Índice

1. Processos vs Threads

Thread	Processo
Todas as variáveis são partilhadas	As variáveis não são partilhadas
Podemos usar as mesmas variáveis para ler e escrever a partir dos vários threads	Não podemos usar as mesmas variáveis para ler e escrever a partir dos vários threads
Um thread é uma sequência de comandos sendo executados num programa	Agrupar os recursos que estão a ser usados

Nota: Executar o mesmos programa 2x não é criar mais threads, mas sim criar 2 processos para o mesmo programa.

2. Corridas

O que são? Resultado total depende da velocidade de execução dos vários threads. Levam a erro!
Como evitar? Fazer com que não se encontre mais do que um thread ao mesmo tempo a executar na SC.
Como o fazer? Etiquetar o conjunto de linhas de código onde há mudança de local do código como sendo atómicas ou indivisíveis.
Fazer uso de exclusão mútua! (lock/unlock)
Try finally em vez de try catch: para garantir que há um unlock correspondente a cada lock.

3. Exclusão mútua

Propriedades:
1) proteção (safety): há algo de mau que não deve acontecer. ex.: impedir 2 threads numa SC ao mesmo tempo.
2) animação (liveness): esperamos que o nosso programa faça algo. ex.: não há deadlocks (um dos threads consegue entrar na SC).

Exemplo: 2 threads querem entrar numa SC.

- Utilização de 2 flags;
- Protocolos definidos;

Variante do protocolo: Algoritmo de Peterson.

Se thread 1 está na SC: flag[1]=True; viction=2.

4. Algoritmo de Peterson

- O 2° thread a chegar a uma SC, é sempre a vítima quando há uma colisão.
- Garante exclusão mútua.
- Não há deadlocks. Um thread só fica bloqueado enquanto for a vitima (e só no ciclo while).
- Starvation free.

4.1.Falha do algoritmo de Peterson

Devido ao normal funcionamento de um processador (1º escreve na cache; 2º cache->memória), a modificação de uma variável num programa pode estar em trânsito na cache do cpu, quando há mais do que uma variável. Neste caso, não consegue garantir exclusão mútua.

Solução: escrever o que está na cache do cpu para a memória com a utilização de barreiras de memória. Como é que isto é feito? Declarando as variáveis como 'volatile'(programa + lento) ou 'atomic'. Desta forma, temos a certeza que vamos observar os valores pela ordem certa.
Nota: Arrays é sempre atomic, porque só assim conseguimos ter em conta os valores individuais.

5. Algoritmo de filtros

- Generalizar o algoritmo acima para n threads. Temos de parar n-1 dos n threads de maneira que só um deles entre na SC. Em cada nível, pelo menos um entra e pelo menos um fica bloqueado.
- Como saber se a bandeira está erguida? Se o nº de linhas ocupadas é >= nºlinha.
- Para cada nível L temos no máximo n-L threads que entram nesse nível.
- Garante exclusão mútua(no último nível → L=n-1).
- Não garante justiça, porque os threads podem ser ultrapassados. No starvation.

6. Algoritmo da padaria

Usado para manter a ordem relativa dos threads que tentam aceder à SC. FIFO. Resolve o probelma do deadlocks quando não há progressos por parte de nenhum dos locks, porque há sempre um thread que tem a senha mais antiga.
Exemplo: Se o A é sempre o 1º a ser adquirido, isto não faz com que o A tenha vantagem no jogo, porque a ordem de aquisição de locks é definida pelo lock A. As hipóteses são as mesmas, pq ambos usam a mesma regra para escolher o thread que entra.

7. Locks no mundo real

No mundo real não fazemos uso destes algoritmos porque, p.e., para centenas de locks teríamos de associar um grande nº de arrays a cada lock. Consome tempo e energia do cpu, visto que ficam a observar a mesma variável quando há contenção numa SC.

Operações atómicas: conseguem ler e escrever uma variável de forma indivisível. Usadas para verificar se a SC está disponível.

Nota: Há um lock escondido em cada objeto definido por 'synchronized'. Isto faz com que tenhamos de largar os locks pela ordem inversa que os adquirimos. Poderá não ser desejável.

Sincronização: para ler valores consistentes(recentes). Evita esperas.

Um programa eficiente resume-se a:

- reduzir a contenção e tempo nas SC;
- fazer com que as SC sejam pequenas e sejam usadas o menor nº de vezes possível;
- usar sincronização para evitar esperas.

