Sistemas Distribuídos

Resumo

Conteúdo

Capítulo 1

Objetivos e Tipos de Sistemas Distribuídos

1.1 Objetivos dos Sistemas Distribuídos

1.1.1 Suporte a Partilha de Recursos

Fornecer acesso a recursos remotos é o principal objetivo de um sistema distribuído.

1.1.2 Transparência de Distribuição

A transparência passa por simular o comportamento de um sistema não distribuído, ocultando falhas, concorrência, replicação, etc.
É uma característica desejável, mas que por vezes não faz sentido. Por exemplo, não é possível esconder a latência de comunicação na internet, ou não seria boa ideia imprimir um documento numa impressora escolhida pelo sistema, quando seria melhor ser o utilizador a escolher. Por vezes é melhor deixar claro que estamos num sistema distribuído e dar mais controlo ao utilizador.

1.1.3 Abertura

Um sistema é aberto se tem regras e interfaces bem definidas. Para se atingir este objetivo temos de investir na definição de interfaces padronizadas, como acordos no uso de formatos e tipos de dados. Alguns conceitos relacionados são:

- Interoperabilidade
- Portabilidade
- Extensibilidade

1.1.4 Escalabilidade

Um sistema distribuído deve ser descentralizado:

- Nenhuma máquina tem informação completa sobre o sistema

- As decisões são tomadas com a informação local
- Uma falha não impede o funcionamento do algoritmo
- Não se usa a hipótese da existência de relógios sincronizados

Para uma melhor escalabilidade deve-se esconder a latência, através de comunicação assíncrona, deve-se particionar e distribuir, e usar replicação.

1.2 Tipos de Sistemas Distribuídos

1.2.1 Sistemas de Computação Distribuída

Clusters

Um cluster é um conjunto de máquinas ligadas numa rede comum que trabalham em conjunto, geralmente usado para processamento paralelo num ambiente homogéneo (todas as máquinas são iguais e correm o mesmo software).

Nos clusters é frequente o uso de virtualização, para uma maior facilidade de migração (portabilidade).

Grids

Por oposição aos clusters, as grids são usadas para computação em grande escala e apresentam um alto grau de heterogeneidade, em que cada nó pode ter um SO, hardware, rede diferentes.

O middleware trata a homogeneidade e dá acesso a recursos a utilizadores da sua organização.

Computação na Nuvem

Na computação na nuvem um utilizador envia sua computação para um serviço e é cobrado pelo uso de recursos. A cloud caracteriza-se por oferecer acesso a um conjunto de recursos virtualizados.

1.2.2 Sistemas de Informação Distribuídos

Sistemas de Processamento de Transações

As transações têm como objetivo proteger recursos contra acessos simultâneos de processos e permitir que um processo faça operações como se essas fossem uma operação atómica. As propriedades ACID das transações são:

- Atomicidade: para um observador externo, uma transação executa na sua totalidade ou não executa, i.e., acontece indivisivelmente
- Consistência: cada transação leva o sistema de um estado válido para um novo estado válido
- Seriabilidade: se diversas transações forem executadas em paralelo sobre os mesmos recursos, o resultado é equivalente à execução dessas transações uma após a outra

- Persistência: os resultados de uma transação que fez commit (END_TRANSACTION) permanecem depois desta acabar

1.2.3 Sistemas Pervasivos Distribuídos

Este tipo de sistema é caracterizado pelo facto dos dispositivos pertencerem a um ambiente, em que a separação entre sistema, utilizador e ambiente não é muito clara.

- Sistemas ubíquos (omnipresentes)
- Sistemas móveis
- Redes de sensores

Capítulo 2

Arquiteturas

2.1 Estilos Arquiteturais

2.1.1 Camadas

Os componentes são organizados em camadas, em que os pedidos geralmente descem e as respostas sobem. Um componente da camada L_i só pode chamar componentes da camada L_{i-1}. O modelo em camadas pode ser usado na organização do software de um nó ou na organização de um sistema distribuído.

2.1.2 Objetos

Cada objeto encapsula um estado e suporta uma série de métodos que podem ser invocadas por outros objetos.

2.1.3 REST

Nas arquiteturas REST, os componentes são organizados como recursos, cada um com nome único mas interface idêntica. As 4 operações suportadas são:

- PUT
- GET
- DELETE
- POST

2.1.4 Eventos

Os processos comunicam através da propagação de eventos que podem, ou não, conter dados. Para conseguir desacoplamento entre processos, a comunicação deve ser feita com ajuda de um middleware, que pode ser um barramento de eventos e/ou um espaço partilhado de dados, usando o modelo publish/subscribe.

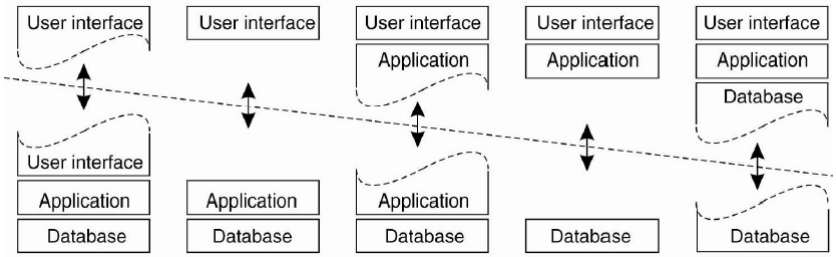
2.2 Arquiteturas de Sistemas

2.2.1 Arquiteturas Centralizadas

Numa arquitetura centralizada tem-se uma divisão de papéis entre os processos que oferecem e os que usam serviços.

Arquiteturas Cliente-Servidor

Para além dos servidores, distinguem-se dois tipos de clientes, os clientes magros, com menor escalabilidade, pior desempenho e maior facilidade de gestão, e os clientes gordos, por oposição.



2.2.2 Arquiteturas Descentralizadas

É possível dividir aplicações cliente-servidor em camadas (interface, processamento e dados), segundo uma distribuição vertical, em que cada máquina tem um conjunto de obrigações diferentes, ou horizontal, em que cada máquina tem todo o conjunto de funcionalidades e só uma parte dos dados.

Existe ainda o conceito de rede overlay, que representa as ligações lógicas entre dois nós do sistema distribuído, podendo seguir uma abordagem estruturada, com estrutura definida, ou não estruturada, correspondendo a um grafo aleatório.

Peer to Peer

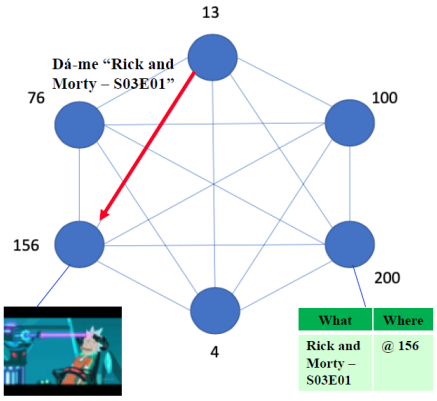
Estruturada Dado um conjunto de processos contendo diferentes conjuntos de recursos, encontra os processos que contêm um certo recurso através do seu id único, permitindo resolver eficientemente o problema da localização de recursos. Este tipo de rede usa um índice independente da semântica da aplicação onde cada recurso é associado a uma chave e é útil em aplicações de partilha de ficheiros. Segue-se um exemplo de uma rede DHT:

Ex: O nó 156 quer partilhar o ficheiro "Rick and Morty – S03E01"

- Calcula o seu hash e dá o valor 201
- Diz ao nó com id mais próximo, o 200, que ele tem esse ficheiro
- O 200 guarda essa informação na sua tabela

