verifica que o coordenador não funciona:
    - Envia mensagem de eleição ("ELECTION") para o seu sucessor
    - A mensagem contém o ID do emissor
- Se o sucessor não estiver disponível, envia para o seguinte e assim sucessivamente.
    - Em cada passo, o processo que recebe a mensagem, anexa o seu ID e envia para o processo seguinte.

- Vai circular novamente, agora para informar todos os processos de:
  - "quem é" o coordenador (processo com o maior ID)
  - "quem são" os novos membros do anel.
- Quando a mensagem volta novamente ao processo que iniciou a eleição, é removida.
- Quando a mensagem regressa ao processo que iniciou a eleição, o que é verificado porque contém o ID do processo, a mensagem é alterada para "COORDINATOR".

6.3. Comunicação Multicast

Segundo Coulouris (2012, p.646-647), O sistema distribuído contém uma coleção de processos, que podem se comunicar de forma confiável sobre canais um-para-um. Como antes, os processos podem falhar apenas por falha.

Os processos são membros de grupos, que são os destinos de mensagens enviadas com a operação *multicast*. Geralmente é útil permitir que os processos sejam membros de vários grupos simultaneamente, por exemplo, para permitir que os processos recebam informações de várias fontes unindo vários grupos. Mas, para simplificar nossa discussão sobre as propriedades de ordenação, às vezes restringiremos os processos a serem membros de, no máximo, um grupo de cada vez.

A operação *multicast* (g, m) envia a mensagem m para todos os membros do grupo g dos processos. Correspondentemente, existe uma operação *deliver* (m) que entrega uma mensagem enviada por *multicast* para o processo chamador. Usamos o termo entregar ao invés de receber para deixar claro que uma mensagem *multicast* nem sempre é entregue à camada de aplicativo dentro do processo assim que ele é recebido no nó do processo. Isso é explicado quando discutimos a semântica de entrega *multicast* em breve.

Cada mensagem m carrega o identificador exclusivo do remetente do processo (m) que enviou

E o grupo identificador de grupo de destino exclusivo (m). Assumimos que os processos não mentem sobre a origem ou destino das mensagens. Alguns algoritmos assumem que os grupos estão fechados.

6.3.1. *Multicast* Básico

É útil ter à nossa disposição uma primitiva multicast básica que garanta, ao contrário do IP *multicast*, que um processo correto acabará por entregar a mensagem, desde que a *Multicaster* não falha. Chamamos o primitivo B-*multicast* e seu correspondente entrega básica B-entrega primitiva. Permitimos que os processos pertençam a vários grupos, e cada mensagem é destinada a um grupo específico.

Uma maneira direta de implementar B-*multicast* é usar um *one-to-one* confiável

Enviar operação, como segue:

Para B-multicast (g, m): para cada processo pg, send (p, m);

Em receber (m) em p: B-entregar (m) na p.

A implementação pode usar *threads* para executar as operações de envio ao mesmo tempo, em uma tentativa de reduzir o tempo total levado para entregar a mensagem. Infelizmente, uma tal implementação é susceptível de sofrer de uma chamada implosão *ack* se o número de processos é grande. Os agradecimentos enviados como parte da operação de envio confiável são susceptíveis de chegar a partir de muitos processos em aproximadamente a mesma hora. Os *buffers* do processo de multidifusão serão preenchidos rapidamente, e é susceptível de deixar reconhecer. Consequentemente retransmitirá a mensagem, conduzindo a ainda mais reconhecimentos e desperdício adicional da largura de faixa da rede.

6.3.2. *Multicast* Confiável

Seguindo Hadzilacos e Toueg [1994] e Chandra e Toueg [1996], definimos um *multicast* confiável com operações correspondentes R-multicast e R-deliver.

Propriedades análogas à integridade e validade são claramente altamente desejáveis em *multicast*, mas adicionamos outro: um requisito de que todos os processos corretos no grupo deve receber uma mensagem se qualquer um deles faz. É importante perceber que esta não é uma propriedade do algoritmo B-*multicast* que é baseado em uma operação confiável *one-to-one* enviar. O remetente pode falhar em qualquer momento enquanto B-*multicast* prossegue, portanto, alguns processos podem entregar uma mensagem, enquanto outros não.

Um multicast confiável é aquele que satisfaz as seguintes propriedades:

- **Integridade**, um processo correto p entrega uma mensagem m no máximo uma vez. Além disso, P grupo m e m foi fornecido a uma operação *multicast* pelo remetente (m). (Como com comunicação um-para-um, as mensagens podem sempre ser distinguidas pelo número do remetente).
- **Validade**, se um processo correto *multicasts* mensagem m, então ele irá eventualmente entregar m.
- **Acordo**, se um processo correto entrega a mensagem m, então todos os outros processos corretos no grupo (m) acabará por fornecer m.