8. Multiple locks

Exemplo: Jogo de CS. Queremos ao mesmo tempo diminuir a vida de um e aumentar a vida de outro.
Se adquirirmos os locks separadamente para modificar cada uma destas variáveis, é possível que um observador conclua que há um jogador que tem pontuação que não corresponde ao decréscimo de vida de outro (Cheating!). **Solução**: Adquirir 2 locks ao mesmo tempo.

Lock later: podemos adiar a aquisição de um dos locks sem que isso contribua para quaisquer problemas. Contudo, não podemos adiantar esta libertação deixando a escrita fora da SC, porque levaria a corridas.

9. 2PL:

1) Todas as operações de lock têm que preceder todas as operações de unlock.
2) O acesso a cada item de dados é feito apenas durante o período em que o lock correspondente está adquirido.
1ª fase: Grow. Vamos acumulando locks.
2ª fase: Shrink. Encolher a quantidade de locks que temos adquiridos.

Vantagem: Reduz a duração do lock global (este só precisa de ser adquirido quando não precisamos de mais locks de granularidade fina). A partir do momento em que temos locks individuais para todos estes jogadores, podemos fazer unlock da coleção e dar a oportunidade a outro thread de ir à mesma coleção procurar outros jogadores concorrentes.

Como reduzir o impacto nas SC?

- Objetos imutáveis (tipo ‘final’ → os valores não podem ser alterados);
- Locks de granularidade fina;
- 2PL (reduz as SC porque podemos atrasar/adiantar a aquisição/libertação, respetivamente, de locks)

Nota: 2PL é impotente em coleções de objetos, visto que os locks serão adquiridos por uma ordem fixa (evita deadlocks). A correspondência de locks a cada um dos dados nem sempre é clara!

10. Pitfall

Manter as variáveis e o lock correspondente encapsulados dentro do mesmo objeto. É aconselhável às vezes!

Nota: Variáveis marcadas com ‘static’ em Java são globais e precisam de controlo de simultaneidade (a não ser que sejam ‘final’ ou se a classe for usada por um único thread).

11. Locks em grande escala

A utilização de locks de leitura e escrita reduz a quantidade de tempo que se poderá estar à espera do lock, na medida em que vários leitores podem usar ao mesmo tempo, mas não reduz o nº de locks que temos que tratar quando temos coleções grandes de objetos. Um lock utiliza memória mesmo quando não está a ser usado. Para evitar o gasto desnecessário de memória vamos usar um gestor de locks, que dispensa os locks a pedido, ou seja, vamos indicar qual é o item que queremos aceder como parâmetro. O gestor de locks vai verificar, num mapa que guarda internamente, se já existe um lock atribuído a esse nome. Se não existir, cria um novo lock e acrescenta-o ao mapa e passa a executar a operação de lock nesse novo objeto.
Lock partillhado(shared): lock para leitores.
Lock exclusivo: lock para escritores. (Se tivermos um lock exclusivo adquirido não vamos puder aquirir mais nenhum lock naquele elemento.)

No gestor de locks fazemos uso de locks de granularidade múltipla. Esta abordagem é usada quando temos um grande nº de itens organizados de forma hierárquica (p.e. ficheiros).

Importante: Mais do que um leitor não é problema. Contudo, um escritor deve excluir todos os leitores e escritores.

- Dar preferência aos escritores: O escritor pode "morrer de fome".
- Dar preferência aos escritores: não permitimos mais leitores se um escritor estiver à espera. Há menor simultaneidade entre leitores.
- Justo e eficiente: leitores e escritores em ordem FIFO. Permitir que os leitores pulem (skip up) até k escritores na fila.

For a reader: while there is a writer wait…
For a writer: while there is anyone wait…

12. Locks de granularidade múltipla

1. **Locks de intenção:** utilizados em coleções de objetos. Entram em conflito com locks que não sejam de intenção.
2. **Locks** (como os conhecemos)

Desvantagem: são ineficientes.

13. Variáveis de condição

Em alguns casos, queremos que um thread espere por outro de forma explícita (p.e. à espera de um acontecimento). Contudo, esta espera é ineficiente se o thread em espera estiver ocupado a pesquisar a condição. Leva a corridas ou a deadlocks se o thread for suspenso pelo SO. As variáveis de cond. permitem que a espera de um thread por outro seja eficiente e correta.

