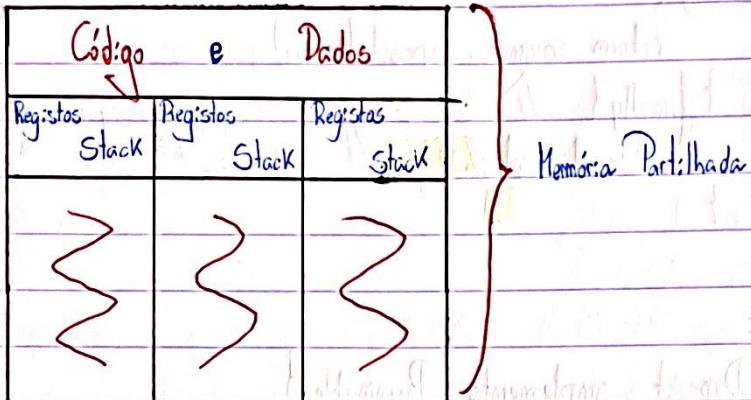


Sistemas Distribuídos

• Threads

- Fios de execução concorrentes de um programa
- Um processo tem, obrigatoriamente, uma ou mais threads.
- Partilham recursos e comunicam entre si através de memória partilhada.



- 1 thread não tem acesso às variáveis que estão na stack
- cada thread tem a sua stack e os seus registros
- cada thread pode executar uma parte diferente do código de forma independente
- podem competir por recursos físicos e lógicos

• Em Java, para a implementação de threads podemos utilizar `Java.lang.Runnable` (implica a utilização do método `run()`) ou `Java.lang.Thread`

• A estrutura para a utilização de threads será algo do género:

1º Main

2º `t = new Thread()`

3º `t.start()`

4º `t.join()`

⋮
espera que a thread termine

5º Continuar a execução do programa

Exemplo:

```
class Bank {  
    private ReentrantLock rl;  
    :  
    boolean deposit (int value) {  
        this.rl.lock(); // bloqueia a zona crítica  
        try {  
            return savings.deposit (value);  
        } finally { // faz com que o que se segue seja SEMPRE executado  
            this.rl.unlock(); // Desbloqueia  
        }  
    }  
    :  
}
```

```
class Deposit implements Runnable {  
    private Bank bank;
```

```
Deposit (Bank b) {  
    this.bank = b;  
}
```

```
public void run() { // método "main" de cada thread  
    final long mDepos = 1000;  
    final int valueDepos = 100;  
    for (int i = 0; i < mDepos; i++)  
        this.bank.deposit (valueDepos);  
}
```

```
{  
:  
:
```

```
ReentrantLock rl = new ReentrantLock();
```

```
Bank bank = new Bank (rl);
```

```
Thread[] threads = new Thread [10]; // array com 10 threads
```

```
for (int i = 0; i < 10; i++) threads[i] = new Thread (new Deposit (bank));
```

```
for (int i = 0; i < 10; i++) threads[i].start(); // inicia todas as threads
```

```
for (int i = 0; i < 10; i++) threads[i].join(); // espera pelo término das threads
```

(2)

- * Se as instruções não forem atómicas poderão ocorrer erros imprevistos pois, por exemplo, a leitura de um valor poderá ser feita por uma thread imediatamente antes de uma outra o alterar. Desta forma, quando a thread que efetuou a leitura for escrever um novo valor, já-lo-á tido em vista o valor antes de ser alterado, podendo, portanto, modificar incorretamente.

THREAD 1	1	2	2	3	(2)
	Read	Write	Read	Write	Read
THREAD 2	1	-	-	2	
	Read			Write	

era suposto ler 3 e não 2

- * Por estarmos perante um evento não atómico (temos leitura + escrita) existe uma leitura que não era suposta ser efectuada com aquele valor.
- * Para comutar este problema temos de garantir a exclusão mútua mas secções críticas, ou seja, nestas zonas apenas 1 thread poderá executar esse bloco de cada vez.

EXCLUSÃO MÚTUAS

- * Propriedade que garante que dois processos ou threads não accederem simultaneamente a um recurso partilhado.
- * Em Java podemos utilizar o **ReentrantLock** para garantir esta propriedade.

