

Parte escrita - Teste 2

Relógios Lógicos vs Relógios Vetoriais:

Relógios Lógicos:

- cada processo P_i mantém um contador C_i ;
- para cada novo evento C_i é incrementado em 1
- mensagens enviadas por P_i levam $ts(m) = C_i$;
- sempre que P_j recebe uma mensagem atualiza C_j para $\max(C_j, ts(m))$ e incrementa C_j em 1

Esta abordagem tem problemas de consistência. Sendo a consistência causal uma delas. Com relógios lógicos não conseguimos garantir que se $a \rightarrow b$ (a antecede b) então $C_i(a) < C_i(b)$. Por outro lado, são bastante fáceis de implementar e não envolvem muita sobre carga da rede.

Relógios Vetoriais:

- cada processo P_i mantém um contador (vetor) VC_i de n dimensões, sendo n o número de processos envolvidos
- para cada novo evento o processo P_i incrementa $VC_i(i)$ em 1
- mensagens enviadas por P_i levam $ts(m) = VC_i$;
- sempre que P_j recebe uma mensagem atualiza $VC_j(K) = \max(VC_j(K), ts(m)[K])$ para todo o K e incrementa $VC_j(j)$ em 1

Para além disso, para que a consistência causal seja garantida é necessário garantir que todos as mensagens com causa envolvida tenham sido entregues a todos os processos. Isto faz com que eventos causais em concorrência não sejam conflitantes (sendo o receptor responsável por detectar estes e só efetuar a atualização quando da receção de todos os mensagens com contador inferior). Estes requisitos tornam esta abordagem extremamente lenta e como os pacotes são maiores relativamente aos relógios lógicos isso implica uma maior sobre carga da rede. Por outro lado, conseguimos garantir consistência causal!

Algoritmos de Exclusão Mútua:

Exclusão mútua centralizada:

- Temos N processos onde apenas 1 é o coordenador e é responsável por gerir uma fila de espera de acesso ao recurso
- Este coordenador é contactado com mensagens de REQUEST e responde (quando for possível aceder ao recurso) com mensagens OK.
- Este protocolo é bastante simples, mas por outro lado é centralizado. Isto implica um único ponto de falha, quando a falha do coordenador a fila pode ser perdida ^{involvendo um processo de eleição} e irrecuperável. Para além disso, processos que falharem dentro da exclusão mútua são um grande problema implicando mensagens de verificação de crash. Assim, tornando esta abordagem de baixa escalabilidade (para além da possível sobre carga do coordenador)

Exclusão Mútua Descentralizada:

- Temos N processos onde um fator (normalmente fixo) dos processos são coordenadores. O número de coordenadores tem de ser ímpar para garantir que existe sempre uma maioria de votos.
- Para além disso, o processo é igual mas o processo que deseja aceder ao recurso contacta todos os coordenadores, que entre eles vão tomar uma decisão (pela maioria)
- Algoritmo fácil com vários pontos de falha, envolve a eleição de novos coordenadores e o consenso entre eles, podendo tornar esta abordagem ineficiente (com muitas mensagens). Podendo chegar a situações de fome por parte de quem quer aceder ao recurso. Para além disso, também tem de lidar com falhas de processos dentro da exclusão mútua.

Exclusão Mútua Distribuída:

- Nesta abordagem estornos na presença de um flat group, onde não existem coordenadores e a decisão é feita por todos os processos
- Cada processo que pretende aceder ao recurso envia um REQUEST em multicast contendo o ts(m) igual ao seu relógio interno e esperando por um OK vindo dos outros processos.
- No caso de concorrência o processo com menor ts(m) irá aceder ao recurso e irá guardar numa fila interna o próximo nó que está à espera de aceder ao recurso
- A falha de qualquer processo é difícil de tolerar e existem muitos pacotes a circular (no limite de mensagens, sendo N REQUEST e $N-1$ OK)

Token Ring:

- Esta abordagem é baseada em anel onde é circulado um token.
- Sempre que um processo tem o token pode aceder ao recurso
- Esta abordagem bastante simples traz enormes problemas, como por exemplo:
 - falha do processo que tem o token: token perdido, token duplicado (no caso de se tentar recuperar da falha com o mesmo token e o processo que falhou voltar)
 - crash de qualquer processo pode "quebrar" o anel: implica que todos conhecem todos
 - para redes maiores o token pode demorar muito tempo a chegar: situações de forme

