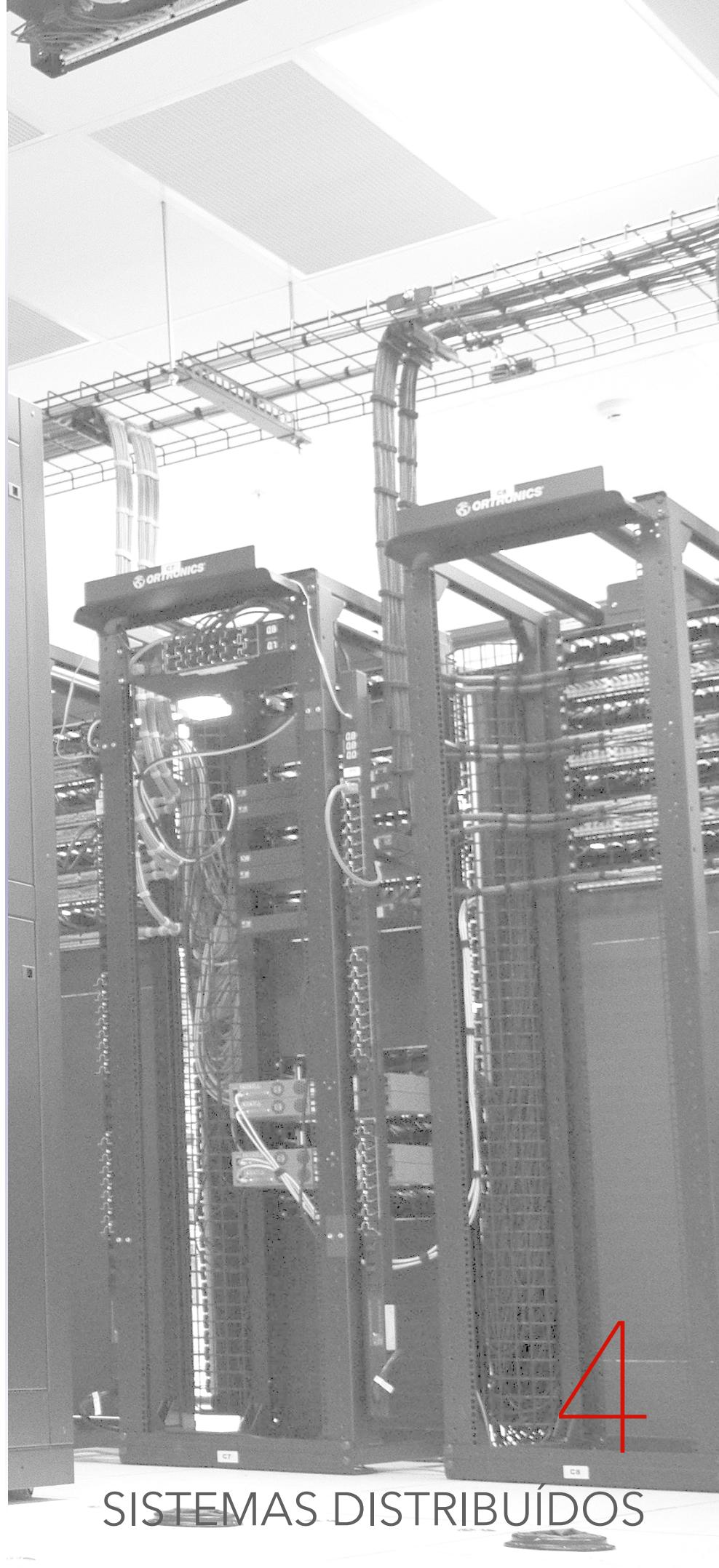


A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie Lamport



universidade de aveiro
teoria poesisis praxis



Atenção!

Todo o conteúdo deste documento pode conter alguns erros de sintaxe, científicos, entre outros... **Não estudes apenas a partir desta fonte.** Este documento apenas serve de apoio à leitura de outros livros, tendo nele contido todo o programa da disciplina de Sistemas Distribuídos, tal como foi lecionada, no ano letivo de 2017/2018, na Universidade de Aveiro. Este documento foi realizado por Rui Lopes.

mais informações em ruieduardofalopes.wix.com/apontamentos

Em Sistemas de Operação (a3s1) estudámos que para que um sistema computacional possa funcionar tem de existir um componente de dimensões relativamente grandes capaz de fazer a interface entre o *hardware* e o *software*. Nesta mesma disciplina também vimos que os sistemas podem, entretanto, funcionar de forma distribuída, isto é, de uma forma em que vários nós ajam em colaboração uns com os outros. Tal conceito é objeto de estudo na disciplina de Sistemas Distribuídos (a4s2).

1. Noções Básicas de Sistemas Distribuídos

Antes de avançarmos para um estudo mais aprofundado acerca de como é que os sistemas distribuídos funcionam é importante, primeiro revisitar parte das nossas abordagens de Sistemas de Operação (a3s1).

Processos e fios de execução (threads)

Um dos conceitos que nos serão mais importantes em toda esta disciplina será o de processo e o de *thread* (em português, fio de execução) — o que significa cada um destes conceitos e para que é que servem?

Quando executamos alguma tarefa num computador esta ação é realizada dentro do contexto de um programa, isto é, dentro de uma região que já fora delimitada para uso próprio de um conjunto de instruções que serão tratadas pelo processador. Esta execução, para posterior identificação e para que possa ter mecanismos de manutenção própria, necessita que haja uma entidade que se responsabilize por ela — a essa entidade damos o nome de **processo**. Em termos latos, dizemos que uma execução de um programa corresponde a um processo.

processo

Um processo é então caracterizado por um conjunto de propriedades, entre as quais um espaço de endereçamento (onde se localiza o código e os valores atuais de todas as suas variáveis a si associadas), um contexto do processador (onde se descrevem todos os valores atuais os registos internos do processador), um contexto de entradas/saídas (onde se descrevem todos os dados que se encontram a ser trocados entre os dispositivos de entrada e de saída) e o estado da execução de um programa.

paralelo

Este processo poderá correr em **paralelo** com outros, dizendo-se assim que é possível a criação de um ambiente multiprogramável. Contudo, na verdade, tal ambiente é uma imagem aparente da execução de vários programas em simultâneo, uma vez que apenas há uma priorização dos vários processos ao longo de execuções, aquando da utilização do processador. Para simplificar, podemos pensar neste problema da seguinte forma: se pensarmos que temos um conjunto de vários processadores virtuais onde estamos a executar vários processos em simultâneo, então é possível atingir o objetivo de programação em simultâneo, sendo que cada processo possui um destes processadores somente para si. Podemos então considerar, com este modelo, que a execução de um processo não será afetada pelo instante de tempo e pelo local onde a comutação de código acontece, e que nenhuma restrição será aplicada pelo tempo de execução total ou parcial do mesmo.

Temos então que a comutação de contexto de execução é simulada pela ativação (ou desativação) dos processadores virtuais sobre os quais estão os processos a ser executados, com controlo dos seus próprios estados. Isto, num ambiente monoprocessador, acontecerá com o número de processadores virtuais ativos igual a 1 (no seu máximo de capacidade). Por outro lado, num processador multi-núcleo (*multi-core*), o número de processadores virtuais ativos em qualquer instante de tempo é igual ao número de processadores no seu núcleo (no seu máximo de capacidade).

estados

Um processo poderá, então estar em diferentes situações denominadas de **estados**, ao longo da sua existência. Os estados mais importantes são os seguintes: *em execução*, quando o processo domina o processador; *pronto para execução*, quando este espera que o processador lhe seja atribuído para prosseguir a execução; e *bloqueado* quando não lhe é permitido o uso do processador porque um outro processo está a usá-lo (como um acesso a um recurso, o término de uma operação de entrada/saída, entre outros...).

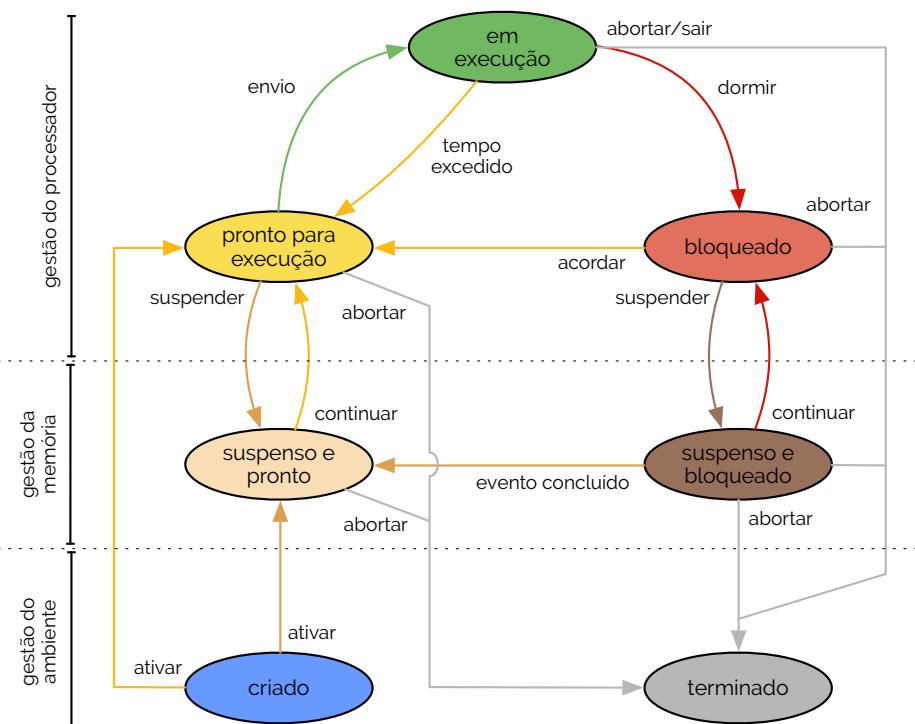


figura 1.1

Na Figura 1.1 podemos ver uma versão mais completa dos vários estados possíveis de um processo. As suas transições de estado são habitualmente realizadas por uma fonte externa ou pelo sistema de operação, podendo também ser ativadas por si próprio. A parte do sistema operativo que manipula os processos é denominado de **escalonador** (em inglês *scheduler*), que é parte integrante do *kernel*, elemento responsável pela manipulação e tratamento de exceções, tal como pelo escalonamento da atribuição do processador e de todos os outros recursos a processos.

escalonador

O conceito de processo contém, então, duas grandes unidades de referência: gera a pertença de recursos (cria um espaço de endereçamento privado, assim como um conjunto privado de canais de comunicação com os dispositivos de entrada e de saída) e gera um conjunto de fios de execução, isto é, possui um contador de programa (em inglês *program counter*) que aponta para a próxima instrução a ser executada, assim como um conjunto de registos internos que contêm os valores atuais de variáveis a serem executadas e uma pilha que mantém o histórico de execução (um *frame* para cada rotina que é invocada e que ainda não tem retorno). Estas propriedades, embora tidas em simultâneo num processo, podem ser tratadas em separado pelo ambiente de execução. Quando isto acontece, os processos são então vistos como um grupo de um conjunto de recursos e **threads** (em português fios de execução), também denominados de *lightweight process* (em português, processos leves), que representam entidades independentes e de execução, dentro do contexto de um só processo.

threads

Para melhor percebermos esta diferença de conceitos consideremos o seguinte cenário, em que estamos perante uma orquestra de guitarras clássicas. Cada guitarrista possui uma pauta para tocar uma música e, uma vez que estamos a considerar guitarras clássicas, cada guitarra possui 6 cordas simples. Neste caso, podemos olhar para o músico como o nosso processo, sendo o seu conjunto de recursos as várias cordas da sua guitarra, onde pretende ler (ouvir) e escrever (tocar) valores (notas musicais) e, as várias *threads*, como sendo as várias execuções realizadas sobre as cordas, por vias dos seus dedos. O guitarrista é que coordena as várias execuções sobre as cordas, não permitindo que os seus recursos sejam mal usados. Caso se queira identificar uma guitarra em particular na orquestra, também se pode referir ao guitarrista, da mesma forma que num sistema operativo identificamos um processo.

Quando temos múltiplos fios de execução dizemos que o ambiente é **multithreaded**. Na Figura 1.2 podemos ver, num pequeno diagrama, quais as diferenças, em suma, entre ambientes *single-threaded* e *multithreaded*.

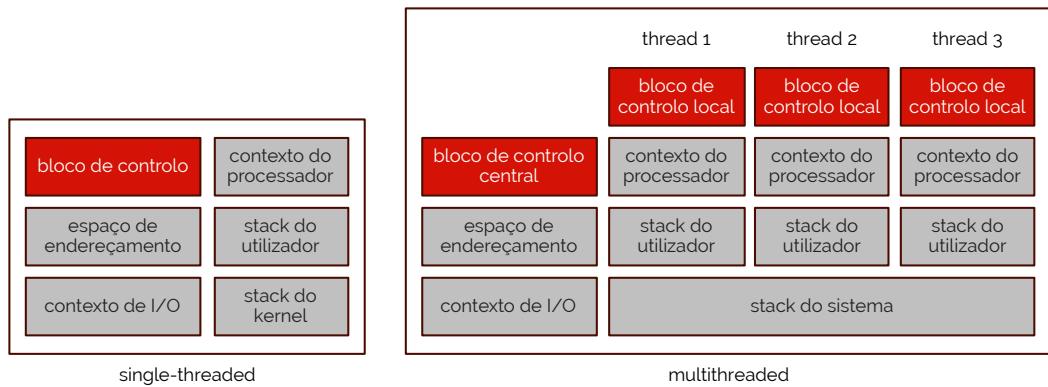


figura 1.2

Perante sistemas como os *single-threaded*, as vantagens de um *multithreaded* são as seguintes: maior simplicidade nas decomposições de soluções e aumento da modularidade nas suas implementações — os programas que envolvem múltiplas atividades e pedidos de serviço múltiplo são mais fáceis de desenhar e implementar numa perspetiva de concorrência, que numa sequencial; maior gestão de recursos de sistemas computacionais — a partilha de espaço de endereçamento e do contexto de I/O entre as várias *threads* permite uma redução da complexidade de gestão da ocupação da memória principal e acesso a dispositivos de I/O; maior eficiência e velocidade de execução — uma decomposição de uma solução com base em *threads* (por oposição à com base em vários processos) requer menos recursos do sistema operativo, permitindo que operações como a criação de processos e o seu respetivo término (e contexto de comunicação) se tornem menos exigentes, por conseguinte, mais eficientes.