ReentrantLock

- * Assegura exclusão mútua
- * Locks são adquiridos por threads (uma de cada vez)
- * Depois de adquirido garante acesso exclusivo aos recursos partilhados

Locks Múltiplos

- Devido a poder querer utilizar diversos recursos de uma mesma classe, pode ser necessário a utilização de um Lock em cada recurso ao invés de um Lock global.

Deadlock

- No entanto, caso ambos os recursos tentem executar uma ação ao mesmo tempo pode causar Deadlocks, a solução para isto é denominada Lock Ordering.

Lock Ordering

- Se A e B efectuam uma mesma ação ao mesmo tempo:
 - A adquire A, B
 - B adquire A, B
- Se: $A \rightarrow B$, $B \rightarrow C$ e $C \rightarrow A$
 - A adquire A, B
 - B adquire B, C
 - C adquire A, C
- Ou seja, é necessário adquirir os locks de forma ordenada e fixa, sendo que depois podem ser libertados normalmente.

`Stream.of(sm, tm).sorted().forEach(m->players.get(sm).i.Q.Lock(m));`

- Mas isto não implica que o primeiro recurso tem vantagem? Não!
 - Por exemplo, se A dispara contra X e X contra A simultaneamente, o resultado será decidido pelo lock de A.
 - Irá funcionar aproximadamente como FIFO.
- Têm exatamente a mesma probabilidade independentemente do lock utilizado.

Multiple Locks

- Adquirir todos os locks necessários no início de uma operação e libertá-los no seu fim funcionará assim como um lock global único.
- Porquê adquirir os locks de forma simultânea?
 - Permitiria alterações não desejadas durante o espaço de tempo entre o unlock do 1º e o lock do 2º.
 - Para evitar isto é necessário dar lock de ambos os objetos.

(3)

✗ Quanto tempo é possível atrasar o lock do 2º recurso?

→ Até ser necessário modificar:

✗ Quanto tempo podemos antecipar a libertação do 1º lock?

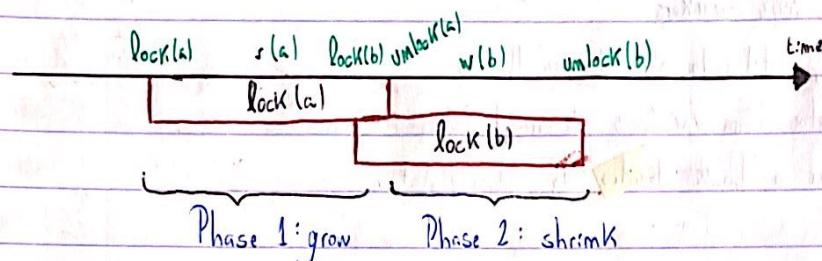
→ Após efetuar as modificações mele desejadas e após dar lock do segundo

Two Phase Locking (2PL)

Regra 1. Todos os lock(a) precedem unlock(b)

Regra 2. Os dados de um item são lidos / escritos dentro do lock correspondente

- equivalente a possuir todos os locks ao mesmo tempo



Locks vs Variáveis

✗ "Que lock corresponde a cada porção de dados?"

✗ Múltiplas threads que accedem aos mesmos dados de forma concorrente devem adquirir o mesmo lock

✗ O programador é que é o responsável por isto!

Problemas

Variáveis automáticas

✗ Variáveis exclusivas de métodos são criadas em cada chamada do método

✗ O método continua a poder aceder ao estado partilhado (variáveis de instância, variáveis static de classes)

Solução: declarar o lock como variável da instância

Variáveis de classe / globais

✗ Variáveis marcadas como static em Java podem necessitar de controlo de concorrência

→ Não é necessário se forem "final"

→ Não é necessário se a classe só for acessada por uma thread

SOLUÇÃO: declarar o Lock como uma variável de instância "static"

Locks Encapsulados

✗ Mantém as variáveis e os locks correspondentes encapsulados no mesmo objeto

SOLUÇÃO: declarar o Lock dentro da classe que utiliza a classe encapsulada

Shared vs Thread-Local

✗ Os estados dos programas contêm:

→ Local thread nos workers

→ Shared state, utilizado por todas as threads

✗ Os diferentes objetos têm variáveis de instância

SOLUÇÃO: declarar o Lock dentro do SharedState.

Collections e Getters

✗ Os getters podem devolver referências a coleções partilhadas

→ Iterators envolvem referências aos objetos originais

SOLUÇÃO: reconsiderar o Lock encapsulado.