Algoritmos de Eleição:

Eleição por Bullying:

- No algoritmo de eleição por Bullying consideremos N processos P_0, \dots, P_{N-1} onde cada processo P_k tem um $id(P_k) = k$.
- Quando um processo P_k percebe que o coordenador falhou, por exemplo porque ultrapassou um timeout, ele começa o algoritmo, que envolve as seguintes etapas:
 - P_k envia uma mensagem ELECTION para todos os processos na rede com id superior a k (incluído o coordenador que deve ser o processo com o maior id), P_k, \dots, P_{N-1} .
 - Se ninguém responder dentro de um determinado tempo P_k torna-se coordenador (líder) e envia em multicast uma mensagem COORDINATOR a dizer que passou a ser o coordenador.
 - sempre que um processo recebe uma mensagem ELECTION ele responde OK e o processo que começou a eleição aberta. Este processo começa a eleição no caso de ainda não ter começado. Assim, o único processo que vai ser elegido é o com maior id, pois não irá receber confirmação (OK) de processos com id superior.
- Este algoritmo envolve bastantes rondas, envolvendo no limite $O(N^2)$ mensagens. Para além disso, se houver falhas durante o processo de eleição podem ser elegidos vários líderes o que torna a abordagem pouco escalável pois precisamos de muito controlo a partição.

Eleição num Anel:

→ Na eleição em anel, cuja rede pode não estar organizada em anel, consideramos N processos $\{P_0, \dots, P_{N-1}\}$ com os respectivos $id(k) = k$. A semelhança com a eleição por Bullying, quando um processo percebe que o coordenador falhou, ele segue vários etapas:

- faz circular pelo anel uma mensagem ELECTION que não contém uma lista ordenada (por ordem de inserção) de id's por onde passou a mensagem. Para isso, envia a mensagem com o seu id na lista para o seu sucessor mais próximo que esteja vivo
- cada processo que recebe a mensagem ELECTION, coloca o seu id na lista e reenvia-a para o seu sucessor (vivo)
- quando a mensagem circular a rede toda (processo repete que o seu id já está na lista) ele assume como coordenador o que tiver o maior id e faz circular uma mensagem COORDINATOR com a respetiva lista, que irá propagar o coordenador

→ Este algoritmo bastante simples trás problemas de escalabilidade, pois para redes maiores circular a rede toda é impraticável. Para além disso, com a lista nas mensagens todos sabem todos mas as mensagens podem ficar intransponíveis.

Eleição em sistemas de grande-escala:

→ Algoritmos de eleição por bullying e eleição num anel são extremamente inefficientes quando falamos de sistemas de grande-escala. Em sistemas de grande-escala, com eleição de um ou mais coordenadores, a prioridade de decisão dos coordenadores tem de ter critérios mais realistas, como por exemplo a latência com processos normais, a cobertura, o processamento, a importância (processo que gera autenticação), ... Um exemplo concreto de eleição em grande-escala acontece em redes P2P na eleição de super-nós, onde (normalmente) seguem os seguintes etapas:

- cálculo contínuo da latência com outros super-nós e nós normais
- privilegiar alta latência entre super-nós e baixa latência com nós normais (^{distribuição} uniforme)
- definir cobertura máxima de um super-nó, sendo preciso mais super-nós em zonas mais sobrecongregadas

Para além disso, também é comum usarem a sobrecarga da CPU como fonte de decisão na eleição de super-nós.

Problemas na replicação:

→ Replicar informação é um processo muito comum em sistemas distribuídos envolvendo criar e manter réplicas de um sistema de informação.

→ Porquê replica?

- Aumenta a fiabilidade do Sistema:

- maior tolerância a falhas

- tolerância a eventuais corrupções de dados

- Eficiência:

- distribuição geográfica para cobrir árees distantes

- evita sobre carga de um servidor

→ Por outro lado, replicar implica viver com uma série de problemas, como por exemplo, a inconsistência de dados. É o famoso "trade-off" do Teorema CAP, onde qualquer sistema de informação pode ter no máximo 2 dos 3 seguintes propriedades:

Consistência, Disponibilidade e Tolerância a Partições.