Organização de um programa multithreaded

Num programa *multithreaded*, geralmente, existe uma organização própria que permite uma implementação de soluções a problemas de uma forma uniforme e constante com uma metodologia correta.

Num projeto que se tenha a necessidade de (ou em que seja mais fácil) implementar o paradigma *multithreaded* é importante saber segmentar as várias partes dos programas de uma forma clara e concisa. Para responder a esse problema criamos *threads* de modo a que cada uma fique associada com a execução de uma função ou procedimento que implementa uma atividade específica.

Através da modularidade aplicada pela contextualização de cada *thread* em cada função ou procedimento, olhando para a estrutura de dados em si, é necessário que esta forme um espaço de partilha de dados, definida em termos de variáveis e de canais de comunicação abertos com os vários dispositivos de entrada e saída. Tais recursos deverão estar, então, abertos para as várias *threads* que coexistem num determinado instante de tempo, tanto para processos de escrita, como de leitura.

Havendo cumprida a implementação desta segmentação e metodologia de programação, é importante que o programa *main*, aqui representado por uma função ou procedimento, constitua a primeira *thread* a ser criado e, tipicamente, a última a ser terminada.

Implementação de threads em Java

Embora não seja uma linguagem única que pode resolver este tipo de problemas, a linguagem Java pode ser uma possível escolha para a implementação de *threads*. Ao longo deste documento iremos então abordar vários exemplos de aplicação criados em Java, dado que a sua verbosidade atende muito facilmente à precisão e detalhe das nossas soluções.

5 SISTEMAS DISTRIBUÍDOS

Para quem possa não conhecer o ambiente de desenvolvimento Java, este é baseado num contexto de uma máquina virtual, denominada de **JVM** (acrônimo para *Java Virtual Machine*). Esta máquina virtual constitui assim o ambiente próprio e exclusivo de execução para aplicações criadas sobre Java, correndo sobre o sistema operativo com uma ligação muito específica.

JVM

As bibliotecas da linguagem Java possuem elementos que permitem configurar a forma como a máquina JVM é executada sobre o sistema operativo, descrevendo tópicos como a memória reservada para esta, o número de processadores, o tamanho que a máquina ocupa no sistema computacional, entre outros... Mais, as várias execuções de programas sobre a máquina Java geram processos próprios, o que acontece uma vez que cada programa é criado, pelo que é executada a classe `Process` permitindo assim que haja uma estrutura com a mesma capacidade de criação, cópia, espera e morte, tal como um processo convencional do sistema operativo onde este é criado.

Sendo o Java uma linguagem de programação concorrente, as *threads* estão implementadas pela linguagem propriamente. Conceptualmente, a criação de uma *thread*, tal como vimos em Sistemas de Operação (a3s1) pressupõe dois acontecimentos: primeiro, que haja um objeto que representa uma *thread* autónoma de execução; segundo, um tipo de dados não-referenciável (estático) ou com um objeto instanciado sobre este, que define o método executado sobre a *thread*, por conseguinte o seu ciclo de vida. A JVM define assim que: cada programa executável consiste, no mínimo, de uma *thread* que é implicitamente criada quando a máquina virtual (depois de se inicializar o ambiente de execução) chama o método `main` no tipo de dados inicial; as restantes *threads* são explicitamente criadas pela *thread main* ou por qualquer outra *thread* criada com sucesso da *thread main*; o programa termina quando todas as *threads* criadas terminarem com sucesso o seu método de descrição associado.

A biblioteca base da linguagem Java, como tal, possui dois tipos de dados para a criação das *threads*, sendo elas usando o construtor-interface proveniente de `Runnable` (como podemos ver no Código 1.1) e, outro, usando um simples construtor de classe (como podemos ver no Código 1.1).

```
public interface Runnable {  
    public void run();  
}  
  
public class ThreadClass {  
    // alguns métodos implementados...  
  
    public void run();  
    public void start();  
  
    // outros métodos possivelmente implementados...  
}
```

código 1.1

Cada *thread* autónoma de execução é uma instanciação do tipo de dados referência `Thread`, que define, como podemos ver no Código 1.1, dois métodos que são operacionalmente relevantes neste contexto: a função `run()`, que é chamada quando a *thread* é colocada em execução e que representa o seu ciclo de vida; a função `start()`, que permite executar (iniciar) a *thread*. Contudo, note-se que não é estritamente necessário criar um novo tipo de dados referência derivado de `Thread`, fazendo *override* o método `run()`, para garantir a execução de tarefas em específico. O mesmo objetivo poderá ser atingido alternativamente através da criação de um tipo de dados referência independente que implementa a interface `Runnable`, da forma como podemos ver no Código 1.1, o qual, por conseguinte, define o contexto da função `run()`.

Podemos então ter uma primeira forma de criar uma *thread* através da criação de um novo tipo de dados referência que extende `Thread`, como podemos ver no Código 1.2. Aqui necessitamos, depois, no programa `main` (ou onde queremos chamar/executar a nova *thread*), de instanciar o objeto que possui a função que define a *thread*, colocando-a em execução logo de seguida.

```

public class ThreadClass extends Thread {
    // alguns métodos implementados...

    public void run() {
        // descrição da função da thread a ser criada e executada
    }

    // outros métodos possivelmente implementados
}

public class ClassWhereWeWantToCreateThread {
    public static void main(String[] args) {
        // algum código de implementação...

        ThreadClass thread = new ThreadClass();
        thread.start();

        // outras implementações da main...
    }
}

```

código 1.2

Numa segunda alternativa, como referido e representado no Código 1.3, temos a implementação do método `run()` na implementação da classe `Runnable` no tipo de dados criado para o nosso projeto.

```

public class ThreadClass implements Runnable {
    // alguns métodos implementados...

    public void run() {
        // descrição da função da thread a ser criada e executada
    }

    // outros métodos possivelmente implementados
}

public class ClassWhereWeWantToCreateThread {
    public static void main(String[] args) {
        // algum código de implementação...

        Thread thread = new Thread(new ThreadClass());
        thread.start();

        // outras implementações da main...
    }
}

```

código 1.3

Uma pequena nota em relação à implementação de *threads* em Java está no uso de uma palavra reservada denominada de **volatile**. Tal palavra modifica as características de uma variável definida no corpo da função da *thread*, informando a máquina Java e o seu compilador de que, durante a sua execução, estas deverão verificar consistência no valor de tal variável. Por consistência pretendemos indicar que o acesso à tal variável deverá ser sempre feito da forma como o código em si foi desenhado, não fugindo a qualquer tipo de protocolo ou razão protocolar.

volatile

Esta informação é crucial uma vez que o modelo de memória da máquina Java permite ao compilador, aquando da geração do *bytecode* de um determinado tipo de dados, e à própria máquina, aquando da interpretação do *bytecode*, desempenhar otimização de código que, sendo totalmente consistente num meio *single-threaded*, poderá produzir uma execução paradoxal em ambiente *multithreaded*.

Um programa *multithreaded* em Java deverá sempre terminar quando as suas partes constituintes terminarem (as suas *threads*). Para executar este tipo de funções a biblioteca principal do Java apresenta-nos duas alternativas: a execução do método `System.exit(int)`, onde o seu argumento não é nada mais do que o código de erro a apresentar e seguir com o retorno da execução, sendo que aqui a aplicação é forçosamente terminada; a adaptação das *threads* instanciadas direta ou indiretamente em demónios (em inglês *daemons*), sendo que a máquina Java termina a sua execução somente após de todas as *threads* apresentarem este estado.

Não obstante, o método mais usado e convencional para o término de uma aplicação *multithreaded* é realizado tornando a primeira *thread* à espera do fim de todas as suas outras, que foram criadas através de si. Para tal, apenas temos de executar a função `join()`, que bloqueia a *thread* invocadora até que a *thread* referenciada termine a sua execução. [1]

Introdução ao conceito de sistemas distribuídos

Portanto, até ao momento já descrevemos que um sistema computacional possibilita um utilizador a execução de múltiplos programas em simultâneo (ou pelo menos, aparentemente). Vimos também que, dentro destes programas, vários poderão estar a trabalhar em colaboração uns com os outros, sendo que cada instância de um programa é identificado e controlado por um único processo com um conjunto de fios de execução a si associados, que partilham um único espaço de endereçamento.

A mesma lógica de segmentação que temos entre vários programas dentro de um mesmo sistema computacional é possível de ser implementado num conjunto de máquinas, todas estas ligadas em rede umas com as outras. A este tipo de topologia e arquitetura de desenvolvimento de programas damos o nome de **sistemas distribuídos**.

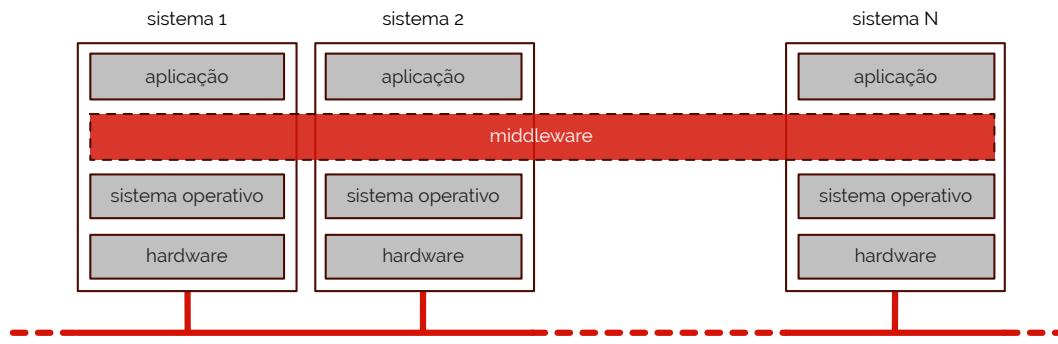
De forma mais correta e formal, temos então que um sistema distribuído é tal sistema que possui sistemas operativos com componentes localizados em diferentes nós de processamento num sistema computacional paralelo, comunicando e coordenando as suas ações através de trocas de mensagens entre eles [2]. A grande motivação para este tipo de construção é, essencialmente, a partilha de recursos, podendo otimizar os níveis de eficiência de execução de um projeto. Aqui, recurso, deverá ser entendido como uma entidade abstrata que incorpora algo material (processadores, infraestruturas de rede, dispositivos de armazenamento ou outros periféricos mais-ou-menos especializados) ou imaterial (como informação obtida de alguma fonte ou outras funcionalidades que são operadas sobre estas). Esta partilha é traduzida de duas formas possíveis: através de uma paralelização de aplicação, isto é, tirando vantagem da existência de múltiplos processadores e de outros componentes de um sistema computacional paralelo para obter execuções dos nossos projetos mais eficientes; ou através da disponibilidade do serviço prestado, uma vez que estando o nosso serviço alojado e fornecido em mais do que uma máquina, é possível que o projeto seja fornecido de uma forma redundante (havendo uma única interface uniformizada para que o serviço possa ser pedido sem quaisquer problemas).

Basicamente, considerando um conjunto de máquinas todas ligadas em rede cooperando entre elas num sistema distribuído, temos que os vários sistemas possuem uma camada lógica em comum, à que denominaremos de **middleware**. Na Figura 1.3 podemos ver uma representação desta mesma camada lógica.

sistemas distribuídos

middleware

figura 1.3



A forma como os sistemas distribuídos são organizados é estritamente dependente do tipo de problemas para os quais necessitam solução. Na prática, podemos dizer que existem quatro tipos de sistemas distribuídos: **sistemas em cluster**, que consistem num conjunto de sistemas computacionais interligados em busca de uma melhor paralelização da execução de aplicações (aqui tipicamente os vários computadores são semelhantes, todos correndo o mesmo sistema operativo e partilhando a mesma rede, com um único nó central que manipula a alocação dos nós de processamento para um programa em específico, com uma fila de trabalhos submetidos e interfaces com os utilizadores do sistema); **sistema em grelha**, onde não há considerações feitas sobre os *hardwares* dos nós que constituem o sistema — nem os seus sistemas operativos —, numa tentativa de juntar todos os recursos de

sistemas em cluster

sistemas em grelha

diferentes organizações, de forma a que várias pessoas possam colaborar entre elas; por fim, **sistemas de transação** de informação, em que os seus principais objetivos são a manipulação de dados sobre bases de dados, através de transações.

sistemas de transação

O desenho de sistemas distribuídos, assim como das aplicações desenvolvidas nestes, requerem algumas preocupações cuja importância é crucial para que haja um excelente desempenho do sistema. Alguns exemplos destes itens sobre os quais devemos tecer alguma preocupação são a possível heterogeneidade dos componentes que formam o sistema computacional paralelo, a sua transparência, o seu grau de abertura, a segurança da informação trocada entre as várias máquinas, o seu potencial de escalabilidade, a manipulação de estados em caso de falha e o seu potencial de concorrência entre máquinas.