Espera ativa: cada um dos threads que está à espera irá consumir tempo de cpu. Como o evitar? Primitiva que permita suspender um thread. ex.: pause() → permite que um thread pare até receber um sinal.

13.1. Problemas das variáveis de condição

1. **Deadlock:** se não libertarmos o lock antes de suspender o thread, os outros threads não vão conseguir entrar na SC.
2. **Corridas:** se libertamos o lock antes de suspender o thread há a possibilidade de o thread ficar suspenso e perder a oportunidade de ser acordado.

Solução: Tornar as operações de unlock e suspendMe numa operação atómica, em que qualquer aviso para acordar o thread só poderá vir depois do lock ter sido adquirido; e, se o lock já foi adquirido, então temos a certeza que o thread está suspenso e a notificação para acordar não se perde.

Primitivas para concretizar isto:

- **await():** desbloqueia atomicamente o objeto e suspende. Faz relock ao despertar. Dento do ciclo while.
- **Signal():** acorda apenas um. Mais eficiente!
- **SignalAll():** acorda todos. Desperdiça recursos! Porém, necessário para evitar deadlocks.
-

Como é que o ReentrantLock seleciona o próximo thread?

1. **Por omissão:** Não há preferências. Qualquer um.
2. **Justo:** O signal irá acordar o thread à espera há mais tempo (mas não garantido), mas o lock precisa de ser readquirido. Pode não ser assim tão justo, porque precisamos de garantir uma ordem de entrada nas SC, o que pode levar a ineficiência.

Condições	Gestores (Monitors)
Uma condição implícita para cada lock.	Várias var. de cond. para o mesmo lock. Evita o signalAll e o desperdício associado.
Threads esperam por uma cond. de ativação em qualquer ordem.	Threads esperam por uma condição obtida por um despertar de um ReentrantLock na ordem FIFO (mas não pode adquirir o lock na ordem FIFO!)

Problema: Uma vez que diferentes threads testam diferentes condições, antes de se bloquearem na var. de cond., pode acontecer que quando acordamos um deles estejamos a acordar aquele cuja var. de cond. não mudou de valor.

Solução: uma vez que no ReentrantLock temos a possibilidade de utilizar mais do que uma var. de cond. associada ao mesmo lock, conseguimos separar as coisas e acordar sempre o thread que interessa. Evitamos assim a utilização do signalAll com a utilização de várias var. de cond..

Será que a espera dos leitores e escritores para entrar na SC é justa? Não é justa para os escritores, porque existe a possibilidade de estes nunca entrarem na SC.

Solução: Dar prioridade aos escritores. Vamos permitir que um escritor possa bloquear a SC tanto para os escritores como para outros leitores. Contudo, este escritor não poderá aceder à SC! Deverá fazer uma 2ª espera, na qual espera que os leitores dentro da SC saiam. Não conseguimos evitar o uso do signalAll! Esta permissão será dada pela introdução de uma nova variável ‘waiting’ que corresponde ao tamanho da fila de espera dos escritores e é decrementada logo que um escritor consiga entrar. A modificação desta variável é feita de forma atómica.

14. Comunicação

Modelo OSI (em camadas):

1. Physical (1º nível): Descrição de sinais físicos/elétricos/radiações eletromagnéticas.
2. Data link: As redes de comunicação transmitem sequências de bits como informação. Nesta camada preocupamo-nos com erros de transmissão física.
3. Network: preocupa-se com a interligação entre participantes da comunicação. Encaminhamento e possibilidade de ter rotas alternativas para o mesmo destino.
4. Transporte: criamos a abstração de canais de comunicação fim a fim.
5. Session: estabelecimento de canais de comunicação entre os intervenientes numa aplicação.
6. Apresentação: formatos de dados trocados.
7. Aplicação: utilização da rede de comunicação.

Podemos resumir estas camadas em níveis de abstração:

1. Hardware: physical; data link.
2. SO: data link; network; transporte. (TCP/IP)
3. middleware;
4. application.

14.1 TCP/IP

Vai funcionar como um canal de comunicação entre 2 participantes e vai-se comportar como um bounded buffer. É uma ligação bi-dimensional (ambos os participantes podem enviar e receber dados). É fiável, porque garante a ordem dos dados enviados (FIFO). Em cada um dos sentidos, os dados recebidos são um prefixo dos dados que foram enviados. Esta conexão é identificado por:

- A local IP address;
- A local port;
- A remote IP adress;
- A remote port.