Readers - Writers Locks

✗ Exclusão mútua com locks é demasiado conservadora:

→ Mais do que um leitor não é um problema

→ Um "escritor" deve excluir todos os outros (leitores e escritores)

✗ Métodos diferentes para leitores e escritores:

Lock readlock()

Lock writeLock()

✗ Mais caro que um lock simples

✗ Se der preferência aos leitores:

→ Permite a entrada de novos leitores, mesmo que um escritor esteja à espera

→ O escritor pode entrar em starvation

✗ Se der preferência aos escritores:

→ Não permite que leitores sejam escritores se um escritor estiver à espera

- Pouca concorrência entre leitores.
- ✗ Justo e eficiente
- Leitores e escritores ordenados em FIFO
- Permite que os leitores ultrapassem até os escritores na queue.

Lock Managers

- ✗ Locks individuais são inefficientes em coleções grandes
- ✗ LockManager é uma interface que permite dar lock a apenas 1 objeto de uma coleção

Multiple Granularity Locks

- "intention" locks em comentários
- "actual" lock no alvo
- ✗ Intention entra em conflito com Actual, mas não com outros Intentions

• CONDITIONS

ESPERA POR EVENTOS

- ✗ Ineficiente se a thread que está à espera está numa espera ativa
- ✗ Provoca deadlocks / race conditions se a thread for suspensa pelo SO

Condition Variables

- ✗ Suspende uma thread atomicamente e libera o lock

Lock l = new ReentrantLock();

Condition c = l.newCondition();

c.await(); ← atomicamente libera l e suspende a sua execução, bloqueia l quando "acorda"

c.signalAll(); ← acorda todos os threads suspensos

c.signal(); ← acorda uma thread

↳ acorda a thread mais antiga (FIFO)

pode acordar sem o signal (então convém colocar num ciclo)

Ordemação

- ✗ Apesar do signal trabalhar com FIFO, o recurso poderá ter de competir pelo lock visto que o ReentrantLock pode alocar o lock em qualquer thread que esteja em espera.

Solução: formar a ordem explícita (p.e. associar um ticket a cada viral)

• ALGORITMOS DE EXCLUSÃO MÚTUA

LockOne

- ✗ Execuções concorrentes podem ter deadlocks
- ✗ Execuções sequenciais funcionam

LockTwo

- ✗ Sempre a exclusão mútua
 - ✗ Não é livre de deadlocks
- Execução sequencial provoca deadlocks, concorrente não provoca

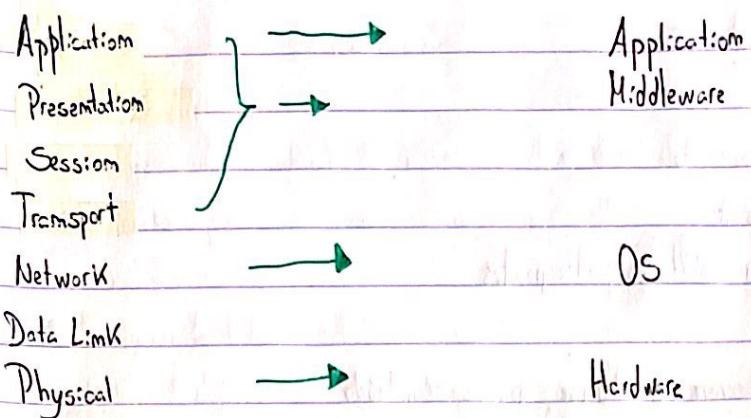
Petersom

- ✗ Não permite exclusão mútua
 - ✗ Livre de deadlocks
 - ✓ Livre de Starvation
- (...)