heterogeneidade

Olhando para o caso da **heterogeneidade** temos que esta se torna visível nos sistemas distribuídos aquando da interligação entre várias redes de computadores diferentes, do uso de diferentes *hardwares* ou sistemas operativos nos vários nós de processamento em simultâneo, do uso de diferentes linguagens de programação numa mesma aplicação (mas em componentes diferentes de máquina em máquina) ou na interligação de funcionalidades dadas por entidades externas. Para corrigir este tipo de problemas deverá haver uma partilha das várias especificações e implementações de uma arquitetura de rede, uma conversão da representação de dados apresentada por cada processador para um tipo único e uniformizado para que todos os nós possam perceber o seu contexto, uma harmonização das interfaces de linguagens de programação oferecida pelos vários sistemas de operação e um ajuste de estruturas de dados implementadas por várias entidades externas para que possa coexistir um formato comum.

transparência

Em termos de **transparência** temos que esta poderá ser atribuída como sendo uma capacidade que exprime o sucesso ou insucesso em mascarar o grau de complexidade do sistema a ser implementado. Assim, tornando o processo de criar um sistema distribuído transparente significa que as descrições das suas funcionalidades deverão ser feitas de uma forma única, conceptualmente simples, ao invés de uma partilha conjunta de informações acerca das várias interfaces sobre as quais o sistema está assente. O objetivo é, então, esconder todos os recursos que não são diretamente relevantes às tarefas-alvo do programa, tornando-os anónimos tanto para o utilizador, como para o programador da aplicação. Este grau de transparência sobre os quais os recursos devem ser acedidos e operados comportam logo o grau de abstração e operacionalidade da camada lógica de *middleware*.

abertura

O critério de **abertura** é uma capacidade tal que se determina a boa ou má capacidade de extensão de funcionalidades ou das suas possíveis reimplementações. Para o caso de sistemas distribuídos, este critério está fundamentalmente relacionado com a capacidade de incorporação de novos serviços e de os tornar disponíveis a uma vasta gama de aplicações. Para tal é necessário estabelecer um mecanismo de comunicação uniforme para o acesso de recursos partilhados, publicar os APIs principais e garantir uma conformidade estrita, tanto a nível de desenho como de implementação, de cada novo componente, com a norma relevante.

segurança

Em termos de **segurança** é importante que um sistema distribuído incorpore capacidades como as de confidencialidade (proteção contra a partilha de dados com entidades não autorizadas), de integridade (proteção contra a modificação de dados indevida) e de disponibilidade (proteção contra interferências aos canais de acesso à informação do sistema). Para corrigir este tipo de problemas detalhes como a introdução de *firewalls*, cifra de mensagens ou o uso de assinaturas digitais podem ser algumas das possíveis soluções. Contudo, alguns ataques são possíveis de ainda serem executados como os ataques de rejeição de serviço distribuído (*denial-of-service attacks*) — este tipo de ataques acontece quando os fornecedores de serviço são entupidos com quantidades muito elevadas de pedidos falsos, ficando sem forma de responder aos mesmos ou a outros.

escalabilidade

Um outro detalhe a ter em conta é a **escalabilidade** de um sistema. Sendo que estamos a tratar de um sistema distribuído é importante que este seja possível de ser aumentado em termos de número de máquinas ou de recursos partilhados. Para que isso possa acontecer é importante que a mesma rede de computadores possa sofrer detalhes de escalabili-

dade, tanto em número como em aumento da qualidade dos canais de comunicação, de forma a que estes possam começar a receber um número de pedidos maior, assim como de respostas. Isto deverá ser feito para que problemas como os de **bottlenecks** possam ser evitados em sistemas de fornecimento de serviços.

bottlenecks

Como também vimos em Arquitetura de Computadores Avançadas (a4s1) um sistema computacional, quer ele seja único ou distribuído, é importante que tenha mecanismos de **recuperação de falhas**, sejam estas graves ou ligeiras. Para isso, uma possível solução poderá ser a implementação de mecanismos de redundância na rede e dentro de cada um dos sistemas computacionais que estão envolvidos na distribuição dos serviços. Fora a recuperação, um sistema também deverá ser tolerante às várias falhas que várias entidades, sejam elas ativas ou passivas, possam criar em plena execução de um sistema.

recuperação de falhas

Por fim, em termos de **concorrência** é importante que uma ou mais máquinas que trabalham em cooperaçãoumas com as outras não colidam aquando do acesso ou da preservação de algum dado numa determinada região partilhada. Por esta mesma razão há que tomar medidas em termos de evitar processos de concorrência direta, também conhecidos como condições de corrida para um mesmo recurso. Uma outra preocupação que existe em termos de concorrência prende-se com o facto de duas ou mais máquinas possuírem relógios que deverão estar sincronizados, como havemos de discutir mais à frente neste documento. Isto é um tópico bastante importante, uma vez que uma dessincronia entre duas máquinas poderá levar a uma falha de leitura, por exemplo, de uma máquina, de uma informação que uma segunda deveria ter já escrito.

concorrência

Princípios gerais de concorrência

Num ambiente multiprogramado, como já vimos, vários processos poderão existir, com diferentes tipos de comportamentos. Isto acontece principalmente porque cada par de processos poderão ser considerados como independentes (isto é, não interagem entre si ao longo da sua vida) ou como cooperantes (isto é, partilham espaços de endereçamento entre eles, trabalhando em uníssono aparente ao longo da sua vida).

Contudo, note-se que não existe uma barreira assim tão visível entre o conceito de processos independentes e cooperantes por uma razão muito simples. Consideremos o caso em que ligamos um periférico, como um ecrã externo, a um sistema computacional. Neste caso mesmo que tenhamos dois processos que não possuem qualquer tipo de relação entre eles, ambos poderão querer escrever neste mesmo periférico. Neste tipo de casos, cabe ao sistema operativo saber resolver a situação e atribuir o recurso a um dos processos de forma exclusiva, através da aplicação de mecanismos de **exclusão mútua**.

exclusão mútua

Tornar uma linguagem precisa para trabalhar este tipo de casos não é fácil, e, na verdade, grande parte das linguagens não possuem qualquer tipo de proteção em relação a estes cuidados. Por esta mesma razão é importante ter noção de que, sobre o mesmo código, poderão haver regiões que, por estarem a ser executadas em paralelo, poderão alterar regiões de memória partilhadas, com a temporização que não a devida. A estas regiões damos o nome de **regiões críticas**.

regiões críticas

Como vimos em Sistemas de Operação (a3s1), existem vários problemas que poderão ser despoletados por este tipo de situações, pelo que, ao longo dos tempos foram criadas várias soluções para o problema. Contudo, com a implementação de tais mecanismos de exclusão mútua, alguns outros problemas surgiram, dado que o seu uso em código corrente não é assim tão trivial. Surgem assim dois problemas muito habituais: **deadlock** (problema que ocorre quando dois ou mais processos estão incessantemente à espera do acesso à respetiva região crítica, deixando à espera processos por eventos que não ocorrerão) e o **adiamento indefinido** (problema que ocorre quando um ou mais processos competem por uma mesma região crítica e, devido a uma conjunção de circunstâncias onde novos processos ocorrem continuamente e competem com o anterior pelo mesmo objetivo, o acesso é sucessivamente adiado).

deadlock**adiamento indefinido**

Contudo, para que possamos compreender de que forma é que os vários recursos poderão receber competição por parte dos vários processos de um sistema computacional

precisamos primeiro de saber descrever o que é um **recurso**. Um recurso poderá ser considerado como a matéria-prima sobre a qual um processo necessita para que possa seguir a sua execução. Basicamente um recurso tanto poderá ser um ou mais componentes físicos de uma máquina (entre processado, regiões da memória principal, entre outros...), como também poderá ser uma ou mais estruturas criadas pelo sistema operativo (como a tabela de controlo de processos — PCT —, canais de comunicação, entre outros...). Uma propriedade essencial dos recursos é o tipo de apropriação de que os processos têm em relação aos vários recursos disponíveis.

Podemos assim descrever os recursos de duas formas: como *preemptables* ou *non-preemptables*. Um recurso diz-se **preemptable** quando pode ser retirado da pertença de um processo sem que algo de inesperado aconteça — um exemplo desta classe de recursos são o processador e as regiões de memória principal onde o espaço de endereçamento é preservado. Por outro lado, um recurso diz-se **non-preemptable** quando não é possível retirar a sua pertença a um processo, como serão o caso uma impressora ou uma região de memória partilhada, onde mecanismos de exclusão mútua são requisitos para a sua manipulação. Este, de facto, é o único tipo de recurso que poderá ser razão de um *deadlock*, isto porque os outros poderão ser sempre retirados da aplicação.

Tal como vimos em Sistemas de Operação (a3s1) temos que, para que uma situação de *deadlock* possa ocorrer, existem quatro condições que deverão ocorrer em simultâneo, são elas a condição de exclusão mútua (cada recurso existente, ou está livre, ou foi atribuído a um e um só processo — a sua posse não pode ser partilhada), a condição de espera com retenção (cada processo, ao requerer um novo recurso, mantém na sua posse todos os recursos anteriormente solicitados), a condição de não-libertação (ninguém, a não ser o próprio processo, pode tomar decisão da libertação de um recurso que lhe tenha sido previamente atribuído) e a condição de espera circular (formou-se uma cadeia circular de processos e recursos, em que cada processo requer um recurso que está na posse do processo seguinte na cadeia). Tendo consciência desta implicação, podemos tentar negá-la e tentar verificar sobre que condições é que não há *deadlock*.

Negando a implicação anterior temos que se não há exclusão mútua no acesso a um recurso ou não há espera com retenção ou há libertação de recursos ou não há espera circular, então não há *deadlock*. Assim, desde que uma das condições necessárias à ocorrência de *deadlock* seja negada pelo algoritmo de acesso aos recursos, o *deadlock* torna-se impossível. Políticas com esta característica designam-se de políticas de **prevenção de deadlock** no sentido estrito. [3]

A primeira condição possível de ser negada é a de exclusão mútua no acesso a um recurso. Efetuar esta negação é uma tarefa muito restrita, uma vez que só poderá ser executada em recursos que se consideram como *non-preemptables*. Caso contrário teríamos condições de corrida o que, por si, poderia levar-nos para problemas maiores, como os de inconsistência de informação. Uma das formas de negar este princípio é a aplicação do acesso de leitura a processos múltiplos, uma vez que em nada esta ação poderá danificar os nossos dados. Contudo, se o fizéssemos para o direito de escrita voltaríamos a ter problemas de condição de corrida e subsequente inconsistência de informação.

Se tentarmos negar a segunda condição, isto é, negar a condição de espera com retenção, note-se que não conseguimos evitar que ocorra adiamento indefinido, sendo que não só devemos negar a condição, mas também garantir que, mais tarde ou mais cedo, os recursos necessários serão sempre atribuídos aos processos que precisam destes para avançar nas suas execuções. Para isto costumam-se introduzir políticas de **aging** para aumentar a prioridade de um processo quanto mais este esperar por um recurso.

A terceira condição poderá ser negada tendo então que um processo, quando não se consegue largar de todos os recursos de que precisa para poder prosseguir, deverá largar todos os recursos da sua posse (sem exceções) e recomeçar do início o seu progresso mais tarde. Alternativamente, isto também poderá significar que um processo só poderá ter um recurso de cada vez, embora esta resolução seja meramente particular para alguns casos, não podendo ser generalizada.

recurso

preemptable

non-preemptable

prevenção de deadlock

aging

Neste processo de cancelar a condição de libertação de processos é importante que se tenha cuidado com cenários de *busy waiting*, ou seja, as tarefas deverão ser executadas de forma a que os processos bloqueiem uma vez tendo os recursos libertados e acordem uma vez que os recursos estão novamente disponíveis. Contudo, o adiamento indefinido não está resolvido garantidamente. Este problema só ficará resolvido quando garantir que, mais tarde ou mais cedo, os recursos necessários serão sempre atribuídos a qualquer processo que tenham necessidade destes. Mais uma vez, uma possível solução é a introdução de mecanismos de *aging*.

Por fim, a nossa última condição poderá ser negada estabelecendo uma ordenação linear dos recursos e fazendo com que o processo, quando tente obter o recurso que precisa para retomar a execução, peça os mesmos em ordem crescente do número associado com cada um. Desta forma, a possibilidade de formação de uma cadeia circular de processos com pertença de recursos e pedidos para outros é previnida. Note-se, não obstante, que voltamos a não excluir o caso de adiamento indefinido, uma vez que esta metodologia deverá contar com a garantia de que os recursos necessários deverão ser sempre atribuídos a qualquer processo que os peça. Voltamos, mais uma vez, a indicar políticas de *aging* como possível solução deste problema.

Para resolver estes problemas em termos de implementação, nos anos '60, Hoare e Brinch Hansen desenvolveram um mecanismo a que damos o nome de **monitores**. Um monitor é assim um dispositivo de sincronização que poderá ser visto como um módulo específico estruturado dentro do contexto de uma linguagem de programação concorrente. É, no fundo, uma estrutura de dados, com um código de inicialização e com um conjunto de primitivas de acesso.

Como vimos na primeira secção deste documento, uma aplicação escrita numa linguagem de programação concorrente e que implemente o paradigma de variáveis partilhadas, pode ser vista como um conjunto de *threads* (fios de execução) que competem pelo acesso a estruturas de dados partilhados. Quando estas estruturas de dados são implementadas com base em monitores, o mecanismo de execução da linguagem em questão deverá garantir que tudo aquilo que seja executado tendo em conta as primitivas dos monitores usadas, acontecerá na disciplina de exclusão mútua. Assim, o compilador, no processamento de um monitor, gera código necessário para a imposição de restrições ao acesso de recursos, de uma forma totalmente transparente ao programador.