Se existir mais do que uma ligação com a mesma porta local, a porta remota será distinta, para que exista apenas uma ligação para cada combinação possível destes 4 elementos.
Devido ao NAT, os participantes podem não ver os endereços do outro extremo da ligação.

Vamos admitir que em cada um dos participantes existem 2 buffers (1 de emissão e 1 de recessão) dentro do SO.

Casos extremos do socket:

1. Um recetor tenta receber dados de um socket onde ainda não foi enviado nada. Encontra o buffer vazio e irá ficar bloqueado à espera que apareça alguma coisa.
2. A operação de envio irá bloquear, porque vão sendo enviados dados que não são lidos e que vão enchendo o buffer até que o emissor de envio será rejeitado porque não há espaço de armazenamento.

Nestas 2 situações, o socket comporta-se como um bounded buffer. Concluímos isso do TCP/IP e que este pode ser visto como uma primitiva de sincronização, uma vez que permite que processos ou threads em diferentes máquinas esperem uns pelos outros.

Nota: Se o emissor está a enviar dados mais rapidamente do que o recetor está a recebê-los, o emissor acaba por ser bloqueado. (vice-versa)

Funções do socket:

1. Interface para estabelecer uma nova ligação.
2. Interface para determinar uma ligação existente.
3. Interface para enviar e receber dados.

Como é que um participante descobre o endereço de outro?

1. Servidor: O endereço tem de ser bem conhecido dos clientes.
2. Cliente: não precisa de ser conhecido previamente. Será descoberto ao longo do estabelecimento da ligação.

Estabelecimento da ligação:
Indica ao servidor qual é o endereço do cliente com o qual irá comunicar. Este estabelecimento é um passo síncrono (exige 2 participantes ativos).

1. O servidor cria um socket.
2. Atribui-lhe um nome.
3. O socket é colocado em modo escuta, pronto a responder a clientes.

15. Serialização (ou Marshaling)

Consiste na conversão de estruturas de dados arbitrárias em arrays de bytes que podem ser depois transferidos através da rede e quando chegam ao seu destino são convertidos de volta em estruturas de dados idênticas com o mesmo conteúdo.

Motivações:

- Possibilita a abstração, ie, permite-nos considerar as mensagens como estruturas de dados genéricos e não como um alinhamento de diferentes bits e bytes.
- Simplifica aplicações distribuídas.
- Heterogeneidade dos SD: diferentes componentes têm diferentes características que obrigam a uma conversão de dados entre eles.

Vantagem: Permite que cada participante no SD tenha que ter apenas uma única versão do código de conversão.
Desvantagem: Podemos ter 2 conversões desnecessárias. Quando temos 2 máquinas a comunicar que tenham a mesma representação entre elas, mas cuja representação é diferente da representação escolhida para a rede.

Solução alternativa: Enviar os dados prefixados com um etiqueta que indica qual a representação a ser utilizada. Desta forma, o recetor saberá se precisa de converter ou não os dados.

Formatos usados na rede:

1. Texto: redundante; lento a descodificar. ex.: HTTP; JSon
2. Binário: mais composto e eficiente. Temos de nos preocupar com o alinhamento dos bytes na memória (porque acessos não alinhados → mais lento).

Como representar tipos de dados compostos?
Desafio: Estes dados não estarão organizados de forma contígua em memória (p.e. listas e árvores). Temos de os percorrer e enumerar os seus componentes. Inclui preenchimento opcional.
A transmissão de dados compostos fazer-se-à à custa da representação de cada um dos seus contribuintes de forma recursiva.

Grafos: A travessia simples não é suficiente com aliasing de ponteiro. Com ciclos, a travessia simples não termina e gera dados ilimitados.
Durante a serialização, devemos usar tags e um mapa auxiliar e acompanhar os objetos a restaurar ponteiros. Se o objeto já está repetido já não fazemos nova travessia. Caso contrário, acrescentamos ao mapa.

Como podemos escrever o próprio código de conversão?
Desafio: própria travessia recursiva das estruturas.