• Comunicação

OSI

- ✗ Camadas como abstrações
- Diferentes maneiras de interpretar a mesma coisa
- Utilizadas para organizar e padronizar a comunicação



- ✗ A camada do middleware é responsável por encapsular soluções para problemas difíceis:
 - Mais do que as camadas sessão/presentação
 - Mais do que comunicação (persistência de dados)
 - Determina como as aplicações são desenvolvidas

TCP / IP

- ✗ Comunicação comporta-se como:
 - um par de pipes do OS
 - construção de dois buffers concorrentes
- ✗ É uma primitiva de comunicação e sincronização distribuída
 - Permite esperar por outros processos / threads em hosts diferentes

Sockets

- ✗ Interface para:
 - Criar uma nova conexão
 - Enviar e receber dados
 - Fechar uma conexão existente

Middleware : Sincronização e persistência

- ✗ Assíncrono vs Síncrono
 - Thread não bloqueada enquanto interage
 - Thread bloqueada até receber resposta
- ✗ Transiente vs Persistente
 - Mensagens persistentes guardadas em disco em diversas fases do processo

→ Utilizado para reiniciar interação

Middleware: Destino e Endereçamento

✗ Point-to-point vs multipoint

→ Interacção entre dois vs múltiplos participantes

✗ Explícito vs Implícito

↳ Subscrive o interesse, publica conteúdo

↳ Junta-se ao grupo e envia para lá

• SERIALIZATION / MARSHALING

Representação

✗ Endianness

→ Inteiro, pode ser em:

→ Big endian bytes

→ Little endian bytes

✗ Character encoding

→ String, pode ser em:

→ UTF8

→ Latin1

Problema de Design: Singular ou Múltiplo

✗ Acordo numa representação comum ("network byte order")

→ Converter quando se envia

→ Converter quando se recebe

• Mesmo que o sender e o receiver sejam idênticos

✗ Utiliza a representação no sender:

→ Envio com uma tag

→ Dependendo da tag, conversão quando se recebe

Problema de Design: Binário vs Texto

✗ Formatos de texto:

- Legível e robusto
- Redundante e lento para parsing
- ✗ Formatos binários:
- Compacto e eficiente
- Ópcos (difícil para debug) e frágeis

Alinhamento e padding

- ✗ Memória é endereçada em offsets de bytes
- ✗ Acessados como words multi-byte
- ✗ Acessos não alinhados:
 - Lentos ou não permitidos
- ✗ Formatos binários devem utilizar padding para alinhamento

Tipos Compostos

- ✗ Records
 - Enumera cada um dos componentes
 - Inclui padding opcional
- ✗ Objects (com subclasses) e unions:
 - Conteúdo contém um prefixo com uma tag que identifica a opção a ser usada
 - Utiliza a tag para determinar o que deserializar
- ✗ Itens opcionais (p.e. campos que podem ser null)
 - Prefixo com um booleano indicando se está presente

Collections

- ✗ Primeira opção:
 - Prefixo indicando o número de componentes e, em seguida, cada um dos componentes.
 - Comum em representações binárias
 - Melhor opção caso o tamanho possa ser determinado facilmente
- ✗ Segunda opção:
 - Cada componente e, em seguida de cada um, o valor especial de finalizador do

término

- Corrimum em representações baseadas em texto
- Melhor opção no caso das estruturas podermos crescer de forma dinâmica

Gráficos

- ✗ Uma travessia simples não é suficiente (pode serializar dados repetidos)
 - Utilizar tags e um map auxiliar para serialização
 - Na desserialização é necessário manter a ordem dos objetos para restaurar os pointers

Código de Conversão

- ✗ Enumera os componentes para serem escritos / lê-los atravessando, de forma recursiva, as estruturas de dados
- ✗ Métodos de filtragem
 - Evitando e propenso a erros
- ✗ Reflexão
 - Permite o override de dados transitentes (Locks, Caches, ...)
- ✗ Código gerado
 - Interface da linguagem e compilador

Problema: Programa vs Data First

- ✗ Program First: formato dos dados inferidos através de um programa existente
 - Convenientemente para desenvolvimento
 - Preso a uma linguagem
- ✗ Data First: programa gerado através de uma descrição de dados abstrata
 - Independência da linguagem e do middleware
 - Força o programador a utilizar novas ferramentas

Problema: Versioning

- ✗ Se a estrutura de dados mudar, o receiver irá "morrer" ou descodificar dados corrompidos
- ✗ Permite que a estrutura de dados seja modificada através de novas versões do

Programa

- ✗ Fazer com que ítems individuais sejam opcionais e/ou providenciados de fábricas
 - p.e. Protobuf
- ✗ Permitir o versionamento da estrutura como um todo
 - p.e. Java Object^{*} Stream