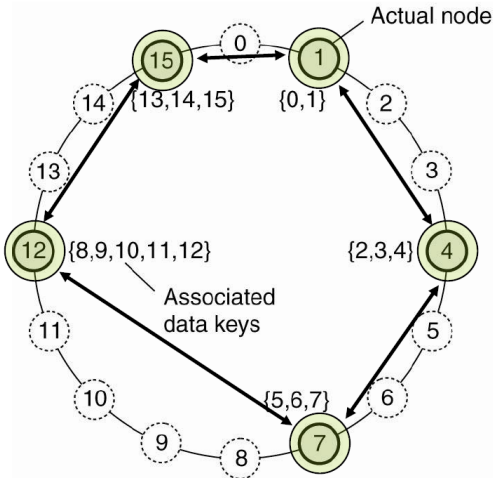
A seguir o 13 quer esse episódio. Ele calcula o seu hash usando a mesma função e obtém também 201

- Então ele pergunta ao nó mais próximo, o 200, sobre quem tem o ficheiro
- O 200 consulta a sua tabela e responde que é o 156
- E o 13 pede o ficheiro ao 156



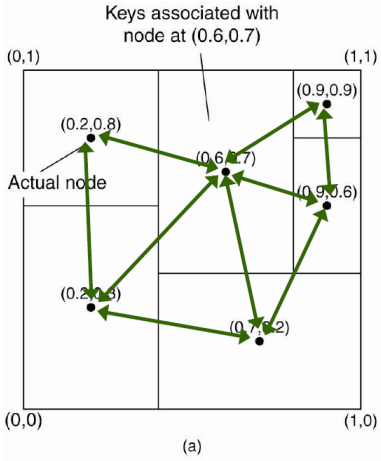
Um exemplo de rede: ***Chord***

- Rede ***overlay* em anel** com **vista parcial** de nós
- Cada nó conhece apenas o **próximo nó** e o **anterior** no anel (e mais uns atalhos, como veremos a seguir)
- Espaço de chaves e *ids* com ***m* bits**, contém no **máximo 2^m nós**
- *id* do **nó responsável** por uma **chave *k*** é dado por ***succ(k)***
- *succ(k):* **nó** com **menor *id*** que seja **maior ou igual** à **chave *k***



Content Addressable Network (CAN)

- Espaço cartesiano de *d* dimensões
  (ao lado [0,1]x[0,1])
- Cada nó é responsável por uma região desse espaço
  (ao lado, temos 6 nós)
- Cada item de dados está associado a um ponto nesse espaço
  - » Requer uma "função de *hash*" um bocado mais sofisticada
- Cada nó da rede tem ligações com nós responsáveis por regiões subjacentes a sua



Não Estruturada A topologia de uma rede não estruturada é tipicamente um grafo aleatório. Numa overlay deste tipo, a pesquisa exata é complicada, mas é mais fácil lidar com falhas, sendo mais adequado para pesquisas por proximidade. As buscas são feitas por:

Flooding: difusão da busca a todos os vizinhos

Random Walk: escolhe um vizinho e pergunta, que segue em frente

Gossip: cada nó retransmite apenas a alguns nós, escolhidos de acordo com alguma estratégia

Para recuperar de falhas e entradas/saídas de nós basta alterar a rede temporariamente.

2.2.3 Arquiteturas Híbridas

Numa arquitetura híbrida existe uma rede overlay com vários nós e alguns servidores que desempenham papéis bem definidos.

Um exemplo é o BitTorrent, onde cada nó contem um ficheiro de outros nós, de forma descentralizada, enquanto que o tracker é um servidor descentralizado que monitoriza que nós ativos têm esses blocos.

Capítulo 3

Comunicação

3.1 Protocolos de Middleware

Frequentemente as aplicações não comunicam diretamente usando protocolos de transporte, fazendo uso de protocolos de middleware de alto nível, que podem suportar serviços como transações, autenticação, autorização e sincronização.

3.1.1 Tipos de Comunicação

A comunicação por protocolos de middleware pode ser classificada em dois parâmetros:

- Persistência
  - Transitória: a mensagem é armazenada apenas enquanto o emissor e o recetor estão ativos
  - Persistente: a mensagem é armazenada até ser entregue ao recetor
- Sincronização na comunicação
  - Síncrona: cliente bloqueia à espera da resposta do servidor
  - Assíncrona: cliente não bloqueia e recebe notificação quando a resposta está disponível

3.2 Chamadas a Procedimentos Remotos (RPC)

No caso geral, a chamada a procedimentos permite a transferência de controlo e dados dentro de um programa. A chamada a procedimentos remotos é uma extensão deste modelo para o caso dos sistemas distribuídos. Numa chamada a um procedimento:

- Os argumentos (por valor, referência ou cópia/reposição) são colocados na pilha pela ordem inversa
- O procedimento é executado
- O endereço de retorno é usado para devolver o controlo ao procedimento que fez a chamada

3.2.1 Empacotamento de Parâmetros

É fundamental que o cliente e o servidor usem o mesmo formato de mensagem de pedido e resposta.

Passagem Por Valor

São enviadas cópias dos parâmetros ao servidor. É necessário prestar atenção ao problema da representação dos dados: interpretação e conversão dos bytes recebidos no servidor. Para tal, é importante haver acordo sobre os tamanhos dos tipos básicos.

Passagem Por Referência

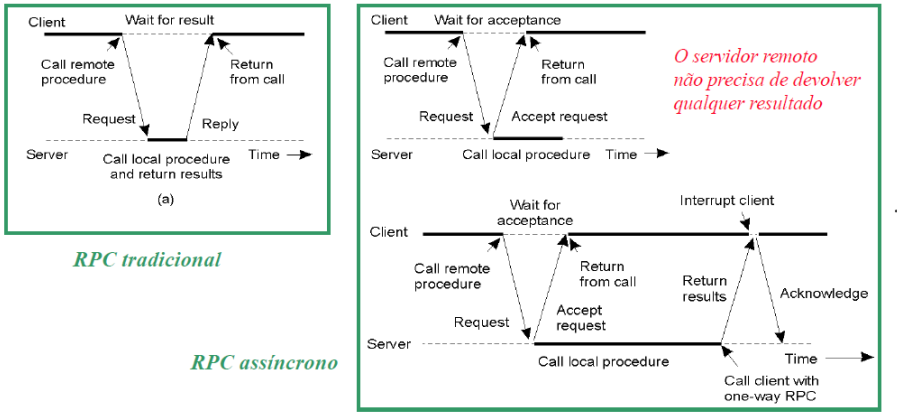
A passagem é feita por ponteiros para zonas de memória. Surge o problema dos endereços das estruturas de dados no cliente não terem significado válido no servidor. A solução passa por transformar os parâmetros por referência em cópia/reposição, que são alterados no servidor e depois retornados de volta ao cliente, juntamente com a resposta. Por exemplo, para tipos básicos passa-se o dado referenciado em vez do ponteiro (int em vez $int*$), ou nos vetores, declara-se explicitamente o seu tamanho ($int[10]$ para um vetor de 10 inteiros).

3.2.2 Localização do Servidor

Pode-se codificar o endereço do servidor diretamente no cliente, mas é uma solução pouco flexível e com pouca transparência. A solução passa por uma associação dinâmica através de um serviço de nomes, que tem como desvantagem ser um ponto central de falha.

3.2.3 RPC Assíncrono

Os RPC assíncronos apresentam melhor desempenho, já que evitam o bloqueio do cliente quando o pedido está a ser processado.



Este padrão é usado para a chamada de procedimentos em vários servidores concorrentemente e essencial para a concretização da replicação.

3.3 Comunicação por Mensagens

3.3.1 Comunicação Transitória por Mensagens

Os interlocutores precisam de estar ativos ao mesmo tempo. Existem os Berkley Sockets, centrados no conceito de socket, usada para ler e escrever mensagem e as Message-Passing Interfaces (MPI), que fornecem uma API mais elaborada para comunicação, sem preocupação com falhas e sincronização.