- 1ª alternativa: escrita manual de métodos chamados filtros, onde enumeramos os componentes de um objeto. Sujeito a erros.
- 2ª alternativa: Reflexão. Permite a substituição de dados transitórios (bloqueios, p.e). Escrita à priori da estrutura.

Design issue:
- **Programa primeiro:** formato de dados inferido de um programa existente. Ligado a um único idioma. Conveniente para desenvolvimento.
- **Dados primeiro:** programa gerado a partir de uma descrição abstrata de dados. Independência de idiomas e middleware. Força-o a desenvolver novas ferramentas.

Uma questão importante no desenho do mecanismo de serialização é a forma como eles lidam com versões diferentes da mesma estrutura de dados. Têm de ser capazes de comunicar entre si. Se utilizarmos a forma mais simples de travessia em que percorremos um conjunto de dados, quando a estrutura muda e o conteúdo das mensagens também, o recetor irá ter problemas, porque ao descodificar os dados num formato que não conhece irá receber dados corrompidos.
Como evitar isto? Deixar que a estrutura de dados seja modificada com novas versões do programa. Permitir o controlo da versão da estrutura de dados como um todo.
O 2º desafio é se ele utiliza uma abordagem 'streaming' (transmissão), em que a conversão é feita à medida em que vai ser feita a travessia, ou se temos um modelo de dados serializado que é criado em memória e pode ser consultado por uma ordem arbitrária.
No **1º caso**, o código que efetua a travessia irá fazer a travessia pela ordem que o código está.
Vantagens e desvantagens: Melhor eficiência, porque os dados são copiados apenas uma vez e não precisam de ser guardados integralmente em memória; Layouts internos e externos exatamente iguais; Dados copiados diretamente de/para representação externa.
No **2º caso**, temos de copiar os dados 2x (uma vez para o modelo intermédio e uma segunda vez para a representação final).

16. Processos e threads

Vejamos o seguinte exemplo: Temos um processo cliente em que um thread envia um pedido ao servidor através da escrita de um pedido no socket e bloqueia-se. Estamos perante uma comunicação assimétrica:
- O servidor passivo espera por solicitações.
- O cliente ativo emite solicitações e espera pelos resultados.

Esta ilusão de que existe um thread distribuído que atravessa as fronteiras entre processos depende de:
- Como usamos a ligação entre o cliente e o servidor.
- Como queremos que o servidor seja capaz de atender pedidos de diferentes clientes. Isto é, até que ponto queremos que o servidor seja multi-threaded?

16.1 Single-threaded

Temos um servidor apenas com um thread. Podemos processar um pedido de cada vez vindo do mesmo cliente. O servidor irá bloquear-se à espera de pedidos de um cliente e para cada pedido irá executá-lo e tendo terminado a execução de pedidos poderá atender pedidos sequencialmente de um cliente seguinte. Pode ser usado com meta-servers e connectionless(sockets UDP).

Nota: Pode ser usado multi-threaded se a execução do serviço por cada pedido for rápida e não bloquear à espera de outros recursos.

16.2. Thread-per-connection

Um thread por cada ligação. Sempre que o cliente estabelece uma ligação com um servidor, a sessão com esse cliente irá ser executada num thread em separado. Permitirá que pedidos de clientes diferentes possam ser processador concorrentemente.

16.3. Thread-per-request

Atribuir um thread a cada pedido, de forma a que vários pedidos que cheguem ao mesmo socket possam ser executados em simultâneo no servidor. É preciso que o servidor seja capaz de correlacionar os pedidos com as respostas. Isto poderá ser feito acrescentando a cada objeto pedido-resposta um identificador correspondente.
Problema: criação de um novo thread para cada pedido. Custo adicional.

16.4. Thread Pool

As solicitações são enfileiradas num buffer limitado. Reduz o custo adicional (ou sobrecarga de solicitações) reutilizando threads em diferentes pedidos. Fornece controlo de admissão para restringir a quantidade de recursos usados no servidor. As solicitações devem ser rotuladas com ids!

1º problema encontrado: Uma solicitação disparar respostas para vários clientes.
Solução: escrita diretamente em cada socket.

2º problema encontrado: Pode bloquear.
Solução: Ter um 2º thread e uma fila de respostas por cada conexão. Os threads vão se limitar a colocar a resposta na fila de espera e o 2º thread irá acordar e, sequencialmente, escrever cada uma das respostas dirigidas a esse cliente.