Problema: Streamlining vs Object model

- ✗ Streamlining:
 - Dados copiados diretamente de/para representação externa.
 - Layouts internos e externos são os mesmos
- ✗ Object model:
 - Um model intermediário é construído em memória
 - O model pode receber questões e travessias
 - Algumas aplicações utilizam o model intermediário diretamente

• PROCESSOS E THREADS

Cliente - Servidor

- ✗ Comunicação assimétrica
 - Servidor passivo espera por pedidos
 - Clientes ativos efetuam pedidos e aguardam respostas
- ✗ Pode ser visto como um cliente ter a sua thread migrar para o servidor para efetuar uma tarefa
 - Requer cooperação entre múltiplas threads do SO em ambos os lados
 - Interage com o gestor de conexões

Single-threaded

- ✗ Não é tão útil como o modelo geral, visto que o servidor só consegue lidar com uma conexão.
- ✗ Pode ser usado em:
 - Protocolos de comunicação connection-less
 - Meta-servers

Thread-per-connection

- Modelo típico e geralmente útil
- Adequado para clientes single-threaded
- Adequados para pedidos rápidos e non-blocking

Thread-per-request

- Modelo geral para pedidos multi-threaded e de longa duração
- Pedidos são respondidos numa ordem arbitrária
 - necessita de identificadores
- Overhead adicional para cada pedido:
 - criação de uma nova thread
 - troca de contexto e sincronização para entregar um pedido

Thread Pool

- ✗ Similar ao thread-per-request, mas...
- Pedidos são armazenados numa queue através de um buffer
- Reduz o overhead dos pedidos reutilizando threads
- Fornece um mecanismo de controlo de admissão para restringir a quantidade de recursos utilizados no servidor

1-to-n Notifications

- ✗ O que acontece se um pedido provocar respostas a diversos clientes?

→ Escrever diretamente em cada socket:

- Sequencial e pode bloquear
- A solução é ter uma segunda thread e uma queue para cada conexão:
 - O pedido dá enqueue e responde ao ~~thread~~ aos diversos destinos.
 - A thread do writer "acorda" e escreve para os sockets

Single-Threaded

- ✗ Aplicação possui apenas uma thread
- ✗ A thread da aplicação envia pedidos e aguarda por respostas

Multithreaded

- ✗ Aplicação possui múltiplas threads
- ✗ O código do cliente é protegido numa seção crítica
 - Apenas uma thread utiliza a conexão
 - As threads acumulam-se numa queue para um serviço remoto
- ✗ Invocações remotas são propensas a demorar muito

Multithreaded com Dispatcher

- ✗ Múltiplas threads de aplicações efetuam pedidos sem esperar pelas respostas
 - Os pedidos devem possuir ids
- ✗ Uma única thread é dedicada a coletar respostas e colocá-las no buffer
- ✗ As threads da aplicação coletam as respostas corretas do buffer

Callbacks

- ✗ E se o dispatcher do cliente receber um pedido do servidor?
 - O dispatcher pode utilizar uma thread-pool local para o executar e em seguida responde-lhe
- ✗ Isto é um modelo generalizado e simétrico que permite que o servidor possa "callback" o cliente a qualquer altura.

Migração do código para o servidor

- ✗ Evita que dados ~~migrarem~~ maveguem ao longo da rede

Migração do código para o cliente

- ✗ Evita a latência de múltiplos round-trips ao servidor

Desafios

- ✗ Problemas de Conflito principais:
 - Segurança: Restringe o que o código pode fazer
 - Eficiência: Permite que o código corra neto do hardware
- ✗ Heterogeneidade:
 - Diferentes arquiteturas de processadores

→ Diferentes SOs e bibliotecas

Computação na Cloud

- ✗ Combina a virtualização com infraestruturas programáveis pay-per-use

Edge & Fog Computing

- ✗ Servidores colocados na fronteira da rede (CDN)
- ✗ O fog computing combina a cloud com a fronteira

Inovação Remota

RPC/RMI

- ✗ Middleware que esconde a interação cliente/servidor como uma invocação a um procedimento (método)
- ✗ Junte:
 - Comunicação entre sockets
 - Serialização
 - Estratégias de threading em clientes e servidores
 - Naming

Naming

- ✗ Servidor registra referências ao serviço de naming
- ✗ Cliente procura o endereço utilizando o nome
- ✗ O lookup encontra-se encapsulado no código do cliente