Então como é que isto poderá ocorrer? Basicamente, quando uma *thread* entra num monitor através da invocação de uma das suas primitivas que constituem a única forma de entrada nos mesmos, esta exige o fecho do mecanismo de exclusão mútua, permitindo que um determinado recurso esteja exclusivamente na pertença de um processo em particular. Se outra *thread*, entretanto, tentar entrar neste mesmo período, será bloqueada, ficando à espera do seu turno.

Para que isto aconteça há a necessidade de ocorrer **sincronização** entre as várias *threads*, através de **variáveis de condição**. Uma variável de condição é um dispositivo especial, definido dentro de um monitor onde uma *thread* poderá ser bloqueada, enquanto aguarda por um evento que lhe permita continuar a execução. Dentro do monitor, existem duas funções que permitem a sua manipulação: a primitiva **wait** — a *thread* invocadora fica bloqueada na variável condição passada como argumento e é colocada fora do monitor para permitir que outra *thread*, que queira entrar, possa prosseguir; **signal** — se existem *threads* bloqueadas na variável de condição passada como argumento, estas ficam acordadas, prosseguindo execução (caso contrário, nada acontece).

Para impedir a coexistência de duas *threads* dentro de um monitor, é necessária uma regra que estipule como a contenção decorrente do sinal é resolvida. Assim sendo, várias soluções foram aparecendo, entre as quais os monitores de Hoare, os monitores de Brinch Hansen e os de Lampson/Redell.

Comecemos pelos **monitores de Hoare**. Nestes monitores, a *thread* que invoca a operação de envio de sinal é colocada fora do monitor, para que a *thread* acordada possa prosseguir. Esta implementação está designada num diagrama na Figura 1.4.

• Sir Charles Hoare
monitores

• Per Brinch Hansen

sincronização
variáveis de condição

wait

signal

• Butler W. Lampson
monitores de Hoare

• David D. Redell

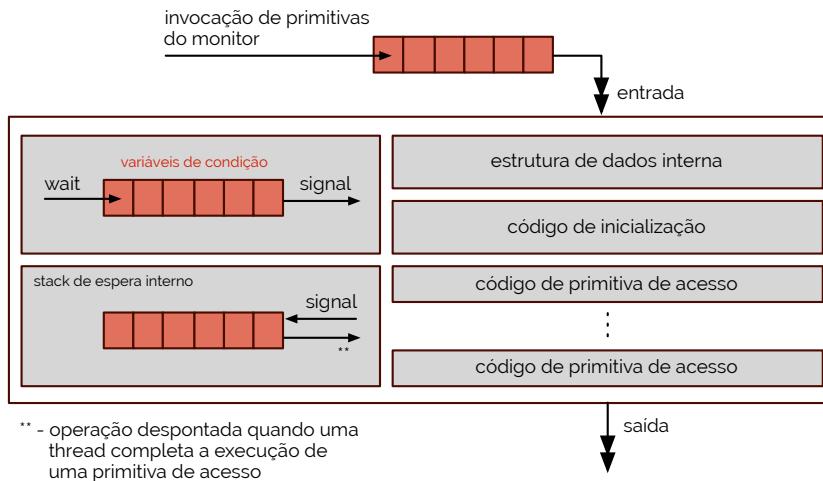


figura 1.4

A solução apresentada por Hoare é muito geral, mas a sua implementação exige também a existência de uma *stack* (não visível na Figura 1.4) onde são colocadas as *threads* postas fora do monitor por invocação de um sinal.

Com a solução implementada por Brinch Hansen, a *thread* que invoca a operação de envio de sinal liberta imediatamente o monitor, como podemos ver na Figura 1.5.

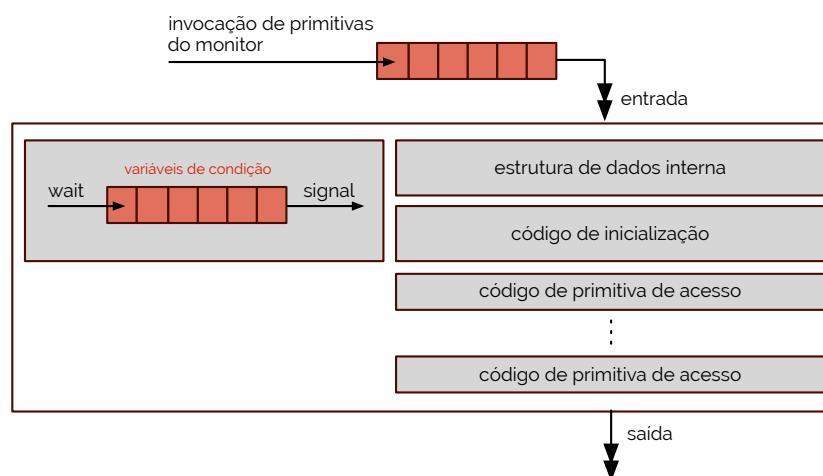


figura 1.5

A solução da Figura 1.5, sendo que, desde a anterior, lhe removemos o *stack* de espera interno, é simples de implementar, mas pode tornar-se bastante restritivo porque só há a possibilidade, agora, de execução de um sinal por cada invocação de uma primitiva de acesso. Para corrigir esta situação chega-nos o **monitor de Lampson-Redell**, onde a *thread* que invoca a operação de envio do sinal prossegue a sua execução e a *thread* acordada mantém-se fora do monitor, competindo pelo acesso a ele. Esta solução é simples de implementar, podendo ser vista na Figura 1.6, mas podendo originar situações em que algumas *threads* são colocadas em adiamento indefinido.

A implementação deste tipo de mecanismos em Java é bastante simples, uma vez que, por se tratar de uma linguagem de programação concorrente, já possui um monitor integrado em todas suas classes que poderão ser criadas. Na verdade, a linguagem Java suporta monitores de Lampson-Redell como o seu dispositivo nativo de sincronização. Neste cenário, cada tipo de dados referência poderá ser tornado num monitor, sendo que se ativa o mecanismo de exclusão mútua e de sincronização de *threads* quando métodos estáticos são invocados sobre este. Também neste cenário, cada objeto instanciado poderá ser tornado num monitor, ativando a garantia de exclusão mútua e de sincronização de *threads* quando métodos instanciados são invocados sobre este.

De facto, e tendo em conta que Java é uma linguagem que segue o paradigma de orientação a objetos, cada *thread*, sendo um objeto, também poderá ser tornada num moni-

monitores de
Lampson-Redell

tor. Esta propriedade, se for mal usada, poderá ativar o cenário em que uma *thread* bloqueia o seu próprio monitor, uma vez que ambas entidades são equiparáveis.

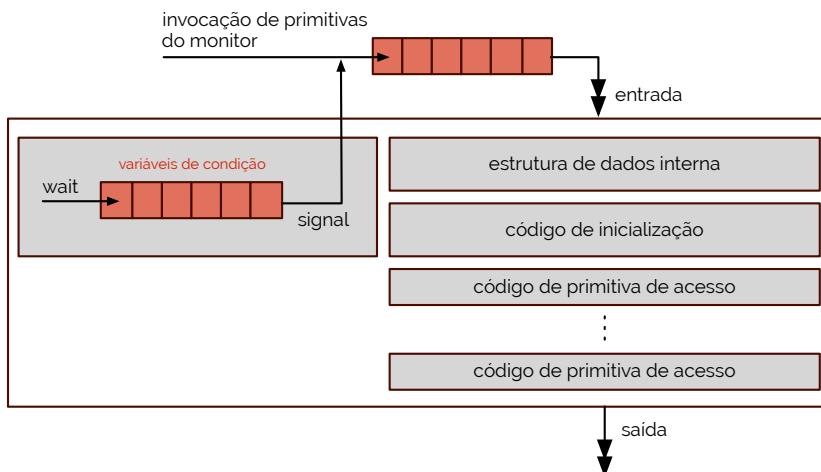


figura 1.6

Contudo, a implementação de um monitor de Lampson-Redell na linguagem Java possui algumas peculiaridades. Primeiro, o número de variáveis de condição está limitada a uma, referenciada de uma forma implícita através do objeto que é representado em pleno tempo de execução (*runtime*) pelo tipo de dados referência ou qualquer uma das suas instâncias. Segundo, a operação que em Sistemas de Operação (a3s1) denominámos de `signal` é aqui denominada de `notify`, havendo ainda a variação `notifyAll` (mais usada), que permite que se acordem todas as *threads* atualmente bloqueadas na variável de condição. Mais, existe um método no tipo de dados `Thread`, denominado de `interrupt`, que quando invocado numa *thread* específica tenta acordá-la através do lançamento de uma exceção, se esta se encontrar bloqueada numa variável de condição.

A necessidade da operação `notifyAll` tornar-se-á evidente quando pensarmos que, por ter uma única variável de condição por monitor, a única forma de acordar uma *thread* que se encontra bloqueada por uma condição em específico, é acordando todas as *threads* existentes, modificando apenas o valor da *thread* pretendida.

Um outro mecanismo que pode ser aplicado para sincronizar várias *threads* no acesso a regiões partilhadas e, assim, aplicar critérios de exclusão mútua, são os **semáforos**. Como também abordámos em Sistemas de Operação (a3s1), Edsger Dijkstra, nos anos '60, criou os semáforos, sendo estes apenas uma variável inteira que controla o número de entidades que estão dentro e fora de uma região partilhada (esta última usando uma fila de processos em espera), sendo que nela apenas duas operações poderão ser tomadas: a operação de **up** — se existem processos bloqueados em fila de espera, então um deles é acordado, caso contrário o valor do semáforo é aumentado numa unidade; e a operação de **down** — se o valor do semáforo for diferente de zero, então o valor é decrementado numa unidade, caso contrário o processo que invoca esta operação é bloqueado e colocado em fila de espera para a obtenção do recurso de que necessita. [4]

semáforos

○ **Edsger Dijkstra**

up

down

2. Modelos de Sistemas Distribuídos

Como já abordámos, os sistemas distribuídos são tais cuja complexidade se extende por um conjunto de componentes que, por definição, se encontram disponíveis entre várias máquinas. Estando desenhadas para operar no contexto do mundo real, este tipo de sistemas tem de estar pronto para qualquer eventualidade, independentemente do local onde se instalam (isto é, poderá ser sujeito a sistemas operativos diferentes, a *hardwares* distintos entre as várias máquinas do sistema, entre outras variáveis). Por esta mesma razão é importante ter consciência de uma criação de modelos próprios, convencionais, para a implementação de sistemas de grande escala, para que as várias partes que possam formar o todo trabalhem futuramente em uníssono.

Uma das partes mais importantes da criação de sistemas distribuídos é a separação de aplicações das suas plataformas onde correm, através do fornecimento de uma camada lógica de funcionamento a que denominámos já de *middleware*. Com esta transparência realizada, falta-nos saber como é que poderemos montar os nossos sistemas daí para baixo, isto é, da nossa camada de aplicação até à nossa camada física. Por esta mesma razão é importante ter um conjunto de **modelos** que nos sejam capazes de descrever os vários requisitos e o protocolo de transmissão e de comunicação entre as várias máquinas e entidades ativas/passivas do sistema.

modelos

Começando pelo ponto onde ficámos na secção anterior, a partir da nossa camada lógica de *middleware* necessitamos de saber descrever como é que funciona o corpo da nossa aplicação em termos funcionais — verificando modelos fundamentais, isto é, olhando para a forma como o nosso sistema poderá comunicar em termos abstratos (comunicação entre várias entidades e regiões partilhadas) —, como é que a nossa aplicação se dispõe em termos lógicos entre as várias máquinas de que é constituída — verificando modelos de arquitetura — e, finalmente, como é que o nosso sistema é constituído a nível físico — do que é que é constituído, em termos de componentes físicos.

Modelos de arquitetura (cliente-servidor, pares e publicador-subscritor)

A **arquitetura** de um sistema traduz-se pela sua estrutura em termos dos seus componentes e das suas relações entre eles. Pensemos em termos da construção de uma casa: quando se desenha uma casa não só se tenta elaborar um raciocínio próprio em termos de estilo artístico, como também se tenta julgar quais os requisitos a que o novo edifício se terá de reger, assim como tentar prever quais são os futuros requisitos que poderão vir a existir. Temos assim que a criação de um modelo de arquitetura em muito depende dos requisitos atuais e futuros de um sistema enquanto implementado. Para a sua criação temos, então, que pensar em termos de escabilidade, heterogeneidade, abertura e de qualidade de serviço disponibilizado.

arquitetura

Antes de avançarmos mais temos, primeiro, de tomar consciência sobre quais são os elementos fundamentais dos nossos modelos, isto é, quais são as várias entidades e regiões que formarão o nosso sistema, que enquanto está implementado, como quando está em desenvolvimento. Ora, uma vez que partimos de uma solução com vista a tópicos de concorrência, sendo que teremos sempre várias *threads* a concorrer por regiões que lhes são partilhadas, temos então que possuímos um conjunto de entidades comunicantes que são formadas, essencialmente, por processos. Contudo, note-se que este tipo de discussão poderá não ser assim tão clara aquando da perspetiva de um sistema. Consideremos um ambiente primitivo, tal como uma rede de sensores, em que os sistemas operativos incluídos poderão não suportar abstrações de processos (ou qualquer outro tipo de isolação) — neste caso teremos, como entidades comunicantes, nós de uma rede. Contudo, no caso da maior parte dos sistemas distribuídos, os processos, como são suplementados por conjuntos de *threads*, podemos dizer que as suas entidades comunicantes são *threads* (fios de execução).