→ Existem numerosas formas de combater a inconsistência e uma delas é perceber bem os nossos requisitos e ver se podemos tornar as operações imediatamente consistentes, e implementar algoritmos de consistência baseado nos nossos requisitos, com por exemplo:

- consenso por flooding

- consenso com o Algoritmo Paxos

- através de push e pulling updates

- leases (alternando dinamicamente os push e pull updates através de um contrato)

- (...)

Modelos de consistência Data-Centric:

→ Um (data-centric) modelo de consistência é um contrato entre uma data-store (distribuída) e processos, em que a data-store especifica concretamente qual o resultado de operações de leitura e escrita em concorrência.

→ Três tipos de consistência data-centric são:

Consistência contínua:

→ Consiste em definir limites para o menor grau de consistência (a unidade de dados que define a consistência é o Comit), entre eles estão:

- limites de diferenças de valores numéricos
- limite de tempo desde a última atualização
- limite de operações em atraso

Consistência Sequencial:

→ Neste nível de consistência não nos interessa qual a ordem pela qual os operações são executadas, queremos apenas garantir que a ordem é igual em TODOS os processos, ou seja na mesma ordem sequencial

"O resultado de qualquer execução é o mesmo se todas as operações em todos os processos forem executadas na mesma ordem sequencial, e as operações de cada processo aparecem na mesma ordem especificada no programa"

Consistência Causal:

→ Neste nível de consistência queremos garantir que, se $a \rightarrow b$ (a antecede b) e existe $w_x(a)$ e $w_x(b)$ então obrigatoriamente todos os processos tem "a" primeiro que "b" estrutura de dados w_x .

"Escritos que estão relacionados causalmente têm de ser vistos por todos os processos com a mesma ordem e pela ordem correta. Escritos concorrentes podem ser vistos por ordens diferentes em diferentes processos"

Consistência Eventual:

→ Neste nível de consistência, o objetivo é manter a consistência entre eventos, ou seja, que o cliente tenha uma experiência consistente. (esta consistência também pode estar enquadradada em modelos de consistência Client-centric)

→ Assim, temos de garantir uma coisa:

- todos os atualizações chegam a TODAS os réplicas (não é necessária a rápida atualização)

Modelos de consistência Client-Centric:

Leituras monotônicas:

→ Se um processo ler um valor x , qualquer operação de leitura sobre x pelo mesmo processo retorna x ou um valor mais recente

Escriftas monotônicas:

→ Uma operação de escrita por um processo num dado x é completado antes de qualquer operação de escrita em x pelo mesmo processo, não existindo escritos concorrentes.

Leitura após escrita: (read-your-writes)

→ O efeito da operação de escrita por um processo nos dados x , será sempre visto pelos operações de leituras sucessivas em x pelo mesmo processo.

Escrifta após leitura: (write-follows-reads)

→ Numa operação de escrita por um processo sobre os dados x após uma operação de leitura prévia sobre x pelo mesmo processo, é garantido que ocorre sobre o mesmo valor de x que foi lido ou um mais recente

Como se define o grau de confiabilidade de um SI:

→ A confiabilidade de um sistema de informação (normalmente) é definida analisando as seguintes propriedades:

- Disponibilidade (prontidão para ser usado)
- Fiabilidade (disponibilidade contínua do serviço)
- Segurança (baixa probabilidade de catástrofe)
- Manutenibilidade (quão fácil é reparar o sistema)

Três tipos de redundância para moscoror falhos:

→ redundância na informação (acrescentar bits nas mensagens para poder recuperar mensagens no caso de estarem danificadas)

→ redundância temporal (construir o sistema para que uma falha temporal seja tolerada, isto é, reenviar os mensagens no caso de não ter resposta)

→ redundância física (adicionar mais componentes ao sistema, por exemplo, mais processos / backups)

Distribuição de conteúdos Cliente - Servidor:

→ Normalmente, é feito através de 2 tipos de updates:

- push updates (iniciado pelo servidor e é propagado independentemente do cliente ter pedido ou não, normalmente utilizado para guardar cache no cliente)
- pulling updates (iniciado pelo cliente, onde o cliente solicita atualizações ao servidor)

→ Para além disso, é possível comutação dinamicamente entre os dois modos através de "leases":

um contrato entre o cliente e o servidor, onde o servidor promete atualizar (fazer push update) até ao contrato terminar, podendo o cliente renovar fazendo um pull update. O tempo entre updates pode também ser definido dinamicamente:

- baseado na idade, é comum um processo que não é alterado à muito tempo, não vir a sofrer alterações nos próximos tempos (tempo maior)
- baseado na frequência, para evitar overheads na comunicação e sobre carga um cliente que pede muitas atualizações deve receber com maior frequência
- baseado no estado de corja, se o servidor estiver muito sobre carregado os períodos de leases devem ser menores

Consenso por Flooding:

→ O algoritmo de consenso por flooding permite obter consenso e moscas eventuais falhas de processos durante a sua execução (não é para falhas arbitrárias).

→ Para isso, consideramos N processos P_0, \dots, P_{N-1} e cada processo tem a lista dos processos propostos. Baseado em rondas, periodicamente o algoritmo segue estes etapas:

- na ronda r , o processo P_j envia o seu conjunto de comandos Cmd_j^{r+1} para TODOS os outros, mensagem RESULT
 - no final da ronda r , um processo P_k que recebeu de TODOS os processos finde todos os comandos em Cmd_k^{r+1} e envia para TODOS a confirmação, mensagem CONFIRMATION. Esta função é obrigatoriamente um processo determinista conhecido por todos.
 - quando um processo que não recebeu RESULT de todos recebe um CONFIRMATION ele atualiza a sua lista de comandos.
- O problema deste algoritmo é que todos têm de conhecer todos e funciona em multicast, ou seja, para redes maiores não é viável. Para além disso, mesmo que um processo P_i não tenha mudado a sua lista de comandos tem de enviar periodicamente o RESULT.

Consenso Realista - Paxos :

→ O Algoritmo Paxos é um algoritmo de consenso realista criado por Leslie Lamport e que provou, assumindo $2K+1$ processos e que mesmo com falhas por crash a rede chega a consenso.

→ Assumindo algumas coisas:

- sistema é parcialmente síncrono (podendo ser assíncrono)
- comunicação pode ser instável
- TODAS as operações são determinísticas
- processos podem assumir falhas por crash, mas não arbitrários
- processos não podem conspirar

→ Assumindo três papéis principais, que podem acontecer no mesmo processo:

- Proposer (quem propõem o valor para chegar ao consenso)
- Accepter (quem contribui para chegar a um consenso)
- Learners (apresentam apenas o valor final, podem ser questionados, por exemplo, por clientes)

→ O algoritmo segue os seguintes etapas:

- Proposer começa o consenso enviando uma mensagem PREPARE para todos os Accepters incluindo o id (que tem de ser único e maior do que o anterior).

- Os Accepters recebendo o PREPARE, id respondem (no caso de não terem prometido o contrário a outro processo) com PROMISE e com um valor se tiverem algum valor de consenso já acordado. Este promise reflete que este Acceptor não irá responder mais a pedidos com id inferior a este.
 - O Proposer por outro lado, ao receber uma maioria de PROMISE por parte de Accepters começa então a fase troca de valores enviando uma mensagem ACCEPT, id, valor sendo "valor" o valor que pretende que seja acordado
 - Por outro lado os Accepters recebendo o ACCEPT e guardando o valor acordado enviam a confirmação com LEARN e o valor, tanto para o Proposer quanto para os Listeners
 - Quando os Listeners e o Proposer receberem a maioria dos LEARN guardam o valor acordado e o consenso acaba
- Depois disso, outra ronda com um id superior poderá começar!