Concluindo, tudo aquilo que um thread local podia fazer também um thread distribuído poderá fazer. Garantimos assim o principal objetivo nos SD: transparência de acesso (deve ser indistinguível um acesso a um recurso local ou remoto).

Problema dos SD: Não podemos analisar caractere a caractere, visto que isso implicaria uma viagem de ida e volta ao servidor. (Latência acumulada)
Solução: Migração de código(ex.:SQL). O cliente não trabalha diretamente sobre os dados. É enviado um código para o servidor de forma a que esse código possa interagir com os dados sem que isso implique que os dados sejam enviados pela rede várias vezes. Evita latência de várias viagens de ida e volta para o servidor. Reduz o tempo de resposta.

2 questões importantes:

- Segurança: Restringir o código.
- Eficiência: Permite que o código seja executado + próximo do hardware.

Estas 2 questões são contraditórias, visto que para ter uma maior segurança queremos validar cada uma das operações que o código faz e, no entanto, se tivermos esse código a correr como código nativo vamos perder oportunidades para fazer essa validação.

Solução: Virtualização. Fazer uso de bibliotecas e do SO sobre o hardware. O byte code permite ultrapassar a heterogeneidade entre diferentes arquiteturas de processador.
Outra alternativa: utilização de virtualização ao nível do próprio sistema e não apenas ao nível da aplicação.

Cloud computing: Permite dar início a servidores em localizações precisas onde podem depois ser usados para correr componentes de aplicações distribuídas.
Edge computing: colocar servidores em posições estratégicas.
Fog computing: edge computing + cloud computing.

CDN(redes de distribuição de conteúdos): Alguém que publique conteúdos, em vez de servir diretamente esse conteúdo para os utilizadores finais, aquilo que faz é copiar uma versão inicial desse conteúdo para servidores que vivem na periferia da rede. Os clientes dentro destes 'service providers' ou destas instituições acedem depois à cópia local deste recurso.

17. Naming

A gestão dos endereços dos serviços/servidores pode tornar-se complicada. Para resolver este problema podemos usar um espaço de nomes plano (flat naming) e guardar esses nomes e o serviço de diretoria num servidor local. Desta forma, quando o cliente arranca pode perguntar ao serviço de diretoria qual é o endereço associado a um nome. ex.: JMI
Problema: temos de conhecer o endereço do servidor diretoria.
Alternativa: utilização de primitivas de difusão de rede para contactar diretamente os serviços/servidores alvo. Cada um dos servidores estará à escuta na rede de perguntas da diretoria e quando queremos contactar um desses servidores podemos enviar para toda a rede, difundindo, uma pergunta "quem é que, neste momento, tem um endereço de um determinado serviço X?". O servidor que tiver poderá responder diretamente. ex.: ARP.

Ambas estas alternativas têm o problema de dificilmente puderem ser utilizadas em grande escala. No caso de utilizador um servidor único teríamos um único ponto de falha. Por outro lado, ao estarmos a utilizar difusão estamos a dirigir todas as interrogações a todos os servidores.
Outra limitação é não permitirem facilmente que a autoridade administrativa seja distribuída, ie, que diferentes partes do espaço de nomes sejam geridos por diferentes entidades.

Solução: Serviço de nomes hierárquico. Deixamos de ter um único ponto de falha, porque diferentes servidores vão ser responsáveis por diferentes espaços de nomes. Permitimos também que exista uma distribuição da autoridade administrativa. Em diferentes níveis deste espaço podemos ter uma administração diferente. Na raiz é + centralizada e + estruturada. Conforme vamos caminhando para as folhas, + fácil será modificar e gerir o espaço de nomes. Tem a vantagem de puder ser consultado de forma iterativa como recursiva.
Esta alternativa é mais eficiente do que o flat naming porque divide a carga por uma hierarquia. Contudo, continua a existir um único ponto de falha na raiz, mas este ponto de falha é aliviado porque é possível fazer caches dos resultados das perguntas para os níveis anteriores.

Alternativa: Tabela de hashmap distribuída. DHT.
Como funciona?
1º passo: Fazer uma correspondência para um espaço númerico de dimensão variável dos obejtos que queremos guardar.
2º passo: Guardar os objetos que correspondem a um determinado nº numa posição.