3.3.2 Comunicação Persistente por Mensagens

A comunicação persistente concretiza-se através de filas de mensagens, que removem a necessidade do interlocutor estar em funcionamento no momento da comunicação e o bloqueio do cliente enquanto o pedido é processado.

Faz sentido usar este tipo de comunicação quando os interlocutores estão ligados através de redes de grande escala, para as quais a probabilidade de existirem desconexões não é desprezável.

3.3.3 Modelo das Filas de Mensagens

Existe um Message-Oriented Middleware, que fornece suporte a comunicação persistente e assíncrona, num espaço de armazenamento dentro do sistema. Fornece a seguinte interface:

PUT	Colocar uma mensagem numa fila
GET	Retirar a primeira mensagem e bloquear até que a fila não esteja vazia
POLL	Verificar se a fila tem uma mensagem e retirá-la nesse caso (ou retornar sem bloquear)
NOTIFY	Definir uma função que será chamada sempre que uma mensagem for colocada na fila

Arquitetura para um Sistema de Filas de Mensagens

- As mensagens são enviadas para uma fila (com identificador único)
- O endereço da fila é mapeado num endereço do nível transporte (IP/porto) para que a mensagem possa ser enviada para o destino.
- A mensagem depois é entregue ou encaminhada por um conjunto de servidores (ou relays) até ser copiada para a fila no recetor
- Em várias localizações da rede existem buffers que permitem o armazenamento temporário das mensagens

3.4 Comunicação por Streams

As streams são geralmente usadas para transmitir áudio e vídeo através da rede, pois dão maior importância ao instante em que os dados são recebidos.

3.4.1 Qualidade de Serviço (QoS)

Existe um acordo com o sistema de suporte da rede em que se estabelece uma especificação da qualidade de serviço. Podem ser definidos os seguintes parâmetros:

- Máximo de perdas
- Taxa de transmissão
- Jitter máximo
- ...

Ao criar uma Stream, a especificação da QoS tem de ser mapeada num conjunto de recursos a reservar na rede de forma a que os requisitos indicados possam ser garantidos.

Para manter a QoS são usados vários mecanismos, tais como buffers de receção, que evitam furos e atrasos na apresentação de conteúdos.

3.4.2 Sincronização entre Streams

Por vezes é necessária sincronização entre diversas streams, tais como vídeo e legendas, ou áudio e vídeo.

Esta sincronização pode ser feita por um middleware ou pelo SO, através da aplicação.

3.5 Resumo

- Comunicação persistente: cada mensagem que enviada é mantida pelo sistema o tempo necessário para que o recetor a vá ler
  - após o envio da mensagem o emissor pode terminar imediatamente
  - o recetor não precisa de estar em execução quando ocorre a emissão
- Comunicação transitória: as mensagens são apenas armazenadas pelo sistema enquanto o emissor e recetor se encontram em execução
  - a mensagem é descartada se recetor não estiver em execução no momento do envio
- Comunicação assíncrona: o emissor continua a sua execução imediatamente após a submissão da mensagem
  - isto sucede mal a mensagem tenha sido copiada para o tampão local ou do primeiro servidor
- Comunicação síncrona: o emissor fica bloqueado até que:
  - a mensagem seja copiada para o tampão da máquina do recetor,
  - ou que o recetor a leia/receba,
  - ou que o recetor a tenha processado

3.6 Comunicação em Grupo

Um grupo é um conjunto de processos que atuam de forma coordenada, com a principal característica que todas as mensagens enviadas para o grupo são recebidas por todos os seus membros.

3.6.1 Tipos de Grupos

Grupos podem ser classificados quando á sua abertura:

- Grupo fechado
  - apenas os membros do grupo podem enviar mensagens para o grupo
  - processos exteriores podem enviar mensagens para membros específicos
- Grupo aberto
  - qualquer processo pode enviar mensagens para o grupo

Ou quanto á sua hierarquização:

- Grupo paritário
  - todos os membros do grupo são tratados de forma igual
  - as decisões são efetuadas com a cooperação de todos os membros
  - melhor tolerância a faltas mas gestão mais complexa
- Grupo hierárquico
  - existem membros diferenciados e existe hierarquia
  - muitas vezes existe um processo especial que coordena as tarefas
  - menor tolerância a faltas mas gestão mais simples

3.6.2 Fiabilidade na Entrega de Mensagens

Sem fiabilidade alguns membros do grupo podem não receber a mensagem. Com fiabilidade existem duas hipóteses:

Os membros do grupo não falham, não entram ou saem membros durante a execução do multicast e mensagens podem ser perdidas. Protocolo:

- O processo envia a mensagem para todo o grupo
- Sempre que um processo recebe uma mensagem verifica se já a recebeu; se sim descarta-a; caso contrário, envia a para todos os processos do grupo

Existem falhas, saídas e entradas durante o multicast:

- A principal questão que se levanta é determinar quais são os membros do grupo que devem receber a mensagem
- Em cada instante, o sistema determina quais são os membros do grupo, e sempre que existe uma alteração todos devem ser informados
- Quando uma mensagem é entregue, todos os membros nessa vista também entregam a mensagem

3.6.3 Ordem na Entrega de Mensagens

Existem várias forma de ordenar a entrega de mensagens a um grupo:

- Sem ordem: não existem restrições em relação à ordem da recepção das mensagens pelos membros do grupo
- Ordem FIFO: se um processo envia a mensagem m e depois a mensagem m' então nenhum processo entrega à aplicação a mensagem m' antes de ter entregue a mensagem m
- Ordem causal: se um processo envia a mensagem m que precede causalmente (happens before, $m \rightarrow m'$, dos relógios) uma mensagem m' então nenhum processo entrega à aplicação a mensagem m' antes de ter entregue a mensagem m
- Ordem total: se dois processos entregam à aplicação as mensagens m e m' então o primeiro processo entrega m antes de m' se e só se o segundo processo entrega m antes de m'

Difusão Fiável com Ordem Total

Por fim, coloca-se a hipótese de os membros dos grupos poderem falhar, sem entradas ou saídas. O protocolo é o seguinte:

- O processo envia a mensagem m para todo o grupo (com temporizadores e retransmissões)
- Sempre que um processo recebe uma mensagem verifica se já a recebeu; em caso afirmativo, descarta-a; caso contrário, envia-a para todos os processos do grupo (com temporizadores e retransmissões)
- Um membro (coordenador) define a ordem de entrega (número de sequência) i para m e envia a mensagem $< ORDER, m, i >$ a todos os processos (com temporizadores e retransmissões)
- Em caso de falha do coordenador, os membros executam um algoritmo de eleição de líder, escolhem um novo coordenador, e enviam lhe informações sobre as suas mensagens ordenadas e mensagens pendentes

Capítulo 4

Nomes

Num sistema distribuído, nomes denotam entidades. Têm um papel fundamental, pois a sua gestão permite identificar, localizar e partilhar recursos.

4.1 Serviços de Nomes

Um serviço de nomes tem como tarefa fundamental a gestão de associações entre um nome e outra designação que permite localizar o recurso ou a entidade responsável pela sua gestão.

Um serviço de nomes suporta operações de registo de associações ou resolução das mesmas.

Arquitetura dos Serviços de Nomes

Em sistemas distribuídos existe muitas vezes um serviço de nomes (SN), que satisfaz 3 propriedades:

- disponibilidade: a resolução de nomes tem de estar disponível
- desempenho: a resolução de nomes tem de ser eficiente
- escalabilidade: deve ser possível gerir um número elevado de nomes

Existem 3 soluções para a gestão dos nomes: replicar a informação de gestão em todas as máquinas, usar difusão na tradução do nomes (ARP) ou usar uma solução tipo cliente-servidor (DNS).