Neste ambiente de concorrência, contudo, temos mais elementos com que contar. Para além das *threads* que trabalham em colaboração umas com as outras, estas trabalham dentro de uma região partilhada que não é nada mais nada menos que um espaço de endereçamento de todos os processos intervenientes, protegido por um monitor que lhe garante exclusão mútua. Este processo de sincronização dado pelo monitor é realizado pela utilização de variáveis de condição por parte do monitor, num modelo de interação que podemos caracterizar como **reativo**, isto é, onde cada processo corre até posterior bloqueio ou término.

reativo

Consideremos o seguinte cenário, numa corrida de cavalos [5], em que vários espetadores vão para um centro de apostas efetuar as suas apostas num ou mais cavalos que estão prestes a correr. Neste caso, considerando o nosso centro de apostas como a nossa região partilhada e os nossos espetadores, assim como o corretor de apostas (quem receberá e tratará do registo das apostas dos espetadores), como as nossas *threads* cooperantes

(as nossas entidades comunicantes), na Figura 2.1 podemos ver uma representação da nossa arquitetura atual.

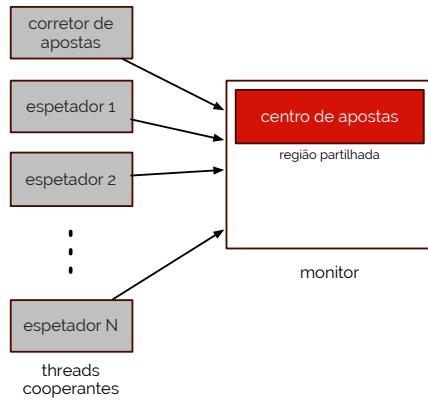


figura 2.1

Se pegarmos no nosso modelo desenvolvido na Figura 2.1 e quisermos levar o mesmo sistema para um conceito distribuído o que é que temos de fazer? Ora, podemos começar por identificar que entidades poderão ser classificadas como ativas e que entidades poderão ser classificadas como passivas. Por entidades ativas pretendemos identificar os vários objetos que, por iniciativa própria, interagem com outros no curso da sua vida. Pelo contrário, por entidades passivas, pretendemos identificar as que, no curso da sua vida, permanecem à espera do contacto de outros objetos e se caracterizam por terem uma vida eterna.

Tendo identificado as várias entidades ativas e passivas o mais importante a ser feito agora é segmentar cada uma destas para uma máquina. Uma vez que estamos perante um conjunto de entidades ativas e outras passivas, muito possivelmente a primeira ideia de arquitetura que iremos obter será tal de **cliente-servidor**. Uma arquitetura de cliente-servidor carateriza-se assim por ter um conjunto de máquinas que necessitam de interromper o seu progresso para a obtenção de informação que outra entidade lhe tem de fornecer, dado um pedido pela mesma.

Neste processo de migração dos processos e dos seus recursos partilhados para diferentes sistemas computacionais, os seus espaços de endereçamento tornar-se-ão disjuntos, sendo que haverá a necessidade de efetuar comunicação entre os vários nós do sistema. Para isto, haverá a necessidade de criar canais de comunicação, quer de forma implícita ou explícita, para que haja uma troca de mensagens ou outro tipo de mecanismo de execução de sistemas distribuídos, como a invocação remota de métodos, como iremos ver mais à frente.

Contudo, sendo as nossas entidades passivas meros servidores que não possuem qualquer tipo de interface para receber pedidos e enviar respostas, como é que poderemos implementar a arquitetura de cliente-servidor? Ora, para que se possa implementar este tipo de serviços há que, necessariamente, criar um novo processo que não só se encarregue de comunicar com os outros processos (que lhe faz pedidos), como também executar localmente as operações pedidas no recurso partilhado.

Assim, uma aplicação do modelo de cliente-servidor do mesmo conceito que o implícito na Figura 2.1 poderá ser visto na Figura 2.2. Interpretando esta figura podemos verificar que cada processo que antes era uma mera *thread* cooperante, agora está incluída dentro de uma região que se encarrega de possuir uma interface de contacto para com o servidor. O servidor, por sua vez, é não só uma região partilhada, como também uma interface de contacto para com os vários clientes que lhe queiram efetuar pedidos e obter respostas — ao processo que trata de criar esta interface dar-lhe-emos o nome de *processo ativo*.

Às operações que podemos executar sobre a nossa nova arquitetura dar-lhe-emos os seguintes nomes: **pedido** — o cliente processa chamadas ao servidor pedindo-lhe que execute operações por ele; **resposta** — o servidor processa respostas ao cliente comunicando-lhe os resultados da operação; e **execução local** — execução por parte do servidor, dos pe-

cliente-servidor

pedido
resposta
execução local

didos efetuados pelo cliente, que posteriormente terão os seus resultados comunicados de volta ao mesmo.

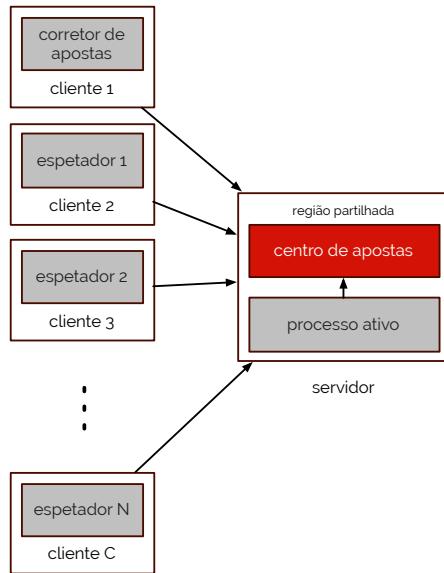


figura 2.2

Na Figura 2.2 temos o nosso servidor que, em suma, possui dois novos papéis: possui um processo ativo, que será o nosso gestor de comunicação (entidade que esperará pelos pedidos de serviço pelos processos de clientes), e um **agente prestador de serviço**, que executará as operações pedidas na região partilhada, como um intermediário do processo do cliente. Note-se que este modelo de comunicação é totalmente assimétrico, ou seja, por ser baseado em pedidos, há elementos que necessitam de aguardar pelo processo de outros, estes, que poderão demorar tempo imprevisível. Mais, neste tipo de arquiteturas, como já referimos de forma mais sumária, o nosso servidor assume um ciclo de vida puramente eterno, enquanto que os nossos clientes têm nascimento, vida e morte. Isto acontece porque o sistema possui a característica de ter disponibilidade máxima, enquanto que os seus clientes só se manifestam de tempos a tempos. Mais, enquanto que os nossos servidores são públicos, os nossos clientes são privados, sendo que a operação de serviço requer que o servidor seja conhecido por todas as partes que lhe são interessadas, enquanto que os seus utilizadores não necessitam de ser previamente conhecidos pelos servidores.

De uma forma mais refinada, na Figura 2.3 podemos ver a arquitetura básica de um servidor.

agente prestador de serviço

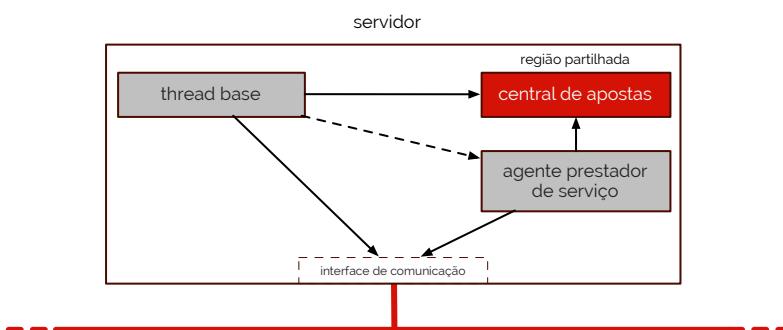


figura 2.3

Na Figura 2.3 temos então que um servidor é essencialmente caracterizado como sendo um elemento que possui uma *thread base*, uma região partilhada (no nosso caso do cenário de uma corrida de cavalos, uma central de apostas), um agente prestador de serviço e uma interface de comunicação para uma canal de comunicação onde os clientes se podem ligar. A **thread base** é tal que instancia a região partilhada e o canal de comunicação, mapeando o servidor num endereço público conhecido. Sendo esta, essencialmente, a primeira entidade que se encontra ligada com a interface de comunicação com o canal principal, quando

thread base

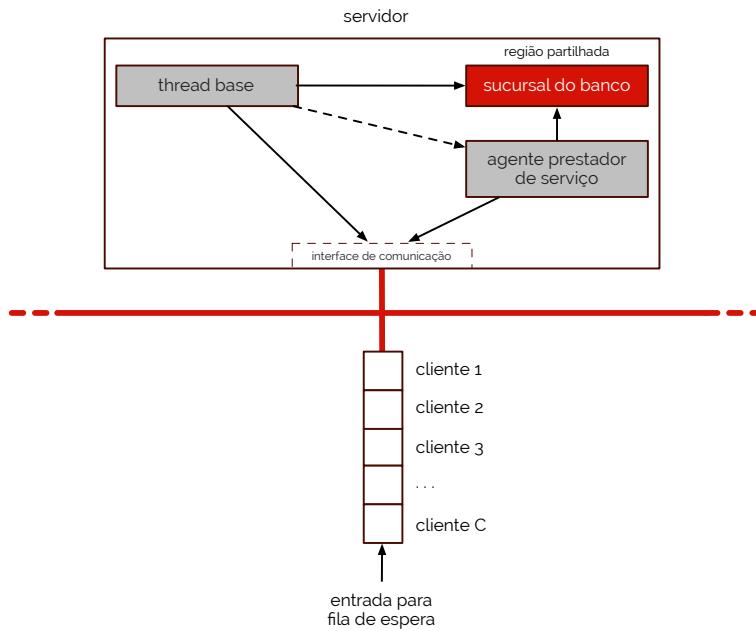
uma conexão com um cliente é estabelecida, esta deverá criar um agente prestador de serviço para que o pedido seja devidamente atendido. Havendo criado o agente prestador de serviço, é então importante que este determine qual é a operação que o cliente pretende que seja executada na região partilhada. Depois disto, o agente deverá, então, executar o pedido e comunicar o resultado de tal operação à processo do cliente, mais uma vez, utilizando a interface de comunicação.

Note-se que, para que este procedimento possa ocorrer, e como alguns dos possíveis resultados poderão estar descritos sobre objetos de classes complexas, é importante que haja uma forma de enviar os dados todos, juntamente com a forma das suas estruturas de representação, pelo canal de comunicação. Para isso diz-se haver a necessidade de **serialização**.

serialização

Esta serialização não passa apenas pela questão da transmissão de dados. O que é que acontece se mais do que um cliente efetuar um pedido ao servidor em simultâneo? Consideremos assim a seguinte analogia: estamos numa sucursal de um banco e precisamos de depositar dinheiro na nossa conta. Para o fazer aproximamo-nos do balcão e alguém nos irá atender, se mais ninguém estiver presente para ser atendido — atender-nos-á uma agente prestadora de serviço (alguém que na região da sucursal do banco, que é passiva, isto é, por si não faz nada, é capaz de efetuar a tarefa de nos depositar o dinheiro). Se houver mais gente para ser atendida no banco teremos de ficar à espera, sabendo que seremos, entre todos na fila, atendidos de vez a vez — eis a nossa serialização, sob a forma de uma fila de clientes, como podemos ver na Figura 2.4.

figura 2.4



Em suma, em termos de uma arquitetura de cliente-servidor, podemos ter uma primeira variante em que temos uma serialização de pedidos, onde somente um processo de cliente é servido de cada vez, o que significa que a *thread base*, quando recebe um pedido de ligação, instancia um agente prestador de serviço e espera pela sua terminação antes de voltar a ficar à escuta do canal de comunicação. Aqui, a região partilhada não necessita de nenhuma proteção especial para garantir exclusão mútua no acesso, sendo que não há mais do que uma *thread* de agente prestador de serviço. Temos assim uma arquitetura muito minimizada, onde de uma forma algo ineficiente o tempo de serviço não é controlado (sendo que não se tira vantagem dos “tempos mortos” entre as várias interações das entidades, devido à falta de competição e dá-se aso a casos de *busy waiting*, onde se efetuará a tentativa de sincronizar múltiplos clientes num mesmo recurso).

Numa segunda variação da arquitetura de cliente-servidor podemos encontrar um caso em que a dimensão dos agentes prestadores de serviço deixam de ser um somente. Consideremos assim o caso em que estamos novamente na sucursal do banco e queremos ser

atendidos. Por sorte, um senhor aparece do outro lado do balcão dizendo que também poderá atender o pedido ao mesmo tempo que a sua colega. Temos assim um sistema em que os agentes prestadores de serviço são múltiplos e poderão atender mais do que um cliente em simultâneo, claro, regendo-se à ordem da fila de espera de clientes, como podemos ver na Figura 2.5.

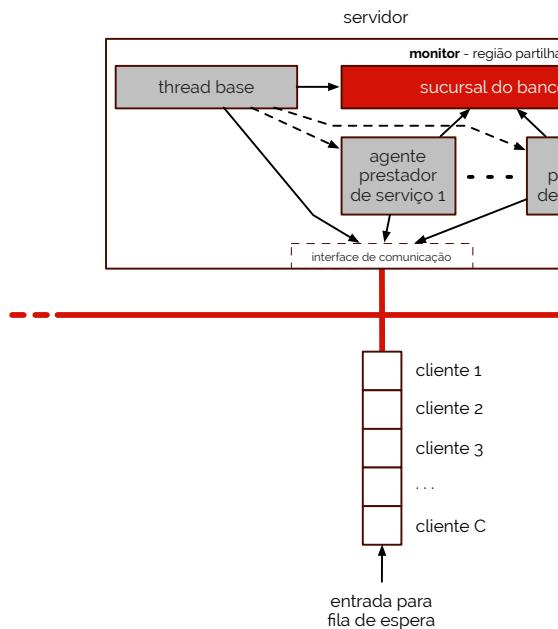


figura 2.5

Neste caso temos então um conjunto de processos de clientes que são servidos de forma concorrente, pelo que a *thread base*, sempre que recebe um novo pedido de um cliente, instancia um agente prestador de serviço e inicia novamente a escuta no canal. A região partilhada é transformada num monitor, de forma a garantir exclusão mútua no seu acesso, sendo que agora temos múltiplos agentes prestadores de serviço em serviço ao mesmo tempo. Esta é a forma tradicional sobre a qual os servidores são montados, onde se tenta tirar o máximo proveito dos recursos de um sistema computacional onde este se encontra. Aqui o tempo de serviço, por serem bem aproveitados os recursos, é minimizado, sendo que se tira vantagem dos períodos mortos das interações entre entidades e a sincronização é feita entre vários clientes sobre a mesma região partilhada.