Como é que conseguimos organizar estas ligações entre servidores de maneira a que esta travessia seja fácil?
Exemplo do anel DHT: Dobrar o espaço numérico em anel. Utilizar a função hash para atribuir um nº a cada objeto, mas também atribuir um destes números a cada servidor. Cada nó guardará na tabela de dados apontadores para os nós que estão imediatamente a seguir (à distância do seu próprio número + 2 (elevado a i-1)). Um determinado nó será responsável por todo o espaço de números que o antecedem.

Vantagens:
- Não existe uma única raiz.
- Resistente e eficiente.
- Qualquer pesquisa efetuada a partir de qualquer nó conseguirá num número logarítmico de saltos chegar ao destino pretendido.
  ex.: BitTorrent.

Transparência de localização: o utilizador não precisa de saber qual é a localização de um serviço para o puder utilizar. Naming is key here! Um ‘naming service’ com referências de objeto oculta a localização.
Transparência de relocalização: os clientes de um determinado serviço não se apercebem quando esse serviço muda de sítio. São necessárias atualizações dinâmicas aqui.

18. Invocação remota

Serve para esconder as interações entre clientes e servidores na invocação de procedimentos/métodos remotos. Junta todos os conceitos discutidos até agora (naming; serialização; sockets).

Geração de código: O stub e o código são determinados mecanicamente pelo protocolo de interface. Pode ser gerado a partir de uma descrição
Code first: escrever código primeiro; gera stubs usando reflexão.
Protocol first: escrever a definição de interface abstrata; gera stubs com um compilador.

Passagem de parâmetro: Quando temos um sistema de invocação remota, os parâmetros são copiados do cliente para o servidor. Podem ser copiados de volta.

Manipulação de erros: Problemas de conexão ou servidor não disponível não podem ser escondidos. Semântica possível:
- At most one: tenta uma vez e lança uma exceção.
- At least one: tenta várias vezes até ser reconhecida. Pode bloquear para sempre. Válido apenas para operações idempotentes (operações cuja execução mais do que uma vez não faça erros).

Método da invocação remota:
Cada objeto possui um identificador.

- No cliente: crie um esboço para cada objeto. Prefixar cada solicitação com o identificador.
- No servidor: crie um wrapper que descodifica solicitações para cada objeto. Mantenha um mapa de identificadores (esqueleto). Procura o objeto em cada solicitação (lookup).

Nota: Middleware de invocação remota oculta a distribuição de serviços (transparência de acesso).

As informações necessárias para criar um stub são: classe de objeto; endereço do servidor; identificador.
Quando um stub é passado como parâmetro ou retornado por um método: enviar referência.
Quando uma referência é recebida: Recrie um stub.
Permite que qualquer objeto possa chamar outro objeto qualquer.

Os objetos pode ser retornados por referência ou valor:

Referência	Valor
É enviada info. que permita reconstituir um stub que aceda aquele objeto do outro lado do sistema. ex.: Remotos.	É criado uma cópia do objeto integral do outro lado do SD. ex.: serializáveis.
Estado imutável	Estado compartilhado
Multiple round trips	Data copied

19. gRPC e Java RMI

gRPC: Protocolo primeiro com linguagem de interface Protobuf. Vários idiomas de destino, pouco esforço em total transparência para qualquer um deles. Parâmetros de entrada única e saída única.

Java RMI: Code first. Preveligia-se a transparência. Usa reflexão para geração de stub. ‘Naming service’ incluída. Interoperabilidade limitada com outras linguagens.

Nota: Um SD geralmente é definido como uma coleção de elementos de computação autônoma, mas resultando num único sistema coerente. É necessário que cada elemento se coordene.

20. Sincronização de relógios

É necessária na medida em que os relógios de diferentes dispositivos podem ter começado em sítios diferentes, ie, podem ter sido acertados manualmente de forma diferente.
Exemplo: A → relógio de referência. B → processo que irá ajustar o relógio.
O B consultará o A através de uma leitura instantânea e ajusta esse relógio com a diferença observada. Temos 2 situações possíveis:

1. A está atrás de B. Quando fazemos tA-tB obtemos um valor negativo. Se atrasarmos essa qt de tempo vamos ter que o instante em que o relógio foi atrasado vai acontecer 2x. Erro!
2. A está à frente de B. Vai fazer com que os processos que estejam nesse dispositivo a medir a passagem de tempo terminem mais cedo.