4.2 Agente (Cliente do Serviço de Nomes)

O agente pode ser uma biblioteca ligada aos processos ou um processo independente em cada máquina. Funciona da seguinte forma:

- Durante a inicialização tem de obter a localização do servidor. 3 opções:
  - o SN está associado a um endereço (IP, porto) bem conhecido
  - o SN difunde periodicamente o seu endereço

  - o agente faz um pedido em difusão
- Resolução de um nome, efetuada totalmente ou parcialmente pelo SN
- Se for parcialmente:
  - iterativo: um SN indica outro SN por onde a busca pode continuar
  - recursivo: um SN contacta outros SN até que consiga resolver o nome

4.3 Servidor de Nomes

Um servidor de nomes tem duas características que facilitam a sua construção: algumas inconsistências são toleráveis e os nomes mudam com pouca frequência, podendo-se usar mecanismos de caching e replicação. A resolução usa um algoritmo simples:

- se a associação nome-objeto existe localmente, responde-se ao agente
- caso contrário, o SN contacta o servidor responsável ou informa o cliente por onde a procura pode continuar
- para otimizar as resoluções remotas, o SN pode utilizar uma cache
- a disponibilidade pode ser aumentada através da replicação dos SN

4.3.1 Mobilidade

As entidades são móveis, e quando ocorre mobilidade existem duas soluções:

- Forwarding pointers: antes de se mover, o serviço deixa um apontador no seu ponto de acesso que leva ao novo ponto de acesso
- Abordagem home-based: existe um agente na rede onde o serviço foi criado que sabe encontrá-lo a qualquer momento

4.4 Tipos de Nomes

- Nomes simples (flat names): Sequencias de bits sem nenhuma semântica associada, usados em PIDs ou portos.
- Nomes hierárquicos: Nomes são organizados de forma a manter uma hierarquia de contextos, usados em IPs ou URLs.
- Nomes baseados em atributos: Além de nomes, usam-se outros pares (atributos/valor) para descrever um objeto e podem também manter estruturas hierárquicas. Usam-se no MS Active Directory e LDAP.

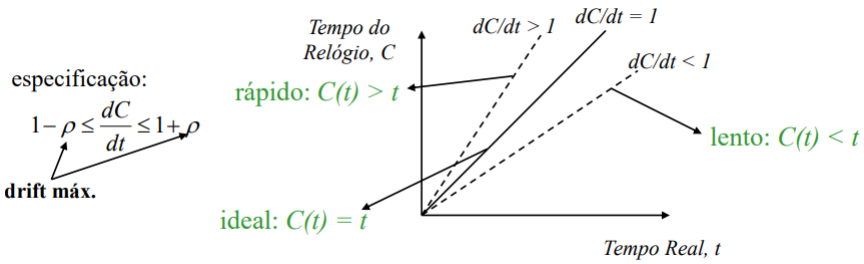
Capítulo 5

Sincronização e Relógios

5.1 Relógios Físicos

Os relógios de um computador são normalmente baseados num cristal de quartzo submetido a uma dada tensão elétrica, oscilando com uma dada frequência. Existe um contador que é decrementado a cada oscilação e, ao chegar a 0, o valor do relógio é incrementado.

Pretende-se então que os processos cheguem a um acordo em relação ao tempo e que esse seja próximo do tempo real. Mas os relógios físicos têm taxas de deriva (drift rates) diferentes, sendo ρ a taxa de deriva máxima. Para garantir um erro entre relórios (clock skew) menor que δ é preciso sincronizar periodicamente, com período $\Delta t < \delta/2\rho$.



5.1.1 Algoritmo de Cristian

O objetivo deste algoritmo é sincronizar os relógios dos clientes pelo relógio do servidor.

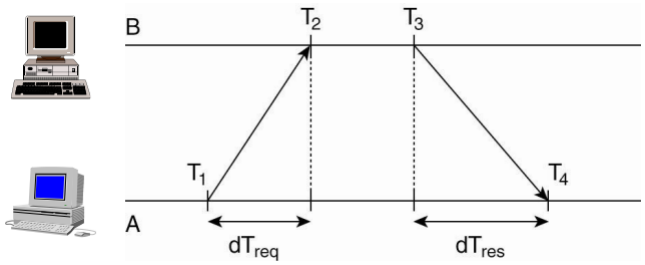
Para acertar o relógio de A a partir do relógio de B:

- Assumir que latência na comunicação é a mesma nos dois sentidos, i.e., $dT_{req} = dTres$
- Calcular o tempo gasto em comunicação (δ):
  - Na resposta o servidor envia os instantes temporais T_3 e T_2

  - O cliente recebe a resposta e determina o valor de δ: $\delta = ((T_2 - T_1) + (T_4 - T_3))/2$

- Calcula o acerto (θ), correspondente á diferença entre o tempo no servidor e o tempo local no cliente: $\theta = (T_3 + \delta) - T_4$

Se $\theta = 0$, A e B estão sincronizados, se $\theta > 0$, A está atrasado em relação a B, se $\theta < 0$, A está adiantado em relação a B (acerta-se aos poucos, para evitar que o tempo ande para trás).



Network Time Protocol (NTP)

NTP é um algoritmo usado para sincronização de relógios na internet, com precisão de 1 a 50 ms. Funciona da seguinte forma:

- Cada par de máquinas A e B aplica o algoritmo de Cristian simetricamente de tempos em tempos e:
  - Armazena os últimos 8 pares (θ, δ) calculados
  - Escolhe o par (θ, δ) em que δ é menor e acerta o relógio com o θ correspondente: $C+ = \theta$

Baseia-se no principio de que se fizermos muitas medições e escolhermos a de menor latência, usamos o valor de θ sujeito a menor erro.

Pode acontecer que uma máquina com um relógio mais preciso se acerte por um relógio menos preciso. Para evitar isto, as máquinas são divididas em estratos, maiores quanto melhor o relógio. Uma máquina A, com estrato n_A, só sincroniza com B, de estrato n_B, se $n_A > n_B$. Após a sincronização, $n_A = n_B + 1$ (A passa ao estrato acima de B).

5.2 Relógios Lógicos

Por vezes não é necessária uma noção absoluta de tempo, bastando acordo em relação á ordem com que ocorrem certos eventos.Os relógios lógicos são usualmente concretizados na camada de middleware.
Relação happens-before($\rightarrow$):

- Se a e b são eventos no mesmo processo e a ocorre antes de b, então $a \rightarrow b$ é verdade

- Se a é um envio de uma mensagem e b o evento da sua recessão, então $a \rightarrow b$ é verdade.

- Transitividade: se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$
- Eventos concorrentes: nem $a \rightarrow b$, nem $b \rightarrow a$

5.2.1 Relógio Lógico de Lamport

Trata-se de um relógio tal que:

- Se $a \rightarrow b$ então $C(a) < C(b)$
- O valor de $C(e)$ nunca decresce
- Dois eventos nunca ocorrem ao mesmo tempo

Construção do Relógio

- Manter um contador $C(e)$ em cada processo, que é inicializado a 0
- Sempre que ocorre um evento interno (i.e., no mesmo processo) relevante, incrementa-se $C(e) = C(ultimo\_evento) + 1$, e atribui-se o seu valor ao evento
- Quando se transmite uma mensagem, anexa-se aos dados o valor atual do contador $C(e\_mesg)$
- Sempre que se recebe uma mensagem, atualiza-se se o contador:

  $$C(e) = max(C(e\_mesg), C(ultimo\_evento)) + 1$$

  e atribui-se o seu valor ao evento de receção.