Geracão de Código

- ✗ O código do servidor e o stub são mecanicamente determinados através do protocolo da interface

✗ Pode ser gerado através de uma descrição

Code first:

→ Escrever código, gerar stubs utilizando reflexão

Protocol first:

→ Escrever uma definição abstrata de uma interface, gerar stubs utilizando um compilador

→ Utiliza um IDL

Passagem de parâmetros

- ✗ Parâmetros são copiados do cliente para o servidor
- ✗ Em alguns casos, estes podem ser copiados de volta:
 - Pointers em C
 - Objetos em Java
- ✗ Os parâmetros podem ser etiquetados como in, out ou in-out

Handle de Erros

- ✗ Problemas de conexão / server não disponível não pode ser escondido:
 - Não existem situações correspondentes em sistemas não distribuídos
- ✗ Semânticas possíveis:
 - No máximo 1x: try e throw da exception
 - No mínimo 1x: try repetidamente até acknowledged
 - Pode bloquear eternamente
 - Válido apenas para operações idempotentes

Invocação de um método remoto

- ✗ Cada objeto possui um identificador
- ✗ No cliente:
 - Criação em stub, para cada objeto
 - Associa a cada pedido um prefixo com um identificador
- ✗ No servidor:
 - Criação de um wrapper que decodifica os pedidos para cada objeto (skelton)
 - Mantém um map de identificadores → skeletons
 - Faz lookup do objeto em cada pedido

Referências Remotas

- ✗ A informação necessária para criar um stub é:
(Classe do Objeto, Endereço do Servidor, Identificador)
- ✗ Referência distribuída ("pointer" num sistema distribuído)
- ✗ Uma referência distribuída é uma versão serializada de um stub
- ✗ Quando um stub é passado como parâmetro ou devolvido por um método:
 - Envie a referência
- ✗ Quando uma referência é recebida:
 - Recriação do stub
- ✗ Permite um sistema distribuído OO, de forma geral, em que cada objeto possa chamar qualquer objeto

Referência vs Valor

- ✗ Objetos podem ser devolvidos por referência ou valor
 - Múltiplos round trips vs Estado partilhado
- ✗ Necessário reconsiderar o design da aplicação:
 - Imutável vs Estado partilhado
- ✗ Java RMI utiliza markers interfaces
 - Remoto é passado por referência
 - Serializable é passado por valor
- Naming / DHT

Coordenação

- ✗ Sistemas distribuídos são, geralmente, definidos como:
 - Coleções de elementos computacionais autônomos mas
 - Resultam num sistema singular coerente
- ✗ É possível visto que os elementos computacionais coordenam-se de forma a efectuar uma função coerente.

Aprendizagem de Outcomes

- ✗ Recomenda a forma como problemas concretos do mundo se mapeiam em problemas genéricos de sistemas distribuídos
- ✗ Recomenda uma solução que serve no ambiente fornecido
 - Conhece os trade-offs de cada algoritmo

Naming

- ✗ Como encontrar o servidor e os endereços de serviços?
- ✗ Endereços máo são convenientes para a leitura de humanos
- ✗ É um desafio nos sistemas distribuídos

Flat naming

- ✗ Local Server:
 - Um servidor mantém um map que associa nomes e endereços
 - Servers registram o nome
 - Clientes pedem o nome
- ✗ Broadcast:
 - Envia uma pergunta para toda a rede: "Quem é X?"
 - Alvo responde: "Sou X!"
- ✗ Não é escalável para um grande número de clientes e servidores
- ✗ Não permite autoridade administrativa distribuída.

Naming Hierárquico

- ✗ Lookup eficiente ($\approx \log N$)
- ✗ Permite autoridade administrativa distribuída
- ✗ Continua a existir um bottleneck e SPOF no modo raiz

Hash Table Distribuída (DHT)

- ✗ Não existe um modo raiz
- Sem bottleneck e SPOF
- ✗ Lookup eficiente ($\approx \log N$)

• TEMPO E EXCLUSÃO MÚTUA

Sincronização de Relógio

✗ Os relógios do hardware não são perfeitos e variam com o tempo

✗ Pode ser um problema com:

→ Ficheiros partilhados

→ Certos algoritmos

✗ O relógio deverá ser ajustado com pequenas incrementações durante um longo

período de tempo tornando-o mais rápido ou mais lento

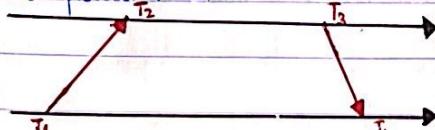
✗ Na prática, existem alguns fatores imprevisíveis:

→ Atrasos de transmissão

→ Atrasos de processamento

✗ A melhor forma é ajustar com base numa estimativa do delay de uma mensagem

→ Network time protocol (NTP)



• assume que os delays são iguais: $= ((T_2 - T_1) + (T_4 - T_3)) / 2$

• repete várias vezes o processo e escolhe o delay mais pequeno

✗ Reference Base Synchronization (RBS):

→ assume que existe true broadcast medium (delay ≈ 0)

Exclusão mútua

✗ Solução num sistema distribuído?

✗ Considera-se:

→ Número de saltos de mensagem necessários para entrar na seção crítica

→ Balanceamento de carga

✗ Queue centralizada mantida por um coordenador

• 1 round-trip para entrar / carga assimétrica

✗ Troca de um token num anel

• $m/2$ saltos para entrar / carga simétrica

✗ É difícil alcançar um algoritmo distribuído:

→ Como os pedidos de locks concorrentes só recebidos por destinos diferentes em ordens

diferentes, a segurança não é garantida.

✗ Aproveitar os relógios sincronizados

- Assumir $\delta > \text{delay da transmissão} + \text{skew}$

- Considerar apenas mensagens entre $t \in \delta$, ordenadas através de uma timestamp

Tempo e causalidade

✗ $\text{Clock}(i) \rightarrow$ tempo em que o i aconteceu

✗ $i \sqsubset j$ precede j then $\text{Clock}(i) < \text{Clock}(j)$

✗ Para um evento j :

→ Quando se tem a certeza que não existe qualquer i tal que $\text{Clock}(i) < \text{Clock}(j)$, então não existe qualquer i que precede j

Relógios Lógicos de Lamport

✗ Evento local: incrementa o contador

✗ Evento de envio: incrementa e dá tag com o contador

✗ Evento de receção: atualiza o counter local para o máximo e, em seguida, incrementa

Exclusão mútua

✗ Resumo do Algoritmo:

→ Inicia assumindo que os processos encontram-se a trocar mensagens de forma contínua

→ $r[i, j]$ é o último timestamp de j em i

→ Considerar os pedidos com $t \leq \min(r[i, j], \text{local } j)$

→ Ordenado através do timestamp, o desempate em caso de mesmo timestamp é feito através do id do processo

✗ A versão completa é dada pelo Ricart-Aryawala distributed mutex ~~algorithm~~ algorithm

✗ 1 salto para entrar / sair simétrica

Tempo Lógico

✗ A abordagem utilizada para a waiting queue no mutex pode ser utilizada em outras aplicações determinísticas

→ Replicated State Machine (RSM)

✗ O tempo lógico é aplicável em diversas resoluções de problemas em sistemas distribuídos

• Acordo

Commit Transacional

- ✗ Coordenação de múltiplas ações irreversíveis ao longo de um sistema distribuído.
- ✗ Transações distribuídas com 2-phase commit (2PC) suportam o acordo em sistemas com falhas
 - Limitado ao crash-recovery do coordenador
- ✗ Muito utilizado no middleware em contexto empresarial para a integração da aplicação

Replicação

- ✗ Mantém diversas cópias acerca dos mesmos dados ou serviços
 - Distribui a carga para permitir escalabilidade
 - Tolerar falhas do servidor
- ✗ Solução máis: escrever e depois propagar
 - divergente
 - volta atrás no tempo
 - não é f_t
- ✗ 2PC: Avança para a próxima fase quando possuir todos os dados (tolera reboots, mas não crashes)

Quorum

- ✗ Regras para dados replicados:
 - $N_r + N_w > N \rightarrow$ os readers obtêm o último valor
 - $N_w > N/2 \rightarrow$ conflito entre writers concorrentes
- ✗ Regras adicionais para tolerar falhas assumindo no máximo f falhas:
 - $N_r + f \leq N \rightarrow$ readers nunca bloqueiam
 - $N_w + f \leq N \rightarrow$ writers nunca bloqueiam
- ✗ Pode ser configurado para assegurar ambos ou nenhum
- ✗ A solução típica é possuir uma maioria:
 - $N_r = N_w = f + 1$
 - $N = 2f + 1$

Eleição do Líder (Bully algorithm)

- ✗ O processo que pretende ser o Líder (suspeita que não existe um Líder), faz broadcast do seu endereço para todos os outros.
- ✗ Se for recebido um endereço $>$ local, reconhece-se o novo Líder.
- ✗ Se endereço recebido $<$ local, faz-se broadcast do endereço local (bully).
- ✗ Assim sendo, podem existir muitos líderes ou memhums.