Por fim, podemos ter o caso em que, de volta ao banco, dentro da mesma sucursal temos um balcão novo, de atendimento. Agora, não só temos múltiplos agentes prestadores de serviço, como também podemos ser atendidos sobre duas ou mais filas de espera. Sendo atendido em diferentes balcões em nada é diferente, sendo que a interação sobre as entidades do banco, principalmente da sucursal, são as mesmas. No fundo, é como se o novo balcão fosse uma réplica da sucursal do banco no exemplo anterior, como podemos ver na Figura 2.6.

Neste caso temos que o serviço está disponível em vários sistemas computacionais, cada um executando uma arquitetura como a representada na Figura 2.5 nos seus servidores. Note-se que, aqui, a nossa região partilhada é replicada em cada servidor, tornando-se assim num modelo de arquitetura algo sofisticado, que possui como objetivo principal a maximização da disponibilidade do serviço e da minimização do tempo do mesmo, mesmo em situações de pico (quando a taxa de chegada dos clientes é máxima), potenciando fatores como a escalabilidade. Note-se que o serviço é então mantido operacional e tolerante às falhas de alguns servidores, sendo os pedidos dos clientes distribuídos entre os vários servidores através da utilização de políticas fornecidas por mecanismos como os de DNS ou de organizações geográficas para pedidos globais. Neste modelo, sempre que há uma alteração de dados locais numa das várias réplicas das regiões partilhadas nos servidores, a necessidade de manter as várias regiões consistentes aumenta e deverá ser tida em conta. Na Figura 2.6 podemos ver uma esquematização deste modelo de arquitetura.

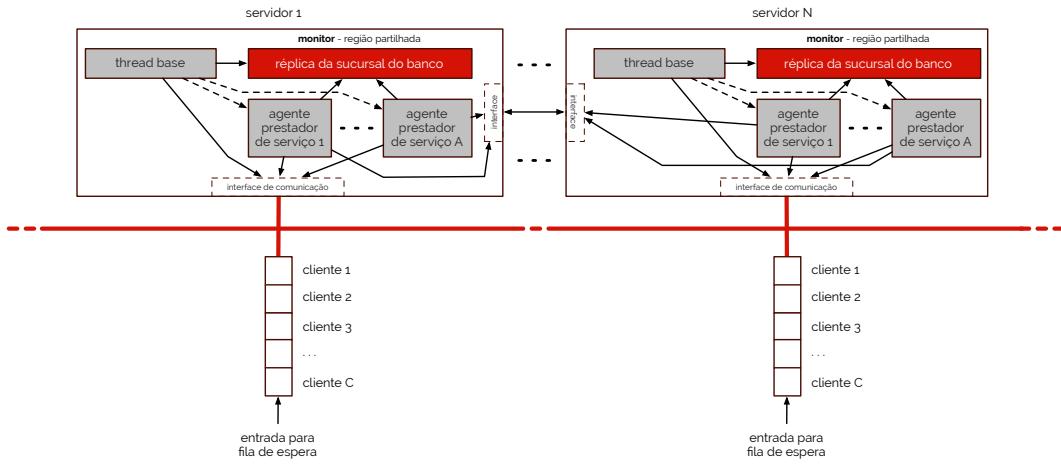


figura 2.6

Um outro tipo de arquitetura possível de ser implementado é a **comunicação entre pares**. Neste caso temos uma topologia principal em que, estando todos os nós ligados da mesma forma a uma rede global, e desempenhando todos a mesma função, o serviço está simultaneamente disponível em vários sistemas computacionais. Por outras palavras, no que toca ao serviço, não existe qualquer tipo de diferença entre nós de processamento, aos quais daremos o nome de **peers**.

Recordando as redes *peer-to-peer* que abordámos em Arquitetura de Redes Avançadas (a4s1), temos que aqui as nossas regiões partilhadas são replicadas de igual forma em cada peer, dando aso a múltiplas cópias. Mais uma vez estamos perante um modelo que poderá ser considerado como sofisticado, uma vez que tenta-se maximizar a disponibilidade do serviço e minimizar o tempo do mesmo, mesmo que em situações de pico de taxa de clientes à chegada, potenciando fatores como o da escalabilidade.

Note-se ainda que, em algumas situações, um dos peers terá de assumir como sendo o nó principal para controlo de toda a rede, de forma a que o sistema por si possa efetuar transições de estados sem qualquer tarefa ou problemas inesperados. A escolha deste nbó líder é tipicamente realizada segundo um processo de eleição, onde se deverá atingir um consenso, como veremos mais à frente. Na Figura 2.7 podemos ver um diagrama que descreve este conceito em termos de ligação dos servidores.

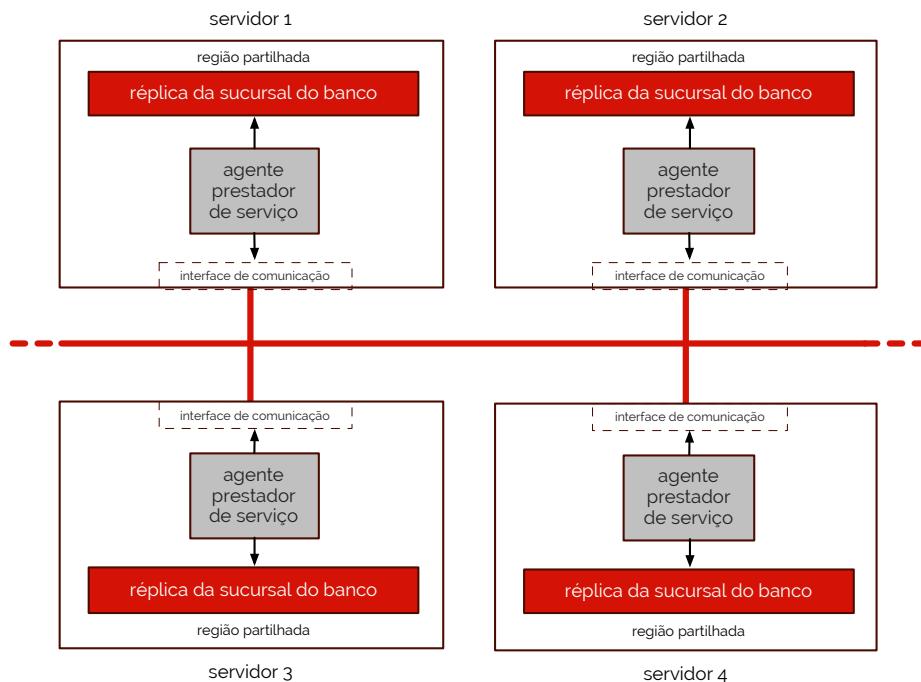


figura 2.7

comunicação entre pares

peers

Com estes modelos entramos num paradigma a que denominamos de **passagem de mensagens**. Neste tipo de troca tem de existir uma entidade que reencaminha mensagens (o passador, que as envia) e uma entidade que recebe as mensagens (o receptor). Para que isto possa ter algum valor, duas primitivas terão de ser designadas: a primitiva de envio e uma de receção. A primitiva de envio terá de possuir, pelo menos, dois parâmetros: um primeiro que descreve o endereço de destino e um segundo que designa uma referência para o *buffer* no espaço do utilizador que possui os dados a serem enviados. De forma algo análoga, a primitiva de receção também necessitará de dois parâmetros de entrada: um primeiro que descreve o endereço de origem e um segundo que designa uma referência para o *buffer* no espaço do utilizador para onde se pretende guardar os dados recebidos.

Normalmente este tipo de comunicação sofre alguns processos de *buffering*, isto é, na operação de envio os dados são primeiramente transferidos para o *buffer* do *kernel* antes de serem entregues à rede. Já na operação de receção os dados também são primeiramente preservados num *buffer* do *kernel* e só depois transferidos para o *buffer* do utilizador quando a respetiva primitiva é invocada.

Estas primitivas poderão, entretanto ser categorizadas entre síncronas/assíncronas e bloqueantes/não-bloqueantes. O que é que cada uma significa? Diz-se que o envio/receção é **síncrono** quando ambas as primitivas estão acopladas uma com a outra, sendo que a operação de envio só se completa quando o processo do passador fica ciente que, no processo do receptor, a mensagem já chegou e completou a sua ação. Por outro lado dizemos que as operações são **assíncronas** se e só se ambas primitivas estão totalmente desacopladas, isto é, quando a operação de envio termina assim que transpuser a mensagem a enviar para o canal de comunicação. Note-se que, neste paradigma, não existe qualquer tipo de primitiva de receção que seja assíncrona.

Para além da característica de sincronismo, dizemos que as primitivas são **bloqueantes** se o controlo só for retomado ao processo invocador depois da operação ter sido terminada (quer estejamos a falar de primitivas síncronas ou assíncronas). Por outro lado, dizemos que são **não-bloqueantes** se o controlo é retomado ao processo invocador imediatamente depois de ser executada, mesmo que a respetiva operação ainda não tenha sido terminada.

Por fim, uma última arquitetura importante de ser referida é a de **publicador-subscritor**. Nesta arquitetura temos um conjunto de entidades (*threads*) que consultam um meio de comunicação global (denominado usualmente de **broker**) onde várias outras entidades (*threads*) vão buscar informação. Às entidades que escrevem dados no *broker* damos também o nome de **publicadores**, enquanto que às que lá vão ler dados damos o nome de **subscritores**. A única diferença entre este modelo e um modelo mais convencional de cliente-servidor está no facto de não haver qualquer tipo de interação síncrona a acontecer.

Alguns dos exemplos de aplicação para este tipo de topologia poderão ser vistos no caso do correio eletrónico, em que há um conjunto de entidades que enviam mensagens para um *broker* que não é nada mais nada menos que a nossa caixa de correio. Depois, num instante de tempo que não tem de ser necessariamente o mesmo que quando recebemos a mensagem, como subscritores da nossa própria caixa de correio, vamos consultar o seu conteúdo.

Modelos fundamentais

Num sistema distribuído, onde vários processos cooperam e comunicam sobre um conjunto de canais montados numa rede de computadores, o desempenho muitas vezes é um fator que diferencia a eficácia dos sistemas e a sua própria eficiência, especialmente no que toca à troca de mensagens sobre os mesmos.

Quando possuímos um serviço externo que corre de forma remota, a rapidez de troca de mensagens irá depender diretamente das capacidades de roteamento e de encaminhamento das interligações que ocorrem entre nós de uma rede maior. Contudo, esta não é a única parte integrante do nosso problema, uma vez que o nosso sistema também terá a sua capacidade de resposta dependente da capacidade dos componentes físicos onde reside.

passagem de mensagens

síncrono

assíncrono

bloqueantes

não-bloqueantes

publicador-subscritor
broker

publicadores
subscritores

Contudo, não nos vale de nada preocuparmo-nos somente com fatores de desempenho para uma melhor qualidade de serviço, quando ainda existem muitos outros fatores que ficam por aberto uma vez que um determinado sistema passa a estar distribuído: referimo-nos a fiabilidade, segurança, adaptabilidade e disponibilidade.

Em aplicações que lidam diretamente com dados que deverão ser distribuídos em tempo-real, a disponibilidade dos recursos de computação e de rede em instantes específicos é mais-que-essencial.

Como vimos em alguns dos conceitos que abordámos em Arquitetura de Redes Avançadas (a4s1) os canais de comunicação entre várias máquinas numa rede de computadores podem ser implementados através de fluxos de dados que começam e terminam nos extremos das ligações ou através de uma troca normal de mensagens que passam por toda a rede. Contudo, em comum, temos propriedades que determinarão o seu desempenho, são elas: a **latência**, sendo esta o atraso entre os tempos em que um emissor inicia a transmissão de uma mensagem e o receptor inicia a sua receção; a **largura de banda**, sendo a quantidade total de informação que pode ser transmitida pela rede num determinado instante de tempo; e o **jitter**, sendo este a variação de tempo que demora a ser entregue uma sequência de mensagens semelhantes entre dois extremos conhecidos.

Num sistema distribuído é, então, muito difícil marcar limites precisos em termos de tempo perdido na execução de processos, na entrega de mensagens ou nos desvios do relógio.

Também é essencial que tenhamos noção que os canais de comunicação que ligam duas ou mais máquinas do nosso sistema distribuído podem falhar. Isto significa que os seus comportamentos poderão desviar-se do padrão que fora desenhado ser o normal ou correto. Podemos, dado isto, classificar as falhas de várias formas: como **falhas de omissão**, quando as ações prescritas a tomarem lugar simplesmente não ocorrem; como **falhas de temporização**, quando as ações prescritas para serem executadas num dado intervalo de tempo não são feitas; ou como **falhas arbitrárias** (ou bisantinas), quando um erro inesperado ocorre e pode, temporariamente ou permanentemente, ser resultado de um problema mais grave de um componente do sistema.

Finalmente, em termos de **segurança** de um sistema distribuído temos que este critério poderá ser atingido através da proteção dos vários processos e dos vários canais de comunicação existentes no sistema. Tal proteção deverá ter em conta possíveis acessos não autorizados ou cifragem dos vários dados que neles estão encapsulados.