5.2.2 Difusão com Ordem Total

É possível fazer com que todos os processos processem a mesma sequência de operações, seguindo o seguinte algoritmo:

- Todas as mensagens são enviadas a todos os processos do sistema
- Cada processo do sistema mantém uma fila de mensagens ordenada pelo contador de eventos da mensagem
- Quando um processo recebe uma mensagem, esta é colocada na fila e é enviada uma confirmação a todos os processos do sistema
- Uma mensagem m da fila só é processada quando:
  - m está na cabeça da fila
  - m foi confirmada por todos os processos do sistema

5.3 Exclusão Mútua

Existem duas propriedades a serem satisfeitas:

- Safety:
  - S1: nunca há mais que um processo na secção crítica
- Liveness:
  - V1: se um processo é o único a tentar aceder à secção critica, ele consegue
  - V2: se há um conjunto de processos a tentar aceder à secção critica, então um destes processos acabará por conseguir
  - V3: qualquer processo que tente aceder à secção crítica acabará por conseguir

5.3.1 Algoritmo Centralizado

Neste algoritmo é selecionado um processo como coordenador e assume-se que os processos não falham e mensagens não são perdidas.

- Sempre que um processo quer entrar numa região crítica, envia uma mensagem ao coordenador com o identificador do recurso
- O coordenador devolve uma mensagem indicando que o processo pode continuar (OK), se nenhum outro processo estiver nesse momento na região crítica
- Caso contrário, o coordenador não devolve qualquer mensagem (ou retorna uma mensagem NOK indicando que o processo não tem permissão), e coloca o processo numa fila de espera
- Quando um processo deixa a região crítica, envia uma mensagem ao coordenador libertando o recurso

Satisfaz S1 e V3.

5.3.2 Algoritmo Descentralizado

Este algoritmo é semelhante ao centralizado, no entanto, são selecionados n processos como coordenadores.

- Sempre que um processo quer entrar numa região crítica, envia uma mensagem aos coordenadores com o identificador do recurso a ser acedido
- Cada coordenador devolve uma mensagem OK indicando que o processo pode continuar, ou uma mensagem NOK indicando que o acesso ao recurso está cedido a outro processo
- Um processo apenas entra na sua secção critica se recebe pelo menos $m > n/2$ mensagens OK de diferentes coordenadores

- Se a maioria das mensagens for NOK, o processo avisa os coordenadores que votaram OK que ele não tem acesso e espera uma quantidade de tempo aleatória para voltar a tentar executar o algoritmo

Satisfaz S1 e V1. É possível usar difusão com ordem total para satisfazer também V3 (Descentralizado II).

5.3.3 Algoritmo Distribuído

Assume que é possível ordenar todos os eventos no sistema, com relógios lógicos e que os processos não falham, nem se perdem mensagens.

- Quando quer entrar numa região crítica o processo envia uma mensagem a todos os outros processos, com a seguinte informação:

  $< identificador\_do\_recurso;\ identificador\_do\_processo;\ instante\_temporal >$

- Quando um processo recebe uma mensagem:
  - devolve OK se não está na região crítica e não pretende entrar
  - guarda a mensagem numa fila e não responde se estiver na região crítica
  - se quer entrar na região crítica:
    - compara os instantes temporais do seu pedido e da mensagem
    - devolve OK se o da mensagem for menor
    - guarda a mensagem na fila e não responde se o do seu pedido for menor
- O processo só entra na região crítica quando recebe um OK de todos os processos
- Quando o processo deixa a região crítica, envia um OK a todos os processos que tenham mensagens na fila e depois apaga todas as mensagens

Satisfaz S1 v V3.

5.3.4 Algoritmo em Anel

Assume-se que os processos não falham e não se perdem mensagens.

- Quando o anel é iniciado, associa-se a um processo o testemunho (token)
- O testemunho é trocado entre os processos, sendo passado pela ordem lógica que define o anel
- Um processo apenas pode entrar numa região crítica quando detém o testemunho
- Um processo deve passar o testemunho quando sai da região crítica
- O testemunho continua a ser trocado entre os processos ainda que nenhum deles queira entrar numa região critica

Satisfaz S1 e V3.

5.3.5 Comparação

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	2mk + m, k = 1,2,...	2k, k = 1,2,...	Starvation, low efficiency
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

5.4 Eleição de Líder

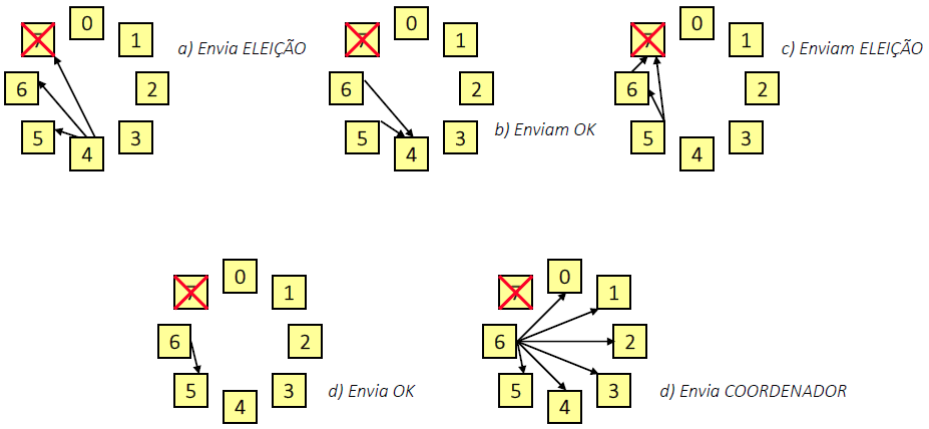
A eleição de um líder deve cumprir as propriedades de segurança (S): após a execução do algoritmo, existe apenas um líder que é conhecido por todos e vivacidade (V): a execução do algoritmo termina.

5.4.1 Algoritmo Bully

Um processo P inicia o algoritmo quando deteta que o líder não responde:

- envia a mensagem ELEIÇÃO aos processos com identificadores superiores
- se ninguém responde, P ganha a eleição e fica como líder
- se um processo com identificador superior responde OK, P termina a execução do algoritmo
- Se um processo recebe ELEIÇÃO de um processo com identificador inferior, responde com uma mensagem OK e executa o algoritmo de eleição se ainda não o tinha feito
- Quando um processo percebe que vai ser o próximo líder (i.e., não recebe OK de nenhum processo com identificador superior a ele), envia uma mensagem COORDENADOR a todos os processos
- Quando um processo recebe uma mensagem COORDENADOR, define o seu emissor como sendo o líder eleito
- Quando um processo parado entra em funcionamento, executa o algoritmo. Se por acaso for o que tem identificador mais alto, envia uma mensagem COORDENADOR a todos os processos passando a ser o novo líder

Grupo com 8 processos (identificadores de 0 a 7), onde o processo 4 deteta que o processo 7 falhou.



5.4.2 Algoritmo em Anel

Assume-se que existem tempos máximos para a comunicação e que os canais são fiáveis.