Solução: Ajustar progressivamente os relógios com pequenos incrementos ao longo do tempo. Esta solução pressupõe que conseguimos fazer uma observação instantânea do relógio num outro dispositivo. Contudo, isto não se verifica num SD. Só conseguimos obter info. de outro participante através da passagem de mensagens.

Problema: Esta troca de mensagens demora tempo e podem haver atrasos imprevisíveis na transmissão.
Solução: Tentar medir o tempo no instante do envio e da resposta e descontar o atraso.
Problema: Qual é o atraso da mensagem?
Soluções:

- **NTP**: Faz um estimativa assumindo que os atrasos são iguais. Como é que escolhe a melhor estimativa? Repete o processo e escolhe o que demorou menos tempo total (ida e volta). ((T2-T1)+(T4-T3))/2
- **RBS**: Assume um meio de transmissão verdadeiro (físico). Um participante envia um mensagem através do canal de difusão e todos os participantes vão assinalar qual foi o tempo em que receberam essa mensagem. Cada um deles ficará à espera que o relógio de referência envie o valor que foi lido a todo os outros.

Exemplo do **exposure notification**: Notificar quem esteve perto de mim nos últimos N dias. Evitar compartilhar e armazenar info. pessoais.
Solução: usa o Bluetooth para difundir ids aleatórios que são gerados para os dispositivos na proximidade. Neste caso, a sincronização de relógios é fundamental como forma de coordenação entre os participantes.

21. Exclusão mútua distribuída

Iremos considerar o balanceamento de carga (se as mensagens trocadas são com 1 ou mais elementos do grupo) e ainda o nº de saltos de mensagens para entrar na SC.
1º solução: Fila centralizada mantida por um coordenador. Os clientes que quiserem entrar terão de fazer uma invocação remota ao coordenador.
Problema: carga assimétrica.
2º solução: organização dos processos num anel.
Regra imposta: cada processo só pode usar 1x o testemunho e deve passá-lo ao seguinte.
Vantagem: carga simétrica(todos os participantes com o mesmo nº de mensagens enviadas/ recebidas).
Desvantagem: n/2 saltos para entrar. Obrigada a enviar mensagens mesmo que nenhum participante esteja de momento a entrar na SC.

Concluindo, um algoritmo distribuído é difícil de obter. Como solicitações de bloqueio simultâneas são recebidas por destinos diferentes em ordens diferentes, a segurança não é garantida. Tirando partido da sincronização de relógios, vamos considerar apenas mensagens até t-S, sendo que S>(atraso+inclinação).

22. Causalidade e relógios lógicos

- clock(i): hora em que i aconteceu.
- Se i precede j, então clock(i)<clock(j).

Relógios lógicos de Lamport:
- Eventos locais: contador de incremento.
- Enviar eventos: incrementar e depois marcar com contador.
- Receber eventos: atualize o contador local para o máximo e incremente.

23. Difusão fiável

Confiabilidade significa que todos os destinos entregam todas as mensagens enviadas.
Como garantir isso? Armazenar num buffer e retransmitir até ser reconhecido. Contudo, não é escalonável para um grande nº de destinos devido ao “ack implosion”.

Multicast: $O(n^2)$ ack implosion. O nº de mensagens passa a ser proporcional ao quadrado do nº de processos, porque cada um dos n terá que enviar n mensagens para os outros processos participantes.

Gossip:
Protocolo para multicast de uma mensagem:
- Selecione um subconjunto de alvos aleatórios.
- Encaminhar mensagens apenas para esses alvos.
- Descartar a mensagem.

Isto evita enviar diretamente a mensagem toda, enviando apenas um anúncio de que a mensagem está disponível.

23. Transações distribuídas

Problema da coordenação: confirmação transacional distribuída.
Uma transação é uma sequência de operações que deve ser executada de forma atómica (indivisível): ou todas as operações são executadas ou nenhuma o é.
Limitações:
- 2PC: Quando existe uma falha no coordenador, os participantes poderão ficar bloqueados à espera da resposta do coordenador. Ou seja, é limitado à recuperação da falha do coordenador. É usado em middleware corporativo para aplicativos de integração.

Primitivas extras:
- accept() → bloqueia o thread que invocamos até que um cliente se tenha ligado.
- flush() → para garantir que os dados estão mesmo escritos.