Note-se que os vários recursos são objeto de uso completamente distinto de utilizador em utilizador. Estes tanto podem ter dados privados, como dados públicos que podem ser acedidos por qualquer um. Para que haja um suporte a tal tipo de ambiente, há a necessidade de criar protocolos de direitos de acesso, definindo e especificando quais as operações estão disponíveis e quem é que as poderá executar num determinado recurso.

Neste processo, é de bom tom que seja o servidor (ou um servidor) o encarregue pela verificação da identidade do causador de cada um dos pedidos que lhe são feitos, antes de qualquer tipo de resposta ser executada. O cliente, por sua vez, deverá verificar também a legitimidade das várias respostas que vai obtendo. Este procedimento é parte integrante dos vários conceitos que abordámos na disciplina de Segurança (a4s1).

3. Sincronização

O **tempo** é um assunto muito importante e interessante na temática dos sistemas distribuídos, por várias razões: primeiro, o tempo é a quantidade que mais pretendemos representar com a maior precisão possível. De forma a sabermos em que momento de um dia é que um acontecimento em particular ocorreu numa máquina específica é necessário antes ter o seu relógio sincronizado com uma entidade reguladora, que lhe forneça uma fonte externa de tempo. Por exemplo, uma aplicação de *eCommerce* tem as suas transações todas sincronizadas com uma única fonte de tempo, de forma a que não haja perdas de precisão nem desvios entre o tempo de um registo do cliente e o mesmo, mas no banco.

latência

largura de banda

jitter

fallhas de omissão

fallhas de temporização

fallhas arbitrárias

segurança

tempo

Segundo, ao longo dos tempos vários algoritmos têm sido criados para resolver a problemática da sincronização de relógios em sistemas distribuídos, como iremos verificar ao longo desta mesma secção.

Como é que medimos o tempo?

Medir o tempo não é tarefa fácil e pode ser uma tarefa extenuante devido à existência de vários momentos que podemos ter como referência. Einstein demonstrou, na sua Teoria da Relatividade, as consequências intrigantes que se podem seguir a uma observação de que a velocidade da luz é constante para todos os seus observadores, não interessando a que velocidade se encontram. Ele provou, desta consideração, entre outras coisas, que dois eventos que se julgam ser simultâneos num *frame* de referência não são necessariamente simultâneos de acordo com observadores noutros *frames* de referência que se movem em relação aos primeiros. Por exemplo, um observador que esteja em Terra e um observador que esteja num vaivém em direção à Lua irão discordar do intervalo de tempo entre eles, quanto mais a sua velocidade relativa aumenta.

Na verdade, a ordem relativa destes dois eventos até se pode inverter — só não se poderá inverter se houver uma relação de causalidade entre ambos. No caso de haver uma relação de causalidade entre ambos, então o efeito físico seguir-se-á à causa física para todos os observadores, ainda que o tempo perdido entre causa e efeito possa variar. Com isto provou-se que o tempo de acontecimentos físicos é relativo ao observador, sendo a noção de Newton (de tempo físico absoluto) uma teoria sem qualquer tipo de fundamentação. Não existe, assim, qualquer tipo de relógio físico especial no universo ao qual nós podemos invocar quando pretendemos medir intervalos de tempo.

A medição do tempo, então, faz-se de forma indireta, tirando partido da sua ligação íntima ao espaço, por consideração de movimentos periódicos. Todos os dias, o Sol aparece ao horizonte, no seu nascer a Este, erguendo-se até ao ponto mais alto do céu e voltando a descer para o horizonte, no seu pôr a Oeste. A este evento do Sol atingir o seu ponto mais alto no céu dá-se o nome de trânsito do Sol. Este evento ocorre por volta do meio-dia, todos os dias, sendo dado o nome de dia solar, ao intervalo de tempo que ocorre entre dois trânsitos do Sol. Sendo que existem 24 horas num dia e cada uma contendo 3600 segundos, um segundo solar é definido como sendo exatamente 1/86400 de um dia solar. A geometria do cálculo de um dia médio solar poderá ser vista na Figura 3.1.

◎ Albert Einstein

◎ Isaac Newton

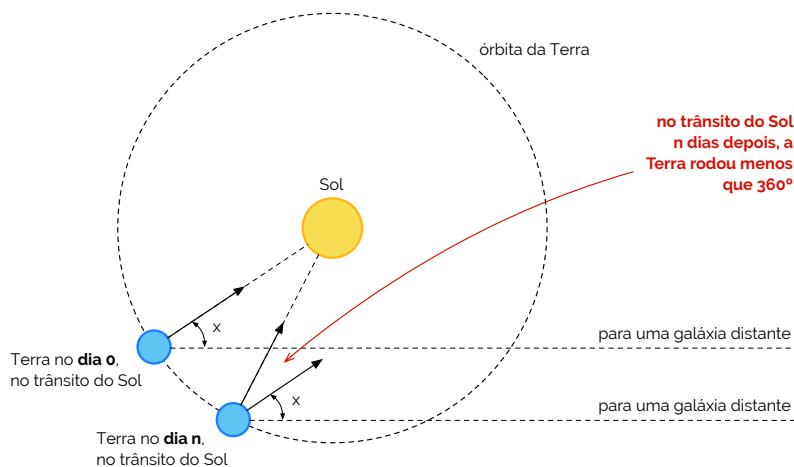


figura 3.1

Por volta dos anos '40 estabeleceu-se que o período de rotação da Terra não é constante. A Terra estava então a tornar-se cada vez mais lenta devido à fricção das ondas e ao atrito da atmosfera. Com base em estudos do crescimento de alguns corais, por parte de geólogos, acredita-se, agora, que há cerca de 300 milhões de anos atrás cada ano possuía 400 dias. Assim, o tamanho do ano (isto é, o tempo de uma volta ao Sol) pensa-se ter alterado, sendo que o dia simplesmente se tornara mais longo. Mais, provou-se que algumas

variações no tamanho do dia também ocorrem, provavelmente causadas por algumas turbulências no núcleo de ferro da Terra. Estas revelações levaram os astrónomos a calcular o tempo de cada dia através da medição de um número maior de dias e tomando a subsequente média, ao invés de dividir por 86 400. A este resultado deu-se o nome de segundo solar médio.

Claro está que, com o evoluir dos tempos, se provou que esta não era a melhor tomada de decisão para afirmar alguma precisão nestes cálculos. Com a invenção do relógio atómico em 1948, tornou-se possível medir tempo de uma forma muito mais precisa e independentemente das turbulências naturais da Terra, através do cálculo das várias transições de estado do átomo de Césio-133. Os físicos tomaram assim o trabalho de manter a precisão em relação às várias medidas do tempo e definiram um segundo como sendo o tempo que demora, ao átomo de Césio-133, a fazer, precisamente, 9 192 631 770 transições. A escolha do valor de 9 192 631 770 foi feita para tentar aproximar o valor de um segundo atómico ao valor de um segundo solar médio, no ano da sua introdução. Atualmente alguns laboratórios pelo mundo trabalham sobre esta medida, contudo, periodicamente, cada um destes laboratórios tem de comunicar ao *Bureau International de l'Heure* (BIH) em Paris quantas vezes é que o seu relógio tinha contado segundos. O BIH, entretanto, fazia a média para poder produzir o que chamamos de Tempo Atómico Internacional (TAI), embora o TAI seja somente uma média do número de contagem de segundos de relógios Césio-133 pelo mundo desde 1 de janeiro de 1958 (o início da contagem do tempo) dividido pela quantidade 9 192 631 770.

Ainda que o TAI seja bastante estável e esteja disponível para qualquer um que se queira predispor a comprar um relógio atómico, existe um grave problema com esta implementação: 86 400 segundos em TAI são cerca de 3 ms menos que um segundo solar médio, isto porque o dia solar médio está a tornar-se cada vez maior de tempos em tempos. Ao usar TAI para manter a noção de tempo significa que, ao longo do tempo, o meio-dia tornar-se-ia cada vez mais cedo, até que ocorreria no início das nossas manhãs. As pessoas, de facto, começaram a sentir isto e eventualmente ocorreria o mesmo que em 1582, quando o Papa Gregório XIII decretou que 10 dias do calendário seriam omitidos. Este acontecimento causou imensas manifestações nas ruas porque alguns senhorios queriam uma retoma da renda total de um mês e os banqueiros um mês completo de interesses, enquanto que os empregadores recusavam pagar os 10 dias em que não houve trabalho nenhum... Os países protestantes, por uma questão de princípio, não aceitaram qualquer tipo de decreto papal e rejeitaram totalmente o calendário Gregoriano por 170 anos.

O BIH resolve então o problema acrescentando o conceito de segundo bissexto sempre que houver uma discrepância entre o TAI e o segundo solar no valor de 800 ms. O uso do segundo bissexto está ilustrado na Figura 3.2.



figura 3.2

A correção como podemos ver na Figura 3.2 dá forma a um sistema temporal baseado nos segundos TAI constantes, mas que estão sempre em fase com o aparente movimento do Sol. A este sistema damos o nome de UTC (abreviatura de Tempo Universal Coordenado) — este nome provém de um acordo de nomenclatura com as designações francesas.

O que acontece desde a criação do UTC é que as companhias elétricas sincronizam o seu tempo dos seus relógios a 60 Hz ou 50 Hz com o UTC, pelo que, quando o BIH anuncia um segundo bissexto, a empresa terá de aumentar a frequência dos seus relógios para 61 Hz ou 51 Hz por 60s ou 50s (respetivamente), de forma a fazer avançar todos os seus

relógios na sua área de distribuição. Sendo que 1s é um intervalo muito bem definido para um computador, qualquer sistema operativo necessitará de manter registo do tempo com a maior precisão possível e de forma coordenada com os vários anúncios de segundos bissextos. [6] Até à data (junho de 2018), o número de segundos bissextos introduzidos foram um total de 27.

A base para manter o **tempo global** é então o UTC, sendo esta a sincronização para todos os relógios civis em termos mundiais (é uma norma internacional). Para fornecer o tempo em UTC para quem necessita de tempo preciso, por volta de 40 estações de rádio de onda curta transmitem um pulso curto ao início de cada segundo UTC. A precisão destas estações é cerca de ± 1 ms, contudo, dadas algumas flutuações atmosféricas que poderão afetar o caminho de transmissão, na prática, a precisão não será muito melhor que ± 10 ms.

Alguns satélites terrestres também têm a capacidade de nos fornecer o serviço UTC. Por exemplo, o GEOS (acrónimo para *Geostationary Environment Operational Satellite*) consegue fornecer o serviço UTC com uma precisão até 0.5 ms, havendo ainda outros satélites que o conseguem fazer melhor. Através da combinação de várias fontes satélite, os servidores de tempo terrestres conseguem oferecer uma precisão de até 50 ns. Para receber estas atualizações é possível adquirir um recetor UTC, havendo já vários computadores equipados com um. [6]

Note-se, não obstante, que para além do tempo global, isto é, do tempo percebido por um observador externo, também existem o tempo local e o tempo lógico. O **tempo local** não é, nada mais nada menos, do que o tempo percebido em cada uma das entidades observadas. Por exemplo, um sistema computacional convencional como os que possuímos nas nossas casas possui um relógio que é composto, essencialmente, por dois elementos: um circuito oscilador, controlado por um cristal de quartzo, e um contador de impulsos. O valor da contagem num dado instante é lido por uma saída em paralelo e o contador pode ser colocado num dado estado, por acerto, usando uma entrada paralela.

Esta mesma contagem é depois convertida num tempo definido numa origem muito própria (nos sistemas UNIX segue-se a convenção de que a origem do tempo está a 1 de janeiro de 1970, pelas 0h 0 min e 0s).

Contudo, alguns problemas poderão ocorrer com os relógios por si só. Um relógio que não consiga manter condições de correção considera-se como em falha. Uma falha de um relógio diz-se existir quando o relógio para de contar segundos, sendo que qualquer outro tipo de falha considera-se como falha arbitrária. Um exemplo de falha arbitrária foi o *bug Y2K*, que quebrou a condição monotónica de registar a data seguinte a 31 de dezembro de 1999 como 1 de janeiro de 1900, ao invés de 2000. Outro exemplo acontece quando as baterias dos relógios estão muito baixas e o relógio sofre desvios.

De facto o desvio é uma das razões pelas quais um relógio poderá apresentar um tempo incorreto. Define-se **desvio** quando a frequência do oscilador não é exatamente a nominal, variando ao longo do tempo com as condições ambiente (basicamente há uma contagem a um ritmo diferente). Uma segunda razão para uma má contagem do tempo existe quando há um **deslocamento**, efeito este que ocorre quando há um valor da contagem que difere do valor correto num número de impulsos fixo, havendo assim um erro na definição da origem. Note-se que o desvio poderá ser um fator caracterizável de um determinado tipo de relógios: por exemplo, um oscilador controlado por um cristal de quartzo, geralmente, apresenta um desvio da ordem de grandeza de $1/10^6$ por segundo.

Na Figura 3.3 podemos verificar os efeitos das várias falhas de relógio, quer em termos de desvios, como de deslocamentos.