- Quando um processo P deteta que o líder não se encontra em funcionamento, constrói uma mensagem ELEIÇÃO e envia a ao seu sucessor:

  $$ELEIÇAO =< identificador\ do\ processo >$$

  Se o sucessor estiver em falta, então o processo envia a mensagem para os próximos sucessores até encontrar um em funcionamento

- Sempre que um processo recebe a mensagem ELEIÇÃO, adiciona lhe o seu identificador e passa a ao seu sucessor (ou seguinte que esteja ativo)

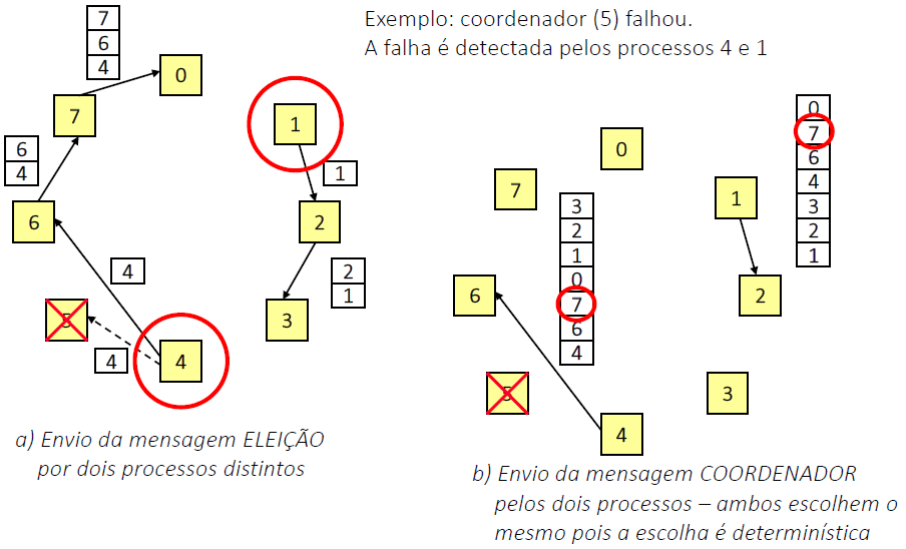
- Quando a mensagem retorna a P:

  - escolhe deterministicamente o próximo coordenador (por exemplo, aquele que tiver o maior identificador)
  - e em seguida faz circular uma mensagem:

    $$COORDENADOR =< coord = ident_i, lista\_de\_proc = ident_k, ... >$$

    que contém a identificação do novo coordenador, bem como dos nós do novo anel

- A mensagem é retirada do anel quando volta a P



5.4.3 Eleição de Líder em Redes Ad Hoc sem Fios

É possível construir um algoritmo que não se baseia nas hipóteses de canais fiáveis ou topologia fixa. Este algoritmo consiste em organizar os processos em árvore, sendo o processo que inicia a eleição a raiz da árvore e o melhor nó é escolhido como líder.

Capítulo 6

Consistência e Replicação

6.1 Consistência dos Dados

Dados são normalmente replicados para melhorar a fiabilidade ou o desempenho, no entanto, a replicação trás o problema da consistência dos dados, já que é preciso garantir que quando uma das réplicas é atualizada, as outras também são. Os acessos aos objetos podem ser divididos em:

- Leitura: pode ser feita em qualquer réplica
- Escrita: tem de ser propagada para todas as réplicas

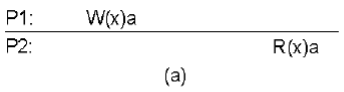
6.1.1 Modelos de Consistência de Baixo Nível

Um modelo de consistência é um contrato entre os clientes e o sistema em que os processos ficam obrigados a fazer os acessos de acordo com certas regras e o sistema tem de se comportar corretamente.

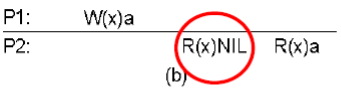
Modelo de Consistência Estrita

Uma leitura no objeto x devolve o valor da escrita mais recente nesse objeto.

Corresponde ao modelo ideal, equivalente a não replicar, mas difícil de implementar, visto que obriga à existência de um tempo absoluto global.



Sistema de armazenamento que **satisfaz** *a consistência estrita*



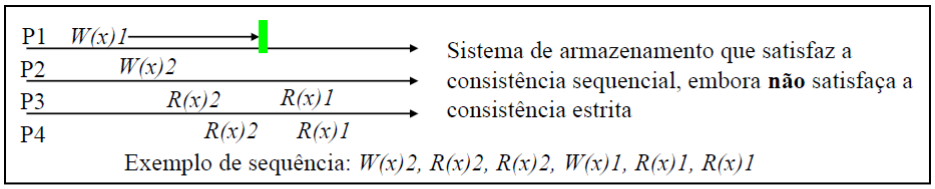
Sistema de armazenamento que ***não*** **satisfaz** *a consistência estrita (primeira leitura de P2 devia ter devolvido* a*)*

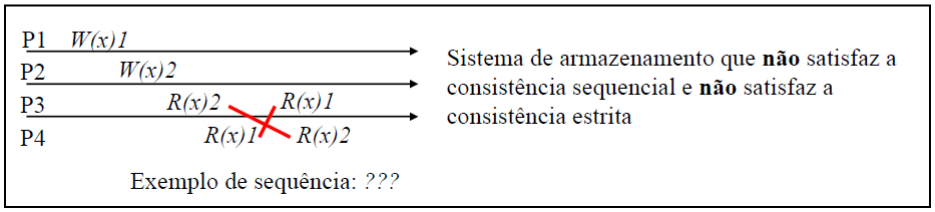
Modelo de Consistência Sequencial

O resultado de qualquer execução é equivalente ao que se observaria se as operações (de leitura e escrita) dos diversos processos fossem executados numa ordem

sequencial e as operações de cada processo individual aparecessem nessa sequência na ordem especificada pelo seu programa.

Neste modelo não existe qualquer referência ao tempo e qualquer sequencia de operações é válida, mesmo as executadas em paralelo.

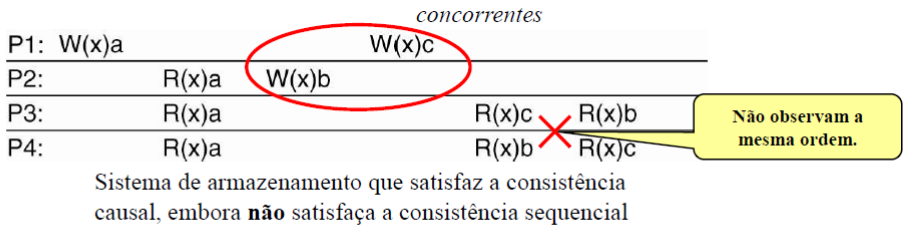


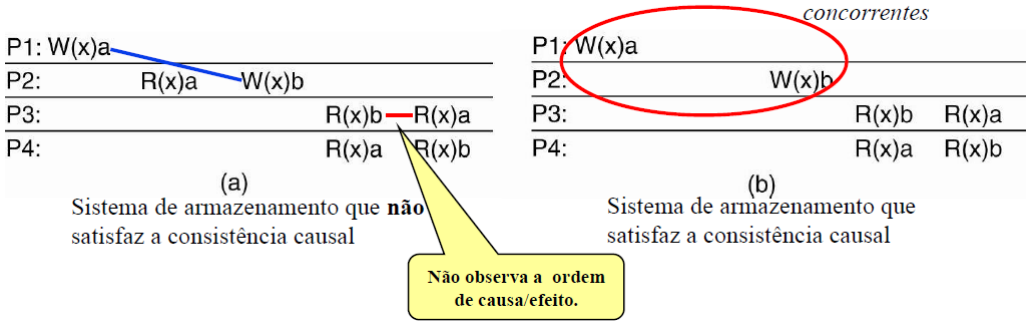


Modelo de Consistência Causal

Escritas potencialmente relacionadas em termos de causa/efeito devem ser vistas por todos os processos do sistema na mesma ordem. Escritas concorrentes podem ser vistas em ordens diferentes por processos diferentes.

Neste modelo é necessário construir um grafo de dependência entre as diversas escritas, de forma a suportar as relações de causa/efeito, o que pode ser feito através de um vetor de relógios.





6.1.2 Modelo de Consistência Centrados no Cliente

Muitas vezes uma operação a ser executada num servidor replicado consiste numa série de leituras e escritas atómicas, em vez das operações de baixo nível vistas anteriormente. Podem então ser usados modelos mais fracos, centrados no cliente e baseados em agrupamentos de operações.

Modelo de Consistência das Leituras Uniformes

Se um processo lê um valor de um objeto x, então qualquer posterior operação de leitura nesse mesmo objeto por esse processo deverá sempre devolver o mesmo valor ou um valor mais recente.

A ideia principal é de que se um processo vê um dado valor num instante, então ele nunca verá um valor mais antigo num instante posterior. É usado em sistemas de armazenamento de emails.