A propriedade de integridade é análoga àquela para uma comunicação um-para-um confiável. A propriedade de validade garante a vivacidade do remetente. Isso pode parecer uma propriedade incomum, porque é assimétrica (menciona apenas um processo particular). Mas observe que a validade e o acordo juntos somam uma exigência de vivacidade: se um processo (o remetente) finalmente entrega uma mensagem m, uma vez que os processos corretos concordam com o conjunto de mensagens que eles enviam, segue que m será eventualmente entregue a todos Os membros corretos do grupo.

A vantagem de expressar a condição de validade em termos de autoentrega é simplicidade. O que precisamos é que a mensagem seja entregue eventualmente por algum membro correto do grupo.

A condição de acordo está relacionada à atomicidade, propriedade de "tudo ou nada", aplicada à entrega de mensagens a um grupo. Se um processo que *multicasts* uma mensagem falhar antes de ter entregue, então é possível que a mensagem não seja entregue a qualquer processo no grupo; Mas se for entregue a algum processo correto, então todos os outros processos corretos o entregarão. Muitos artigos na literatura usam o termo "atômico" para incluir uma condição de ordenação total.

On initialization

Received := {};

For process p to R-multicast message m to group g

B-multicast(*g, m*); // is included as a destination

On B-deliver(*m*) *at process q with g = group*(*m*)

if (*m* ≠ *Received*)

then

Received := *Received m* ∨ *{m}*;

if (q≠p) then B-multicast(*g, m*); *end if*

R-deliver m;

end if

6.3.3. *Multicast* ordenado

O algoritmo multicast básico da Seção 15.4.1 entrega mensagens a processos em um Arbitrária, devido a atrasos arbitrários nas operações de envio um-para-um subjacentes.

Esta falta de uma garantia de encomenda não é satisfatória para muitas aplicações. Para Por exemplo, numa central nuclear pode ser importante que acontecimentos que Condições de segurança e eventos que significam ações por unidades de controle são observados na mesma ordem por todos os processos no sistema.

Os requisitos de ordenação comuns são ordenação total, ordenação causal e ordenação FIFO, juntamente com soluções híbridas (em particular, total-causal e FIFO total).

- Ordem FIFO: Se um processo correto emite multicast (g, m) e, em seguida, multicast (g, m), então cada processo correto que fornece m vai entregar m antes m
- Ordem causal: Se multicast (g, m) multicast (g, m), onde é o passado antes da relação induzida apenas por mensagens enviadas entre os membros de g, então qualquer processo correto que fornece m entregará m antes de m.
- Ordem total: Se um processo correto entrega a mensagem m antes de entregar m, então qualquer outro processo correto que entrega m entregará m antes de m.

7. Conclusão

Das pesquisas realizadas pode-se concluir que para que haja uma sincronização realmente efectiva há uma necessidade de se aplicar algoritmos certos para a resolução que visão de que se respeitem as questões de transparência, e as outras características de um sistema distribuído, dai que são comuns os algoritmos para a sincronização de relógio, de Exclusão mútua, para evitar casos de Deadlock e Algoritmos de eleição para a eleição do algoritmo que coordene a execução dos processos que o cliente inicia. Estes algoritmos são caracterizados por serem algoritmos de troca de mensagem, transportando na sua maioria os estados em que os processos se encontram durante a execução.

8. Referências Bibliográficas

- ✓ Barcelar (2013), Sincronização, http://www.ricardobarcelar.com.br/aulas/sd/7-sincronizacao.pdf visitado em 25/04/201 visitado em 24/04/2017.
- ✓ COULOURIS, George et al., Distributed Systems: Concept and Design, Pearson, 5th Edition, 2012
- ✓ Massetto (2013). Sincronização em sistemas Distribuídos. https://docente.ifrn.edu.br/diegopereira/disciplinas/2013/sistemas-distribuidos-sistemas-para-internet/material-extra/material-professor-isidro-sincronizacao visitado em 25/04/2017.
- ✓ http://www.inf.pucrs.br/~fldotti/sod/
- ✓ Tenanbau, A.; Steen, M. *Sistemas Distribuídos: princípios e paradigmas.* 2 edição. São Paulo: Prentice Hall, 200.
- ✓ LYNCH, Nancy A., PATT-SHAMIR, Boaz, Distributed Algorithm: Lecture Notes for 6.852, 1992