Consenso

- ✗ Problema de acordo distribuído em que os processos propõem alternativas e decidem da seguinte forma:
 - todos os participantes corretos obtêm uma decisão
 - todas as decisões são iguais
 - a decisão é uma das alternativas propostas.
- ✗ Como resolver o problema?
 - 2PC + Quorum + Bully \approx Consenso Paxos

Paxos

- ✗ Processos têm 3 roles principais:
 - Proposers (com um Líder) \rightarrow coordenam a votação
 - Acceptors \rightarrow votam acerca de uma decisão
 - Learners \rightarrow informados acerca da decisão
- ✗ Normalmente, cada participante toma os 3 roles
- ✗ 1^ª fase é a eleição do Líder + leitura quorum
 - Um proposer tenta tornar-se Líder $\xrightarrow{\text{Propose (1a)}}$
 - Utiliza (time, id) para efectuar o bully aos outros
 - O acceptor promete esquecer os líderes anteriores $\xrightarrow{\text{Promise (1b)}}$
- ✗ Um Líder é estabelecido com a maioria das promessas e usa o valor lido ou, caso não haja memhum, a sua própria opinião como valor candidato.
- ✗ 2^ª fase é o quorum write:
 - Accept (2a)
 - o Líder impõe a sua própria decisão
 - Accept (2b)

- Aceite se o líder não for deposto
 - O acceptor confirma a decisão ao líder e aos learners
 - ✗ Ao receber a maioria das mensagens a decisão está feita
 - ✗ Pode sempre recuperar de uma memória de processos falhados caso exista um líder confiável
 - ✗ Quais são os piores casos?
 - Não há líderes confiáveis (sistema instável)
 - Mais do que uma memória de falhas (falha catastrófica)
- Nota:** Em ambos os casos o algoritmo para não produz decisões inconsistentes

Aaplicações do Consenso

- ✗ Replicated State Machine: réplicas executam a mesma sequência de comandos
 - Clientes emitem comandos para os proposers
 - Learners executam cada pedido decidido como sendo o seguinte e enviaram as respostas aos Clientes
- ✗ Comunicação de grupo: Acordo em processos operacionais que recebem cada mensagem
 - A decisão é da responsabilidade dos membros do grupo e as mensagens são vistas por todos os participantes.

Multicast

Nível de Aplicação Multicast

- ✗ Envia, de forma confiável, para múltiplos destinos (grupos)
- ✗ De forma informal, a confiabilidade significa que todos os destinatários devem receber a mensagem
 - todos os destinatários entregam todas as mensagens enviadas
- ✗ Na realidade, os destinatários e os remetentes podem falhar, no entanto:
 - todos os destinatários corretos enviam as mesmas mensagens
- ✗ Como podemos assegurar isto?

Aproximação geral

- ✗ Buffer e retransmissão até receber acknowledgement

Acknowledgments

- ✗ Não escalável para um número grande de destinos devido à possibilidade de "ack implosion"

(13)

Multicast com agreement

- ✗ Ack's precisam de ser enviados para todos os destinos, o que provoca " $O(m^2)$ ack implosion"

Gossip

- ✗ Protocolo simples que permite o multicast de uma mensagem:

- Seleciona um pequeno subconjunto de alvos aleatórios
- Envia a mensagem apenas para esses alvos
- Descarta a mensagem

- ✗ Enquanto se recebe uma nova mensagem, atua-se também como remetente

Gossip e Epidemias

- ✗ Similar com epidemias:

- Remetente - contagioso - espalha o "rumor"
- Destinatário - infectado - conhece o "rumor"
- Ignora duplicados - morto - notícias antigas

- ✗ Parâmetros interessantes:

- m (tamanho da população)
- d (número de alvos)