Modelo de Consistência das Escritas Uniformes

Uma operação de escrita de um processo num objeto x é terminada antes de qualquer outra operação de escrita posterior executada pelo processo em x.

Baseado na ideia de que as escritas feitas por um processo num objeto são executadas pela mesma ordem FIFO em todas as cópias desse objeto. Útil na aplicação de patches.

Modelo de Consistência "Ler suas Escritas"

O efeito de uma operação de escrita de um processo num objeto x, será sempre visto pelas operações de leitura posteriores executadas por este processo em x.

Modelo de Consistência "Escritas seguem Leituras"

Uma operação de escrita de um processo num objeto x que sucede uma leitura executada por este processo em x vai afetar sempre o mesmo valor lido pelo processo ou um valor mais recente.

6.2 Gestão da Replicação

A replicação aumenta a fiabilidade, já que os dados se encontram em várias máquinas, prevenindo falhas e melhorando o desempenho, escalando com o número de utilizadores. Existem vários tipos de réplicas:

- Réplicas permanentes: são as réplicas originais do serviço de armazenamento
- Réplicas iniciadas pelo servidor: são criadas temporariamente pelo dono do sistema de armazenamento para melhorar o desempenho
- Réplicas iniciadas pelo cliente: são usadas para armazenar temporariamente objetos que possam vir a ser acedidos no futuro

É possível propagar as atualizações através de uma notificação de modificação de estado, a transferência de uma cópia do estado atualizado ou a propagação da operação que fez a atualização.

6.2.1 Protocolos de Consistência

O protocolo de consistência é responsável por manter as réplicas atualizadas de acordo com um dado modelo de consistência. Podem ser do tipo:

- Push: Os servidores iniciam o processo. A réplica mais utilizada contacta as outras
- Pull: O cliente inicia o processo. As réplicas desatualizadas contactam a mais atualizada
- Protocolos push provisório (lease): o servidor só envia as atualizações durante um período; passado esse intervalo, o cliente volta a pedir mais tempo ou passa a usar um método de pull

Replicação Passiva

Na replicação passiva cada objeto tem associada uma réplica primária responsável por coordenar as atualizações no objeto. Existem duas variantes:

- Réplica primária fixa: todas as atualizações têm de ser propagadas para esta réplica; as outras réplicas (locais) podem ser acedidas para leitura
- Réplica primária móvel: antes da atualização poder ser efetuada, a réplica primária é movida para o sistema local; a escrita depois é efetuada localmente

Se o primário falhar, esta falha tem de ser detetada e deve haver eleição de um novo primário (usando um protocolo de eleição).

Replicação Ativa

Na replicação ativa todas as réplicas têm de começar no mesmo estado e executar os mesmos comandos pela mesma ordem. As réplicas devem ser deterministas, se uma falhar não deve ser feito nada de especial e ao reiniciar é necessário fazer uma transferência de estado.

Replicação com Quóruns

Um quórum é um conjunto de réplicas. Num sistema de quóruns as operações de leitura e escrita são executadas em quóruns de réplicas, onde cada objeto tem um número de versão associado e antes de efetuar uma escrita ou leitura é preciso obter votos das réplicas, com quóruns maioritários, por exemplo:

- Escrita: contactam-se e atualizam-se metade mais uma das réplicas; essas réplicas passam a ter um número de versão maior que o anterior
- Leitura: contactam-se metade mais uma das réplicas e obtém-se os seus números de versão; faz-se a leitura da réplica com o maior número de versão

Em geral tem-se:

- N = número total de réplicas
- N_R = número de elementos do quórum de leitura
- N_W = número de elementos do quórum de escrita

E é preciso garantir que:

- $N_R + N_W > N$
- $N_W > N/2$

Isto é, existe sempre intersecção entre quóruns de leitura e de escrita e há sempre interseção entre quóruns de escrita

Capítulo 7

Tolerância a Faltas

Os sistemas distribuídos são inerentemente suscetíveis a falhas parciais, que não devem comprometer o funcionamento do mesmo

7.1 Noções Fundamentais

- Falta (fault): causa do estado incorreto do sistema (software ou hardware): avaria de um dos componentes, interferência do ambiente, desenho incorreto
- Erro: manifestação da falta no programa ou na estrutura de dados
- Falha (failure): ocorre quando o serviço prestado pelo sistema difere da especificação

As faltas podem ser permanentes, intermitentes ou transitórias.

7.2 Redundância

O método usado para tolerar faltas no sistema é a redundância. Existem vários tipos de redundância:

- Redundância de informação
  - Incluem-se bits extra nos blocos de dados de tal forma que, mesmo que alguns bits do bloco sejam corrompidos, ainda é possível ler o bloco corretamente
- Redundância temporal
  - Repetir uma ação várias vezes para ter certeza que ela tem efeito
- Redundância física
  - Usar várias réplicas de um mesmo componente físico

7.2.1 Sistemas Replicados Tolerantes a Faltas

Um sistema com n replicas é dito tolerante a k faltas se funciona bem mesmo que k das n réplicas falhem.

- Falhas por crash: basta que uma réplica envie respostas corretas aos clientes. Devemos ter, pelo menos, $n \geq k + 1$
- Falhas arbitrárias (Bizantinas): A maioria das réplicas devem ser corretas para evitar que as que falham enviem o mesmo valor incorreto de uma resposta. Devemos ter, pelo menos, $n \geq 2k + 1$

7.3 Acordo em Sistemas Sujeitos a Faltas

Por vezes é necessário um acordo entre nós num sistema sujeito a faltas, como na ordenação de mensagens ou sincronização.

7.3.1 Consenso Baseado em Flooding

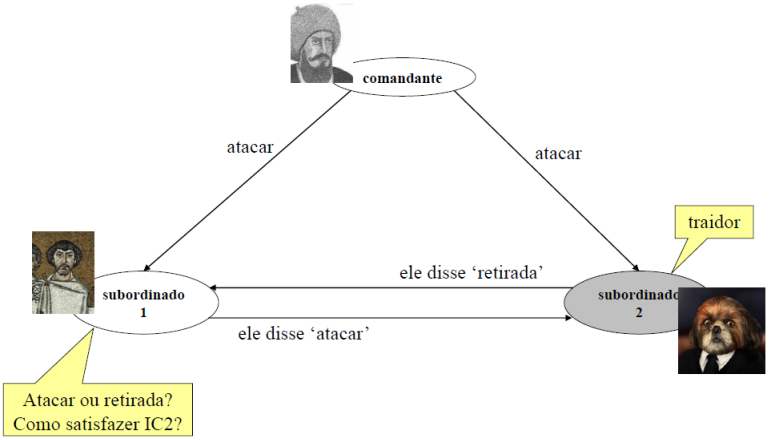
Cada processo:

- Envia a lista propostas que conhece (inicialmente só a sua) a todos
- Espera propostas dos outros e as junta a sua lista
- Se recebeu de todos, decide deterministicamente por uma delas e participa apenas na próxima ronda.
- Se algum falhou passa para a próxima ronda, com a lista atualizada

Só funciona porque conseguimos detetar falhas perfeitamente (impossível na internet).

7.3.2 Problema dos Generais Bizantinos

É um problema fundamental, em que existem n generais e até k podem ser traidores. É impossível de resolver com $2k + 1$ processos.