Consenso com Falha arbitrária:

- Para termos consenso entre nós mesmos quando existe uma falha arbitrária temos de ter $3K+1$ processos, para que, mesmo que a mensagem seja incorreta exista SEMPRE uma maioria para chegar ao consenso
- Para realizar este algoritmo temos de ter um primário P e backups $\{B_0, \dots, B_{3K-1}\}$ e o algoritmo segue os seguintes etapas:
 - O algoritmo começa com P a enviar o valor que pretende acordar para todos os backups (o primário pode também ter falhas arbitrárias, sendo a decisão final tomada de acordo com a maioria de resposta dos backups)
 - Por outro lado, os backups enviam esses valores entre si em multicast para que com maioria possam chegar a um consenso
- Tanto o primário como os backups podem falhar pois temos sempre a maioria.
- Embora se chegue a consenso é um protocolo muito impreciso

O que é necessário para chegar a consenso:

→ Sistema de informação tem de ser:

- Síncrono sem mensagens ordenadas (é síncrono por isso tem um relógio implícito) e com tempo de resposta limitado
- Síncrono com mensagens ordenadas, tanto com tempo de resposta limitado ou não
- Assíncrono com mensagens não ordenadas mas com multicast (ordenação implícita)

Problemas de RPC's fiáveis

→ O que pode acontecer?

- ① - o cliente não encontra o servidor
- ② - a mensagem do pedido do cliente para o servidor é perdida
- ③ - o servidor recebe a mensagem e crasha
- ④ - o servidor recebe a mensagem, executa e crasha (ou a mensagem é perdida)
- ⑤ - o cliente crasha e o servidor responde

→ Soluções:

- ① - reportar para o cliente
- ② - fazer uma verificação com o servidor se o pedido chegou ou não e voltar a tentar
- ③ e ④ - não existe uma única solução, temos duas opções:
 - "at-least-one": a operação deve ser executada em pelo menos um servidor, assim podemos reenviar sem problema até um servidor responder
 - "at-most-one": a operação só pode ser executada no máximo uma vez, se não tivermos confirmação não voltamos a tentar

Vai depender do modelo de negócio!

- ⑤ - não tem grande impacto mas o servidor pode verificar com timeouts se o cliente continua vivo, ou limita o pedido a T unidades de tempo

Protocolos de commit distribuído:

Multicast Fidél:

- Onde a mensagem é entregue a todos os receptores e para tolerar falhas existe um relógio e uma eventual retransmissão
- Nesta abordagem, todos conhecem todos e quando alguém entra o processo pode ser complicado com um eventual lock do grupo

Transmissão distribuída : (2PC)

- Normalmente implementado através do protocolo 2PC, onde cada transação local tem de ter sucesso. Funciona seguindo as seguintes etapas: (parecido ao Paxos)
 - Processo que inicia a computação é designado de coordenador e os restantes não os participantes. Este algoritmo, como o nome indica, envolve duas fases:
 - 1.ª Fase: coordenador envia VOTE_REQUEST para todos os participantes que não responderem com VOTE_COMMIT ou VOTE_ABORT. Se o coordenador receber um VOTE_ABORT ele aborta a computação e envia para todos GLOBAL_ABORT.
 - 2.ª Fase: Caso o coordenador receba VOTE_COMMIT de todos os processos ele envia finalmente o GLOBAL_COMMIT com o valor pretendido a os participantes quando receberem executam a computação
- Normalmente, o 2PC é implementado com base em 4 estados possíveis INIT, READY, COMMIT e Abort.
- Para além disso é comum utilizarem workspaces temporários para recuperar o estado no caso de falhas, assim como timeouts para o coordenador não bloquear com falhas dos participantes

Como recuperar de falhas :

Forward Error Recovery:

- Avançamos para a frente até encontrarmos um estado em que toda a rede concorde e possamos continuar

Backward Error Recovery :

- Recuarmos até uma versão que seja acordada e que nos permita continuar. (recovery points)

→ Pode ser feito das seguintes formas:

- checkpoints coordenados (à medida que vamos avançando vamos guardando estados que consideremos seguros, em conjunto)
- checkpoints individuais (à medida que vamos avançando cada processo individualmente guarda estados, para que quando precisarmos de recuperação os processos possam chegar a um acordo de estado e não voltar tudo para o início)
- logging (guarda a lista de comandos que foram executados até ao momento, para que, quando falharmos possamos recuperação todo o que fizemos até ao momento)