7.3.3 Algoritmo com Mensagens Orais

Assume-se que:

- Todas as mensagens são entregues corretamente
- O recetor da mensagem sabe quem a enviou
- A perda de uma mensagem pode ser detetada

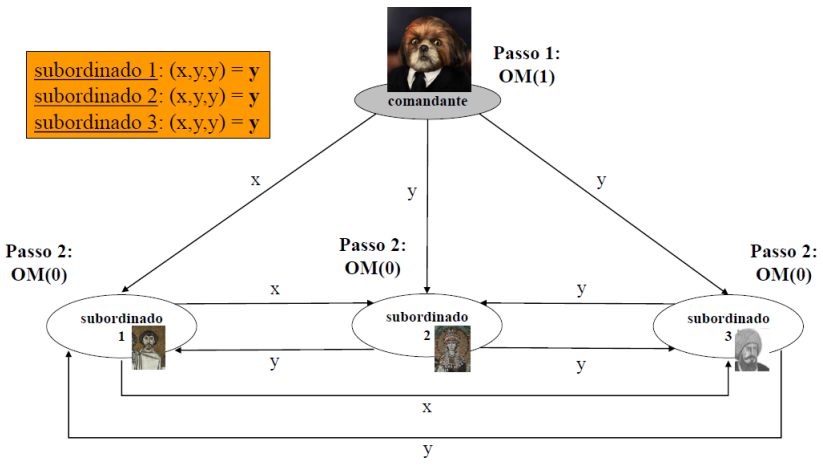
Este algoritmo calcula $maioria(v_1, v_2, ..., v_{n-1})$, em que o valor v_i é a maioria (se existir) ou "retirado". Para $OM(t)$, $t > 0$ (t é o número de falhas a tolerar):

- O comandante envia sua ordem a todos os subordinados.
- Para o subordinado i , v_i é o valor que ele recebeu do comandante ou "retirada"caso ele não tenha recebido ordem.

  O subordinado i age como comandante no algoritmo $OM(t-1)$ para enviar v_i a cada um dos outros $n-2$ subordinados.
- Para cada i e cada $j \neq i$, v_j é o valor que o subordinado i recebeu do subordinado j no passo anterior ou "retirada"se ele não recebeu nenhum valor.
- O subordinado i decide $maioria(v_1, ..., v_j, v_{n-1})$.

E para $OM(0)$:

- O comandante envia a sua ordem a todos os subordinados.
- Cada subordinado usa o valor recebido pelo comandante ou "retirada"caso não receba nenhum valor.



7.4 Semântica do RPC na Presença de Falhas

Na presença de falhas, é difícil fazer com que as RPCs tenham um comportamento idêntico às chamadas locais de procedimentos.

7.4.1 Erro na Localização do Servidor

Neste caso é necessário devolver o erro, através de um retorno de -1 ou lançamento de uma exceção.

7.4.2 Perda do Pedido/Resposta

Aqui uma solução consiste na retransmissão periódica do pedido até que a resposta seja retornada ao cliente.

7.4.3 O Servidor tem uma Falha

O maior problema consiste na determinação do momento em que a falha ocorre, pois, na prática, o cliente apenas percebe a ausência de resposta. Existem várias filosofias de recuperação:

- Pelo menos uma vez: pedido é executado uma ou mais vezes. Espera se o reboot do servidor ou usa se outro servidor
- No máximo uma vez: pedido é executado zero ou uma vez. Basta desistir logo e dar erro
- Exatamente uma vez: pedido é executado uma vez. Em geral não se consegue garantir

7.4.4 O Cliente tem uma Falha

A falha de um cliente após ter enviado um pedido ao servidor pode criar computações órfãs indesejáveis. Existem três possíveis soluções, que não resolvem todos os problemas possíveis:

- Exterminação: antes do cliente enviar o pedido, anota o num log em disco; quando o cliente é reiniciado os órfãos são terminados
- Reencarnação: divisão do tempo em épocas. Começa se uma época sempre que um cliente é reiniciado e faz se broadcast da nova época; quando um servidor recebe mensagem de indicação de nova época, termina todas as computações desse cliente.
- Expiração: associar um tempo T a cada RPC; servidor pede mais T se a RPC precisa de mais tempo; antes de reiniciar o cliente espera T

7.5 Confirmação Atómica

Em transações distribuídas, na execução de uma sequência de operações com atomicidade, ou se executa tudo, ou nada.

Numa transação que envolva acessos a várias bases de dados é necessário que todas tomem a mesma decisão sobre o término de uma transação.

7.5.1 Confirmação em Uma Fase

Num sistema sem falhas basta que uma máquina coordenadora envie uma mensagem definindo se haverá commit ou abort da transação. Essa máquina repetirá a mensagem até que todos os participantes confirmem a operação.

Esta abordagem tem algumas limitações: Um participante não pode decidir unilateralmente por um abort se o cliente pedir um commit e o coordenador não sabe se um participante teve um crash ou foi substituído durante a transação.

7.5.2 Confirmação em Duas Fases

Este protocolo consegue resolver o problema da confirmação atómica. Considera-se uma máquina coordenadora e as restantes participantes. Aqui, nos algoritmos que se seguem os processos (coordenador e participantes) escrevem as suas ações num log estável, de tal forma que possam ser recuperadas após a recuperação do processo, em caso de falha. Assim, qualquer participante pode abortar a sua parte da transação, se uma parte é abortada, toda a transação será abortada.

7.5.3 Confirmação em Três Fases

Confirmação em duas fases pode não chegar a qualquer decisão se o coordenador falhar em certas fases da execução do protocolo, o que resulta no bloqueio dos participantes até que o coordenador recupere. A confirmação em três fases evita esses bloqueios, adicionando um estado de pre-commit com timeout.

7.6 Recuperação de Falhas

Para recuperar de falhas, é necessário armazenar pelo menos parte do processo em memória, para que as operações interrompidas possam ser retomadas após uma falha. Existem duas técnicas complementares:

- Checkpointing: armazenamento periódico do estado do sistema
- Logging de mensagens: armazenamento de mensagens entre checkpoints

Capítulo 8

Sistemas Distribuídos de Ficheiros e na Web

8.1 Sistemas de Ficheiros Distribuídos

8.1.1 NFS

No NFS existe o $fhandle$, que atua como um identificador que representa que representa de forma unívoca um ficheiro num dado servidor. Trata-se de uma estrutura completamente opaca para os clientes.

O servidor devolve um $fhandle$ em todas as operações que envolvem a tradução de um nome ou a criação de um ficheiro. O cliente envia o $fhandle$ sempre que quer efetuar uma operação no ficheiro. Algumas operações suportadas são:

- LOOKUP
- CREATE
- READ
- ...

8.1.2 GFS

O GFS é o sistema de ficheiros usado nos data centers da Google. Possui a seguinte organização:

- Master: servidor que controla os meta-dados do sistema de ficheiros ("serviço de nomes")
- Chunk server: armazena blocos de dados (tipicamente de 64M)

Escala porque o controlo é centralizado, mas o master não é o bottleneck. Os chunk servers fazem a maior parte do trabalho, mas os dados para tradução são mantidos na memória primária do master.

8.2 Sistemas Distribuídos na Web

8.2.1 Nomes na Web

Na web usa-se o Unified Resource Identifier (URI). Existem dois tipos de URI:

- URN (UR Name): nome independente da localização
- URL (UR Location): nome dependente da localização

8.2.2 RPC na Web

Para RPC geralmente utiliza-se o SOAP: Simple Object Access Protocol. O SOAP é uma sintaxe de protocolo centrada em XML. As mensagens são enviadas em envelopes SOAP com cabeçalho e corpo e a comunicação pode ser concretizada sobre SMTP ou HTTP.

8.2.3 Clusters de Servidores Web

Um servidor Web pode ser acedido por muitos clientes ao mesmo tempo. Para aumentar a escalabilidade, replicam se os servidores e faz se balanceamento de carga. Na prática, utiliza-se a distribuição de pedidos de acordo com seu conteúdo, ou content aware request distribution.

8.2.4 Replicação na Web

Existem duas formas básicas de caches na Web:

- Cache dos navegadores
- Proxies de rede

Sistemas mais avançados utilizam redes overlay de distribuição de conteúdos, onde geralmente são replicados conteúdos embebidos, ex: Akamai.