Sincronização de relógios locais

De forma a podermos saber a que instante de tempo de um dia um determinado acontecimento ocorre no nosso sistema distribuído é necessário **sincronizar** os vários relógios dos vários processos C_i , com uma entidade externa de valor, fonte de tempo externa. Denomina-se a este processo o termo de **sincronização externa**.

tempo global

tempo local

desvio

deslocamento

sincronizar

sincronização externa

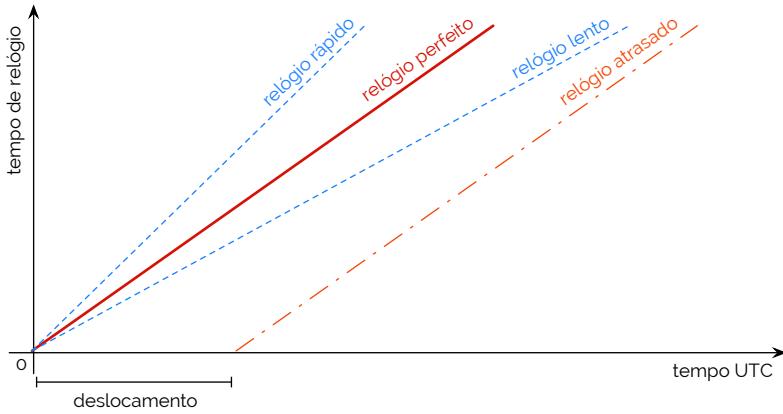


figura 3.3

Mais, se os vários relógios C_i estiverem sincronizados uns com os outros com um grau de precisão conhecido, então podemos medir o intervalo entre dois eventos que ocorrem em diferentes máquinas apelando ao relógio local de cada uma, mesmo que não estejam sincronizadas com uma fonte de tempo externa. Chama-se a este processo de **sincronização interna**. Definem-se assim os seguintes modos de sincronização, dentro de um intervalo de tempo I : para um intervalo de sincronização $D > 0$ e para uma fonte S de tempo UTC, dizemos que fazemos sincronização externa quando $|S(t) - C_i(t)| < D$ para um $i = 1, 2, \dots, N$ e para todos os tempos reais t em I (por outras palavras temos que os relógios C_i encontram-se precisos dentro do intervalo D); para um intervalo de sincronização $D > 0$, dizemos que fazemos sincronização interna quando $|C_i(t) - C_j(t)| < D$ para um $i, j = 1, 2, \dots, N$ e para todos os tempos reais t em I (por outras palavras dizemos que os relógios C_i concordam todos com o intervalo D).

sincronização interna

Note-se que dizer que relógios se encontram internamente sincronizados não é o mesmo que dizer que estes se encontram externamente sincronizados, nem uma coisa implica outra, uma vez que poderão coexistir desvios coletivos para com uma fonte externa de tempo, mesmo quando estes são concordantes entre eles (internamente sincronizados). Contudo, define-se que se o nosso sistema distribuído é externamente sincronizado com um intervalo D , então o mesmo sistema é internamente sincronizado com um intervalo de $2D$.

Algumas noções de correção para relógios têm sido sugeridas ao longo dos tempos. É então comum definir um relógio físico H como correto se o seu desvio ocorre dentro de um intervalo conhecido $\rho > 0$ (um valor definido pelo fabricante do mesmo, tal como os 10^6 ms por segundo do relógio de cristal de quartzo). Isto significa que o erro em medir o intervalo entre tempos reais t e t' (sendo $t' > t$) é contido como podemos ver em (3.1).

$$(1 - \rho)(t - t') \leq H(t') - H(t) \leq (1 + \rho)(t' - t) \quad (3.1)$$

Esta condição inibe que ocorram saltos no valor dos relógios físicos (durante um período de operação normal). Por vezes precisamos também que os relógios de *software* obejam a esta condição, contudo, uma condição mais leve de mera monotonia poderá ser suficiente. A monotonia é a condição sobre a qual um relógio C pode avançar, como indica (3.2).

$$t' > t \implies C(t') > C(t) \quad (3.2)$$

Por exemplo, o programa `make` dos sistemas UNIX é uma ferramenta que permite a compilação de códigos-fonte somente de ficheiros que sofreram diferenças ao longo do tempo desde a última vez compilados. Para isto a ferramenta usa as datas de modificação de cada par ficheiro de código/ficheiro compilado para determinar esta condição de (3.2). Se um computador cujo relógio estiver a trabalhar mais rápido corriga o seu relógio para uma data antes da compilação erradamente o `make` não o irá compilar.

Ainda assim, podemos usar a condição de monotonia mesmo que os processadores possuam relógios que trabalham mais depressa, porque, na verdade, só precisamos de mudar a taxa com que as atualizações são feitas ao tempo em cada máquina. Esta tarefa é bastante fácil de ser cumprida em *software* sem que tenhamos de mudar a velocidade com que o relógio funciona — relembrar-se que $C_i(t) = \alpha H_i(t) + \beta$, onde estamos livres de escolher os valores de α e β .

Método de Cristian para a sincronização de relógios

Em 1989, Flaviu Cristian sugeriu o uso de uma máquina própria para servir o tempo a vários clientes numa rede, conectada a um dispositivo que servisse uma fonte de UTC, para sincronizar externamente vários sistemas computacionais. Neste sistema, após um pedido, um processo de servidor S fornece o tempo de acordo com o seu próprio relógio, como podemos ver na Figura 3.4.

◎ Flaviu Cristian

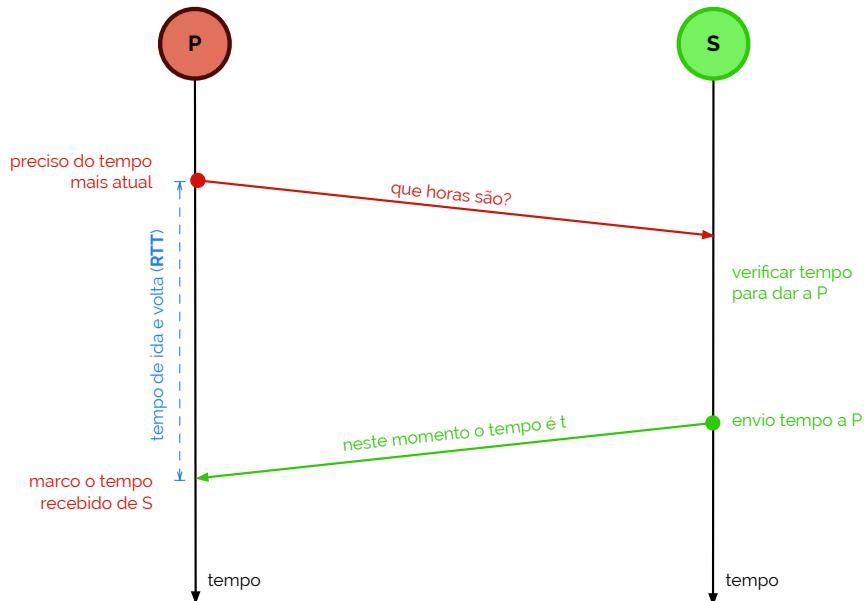


figura 3.4

Com a ideia atingida, Cristian observou que sendo que não existe majorante em relação a atrasos de transmissão num sistema assíncrono, o tempo de ida e volta (geralmente designado de **round-trip time** — RTT) para as mensagens trocas entre pares de processos são geralmente baixos — uma pequena fração de segundo. Acaba, então, por descrever este mesmo algoritmo como probabilístico, uma vez que o método consegue atingir sincronização somente se o tempo de ida e volta entre cliente e servidor for suficientemente baixo, dentro da precisão necessária.

round-trip time

Basicamente, e como podemos rever na Figura 3.4, o **método de Cristian** funciona da seguinte forma: de forma algo análoga a uma pessoa que necessita de saber o tempo e questiona a alguém que tenha um relógio, um processo p envia uma mensagem m_r ao servidor S com um pedido sobre o tempo atual, recebendo o tempo t numa mensagem m_t (t é inserido em m_t na última comunicação de S com p). O processo p deverá, então, gravar o tempo de ida e volta T_{round} que demorou entre o envio de m_r e a receção de m_t . Este tempo poderá ser medido com uma precisão algo elevada e a taxa de desvio do relógio interno for suficientemente baixa.

método de Cristian

Uma boa estimativa do tempo para o qual p deverá marcar o seu relógio é $t + T_{round} / 2$, o que assume que o tempo perdido é igualmente divisível antes e depois de S ter colocado o valor de t em m_t . Esta é uma consideração razoavelmente precisa, a não ser que as duas mensagens tenham sido despachadas sobre duas redes de computadores diferentes. Se o valor do tempo mínimo de transmissão min for conhecido ou puder ser estimado com segurança, então podemos determinar a precisão do resultado da seguinte forma: o ponto

mais cedo em que S poderá colocar o tempo em m_t é \min depois de p ter despachado m_r ; o último momento em que a mesma ação poderá ocorrer é \min antes de m_t chegar a p ; o tempo em S quando a mensagem de resposta chega é, então, definida no intervalo $[t + \min, t + T_{round} - \min]$; o tamanho deste intervalo de tempo é $T_{round} - 2\min$, pelo que a precisão é de $\pm(T_{round}/2 - \min)$.

Algoritmo de Berkeley

Em muitos algoritmos de sincronização de relógios o servidor de tempo é um agente passivo. Quando isto acontece há a necessidade de outras máquinas lhe efetuarem um pedido sobre o tempo atual, como foi o caso descrito no método de Cristian.

No sistema UNIX Berkeley uma aproximação completamente inversa foi tomada, fazendo com que o servidor de tempo fosse um agente ativo, ou seja, fosse este mesmo quem questionasse as várias outras entidades sobre o tempo, enviando-lhes, de seguida, um acerto em caso de necessidade. Este acerto não é nada mais nada menos do que o intervalo que existe entre o relógio de cada máquina cliente e a média dos resultados de todas as máquinas clientes.

Este método acaba por ser uma boa alternativa para efetuar a sincronização interna, ou seja, para aplicar em locais onde não há forma ou interesse de sincronizar ou comunicar com uma máquina externa que possa servir tempo UTC. Na Figura 3.5 podemos ver uma representação do funcionamento deste método com dois clientes P_1 e P_2 e um servidor S .

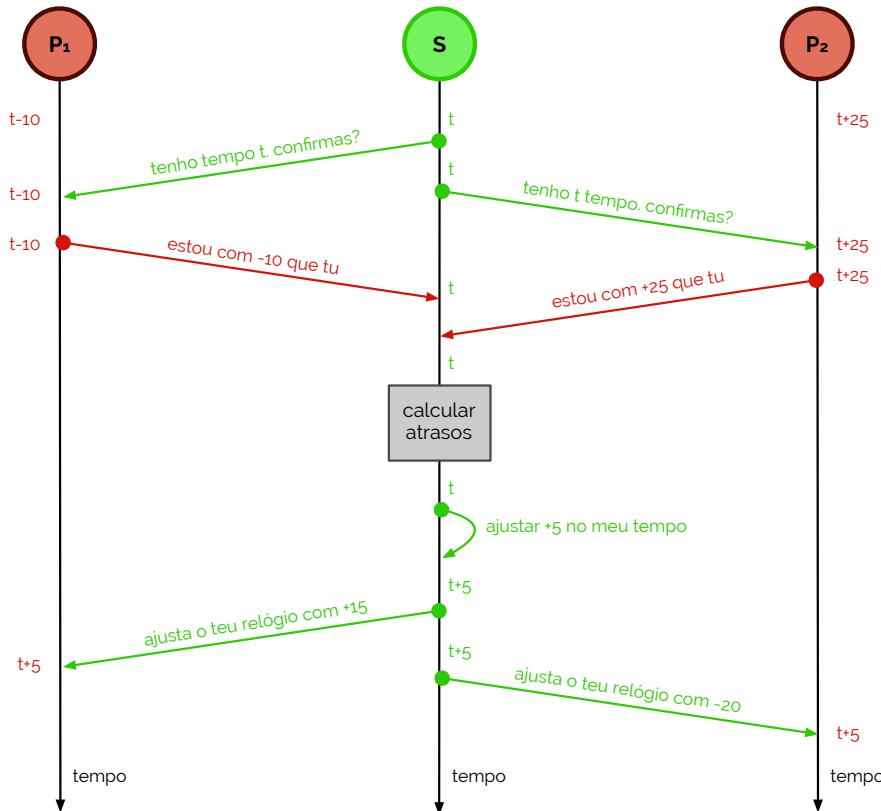


figura 3.5

Na Figura 2.5 podemos então verificar como é que o método de Berkeley funciona. Como já referimos, neste processo é o servidor, como entidade ativa, que interroga os vários clientes sobre que tempo atual é que cada um afirma possuir. Este cenário é algo semelhante a um conjunto de três amigos que pretendem sincronizar os seus relógios para fazer uma corrida. Um dos três encarrega-se de verificar as horas de cada um e acertar o três relógios de forma a que os três possam estar em sincronia. No fundo, depois dos vários clientes responderem com o deslocamento que possuem ao servidor, este deverá calcular os vários atrasos entre todas as máquinas e atribuir um conjunto de deslocamentos para cada

um dos sistemas computacionais, para que estes possam aplicar em busca de ficarem todas as entidades com os relógios sincronizados.

Note-se que é suficiente que todas as máquinas concordem todas num tempo comum. Não é essencial que este tempo também tenha de estar concordante com o tempo real anunciado globalmente a cada hora. Na nossa Figura 2.5 a máquina central (o servidor S) nunca será sujeito a uma alteração manual da sua hora, sendo que todas as entidades irão simplesmente ficar satisfeitas desde que todos tenham uma hora igual entre elas.

Protocolo de tempo de rede (NTP)

Os métodos anteriores (método de Cristian e de Berkeley) estão vocacionados a sincronização de relógios locais de máquinas interligadas numa rede local. Em 1995 foi criado um protocolo que permite a sincronização de relógios entre máquinas de uma rede global, denominado de **NTP** (sigla para *Network Time Protocol*). Este protocolo define assim uma arquitetura para um serviço de tempo e um protocolo para a distribuição de informação de tempo sobre a Internet.

NTP

O NTP tem como objetivos permitir que qualquer sistema computacional ligado à Internet possa acertar o seu relógio local através de uma fonte UTC, com uma precisão algo razável e efetuando o seu reacerto a um ritmo suficientemente elevado que impeça o surgimento de discrepâncias resultantes de desvios de relógio. Mais, com o NTP também se pretende que o serviço fornecido possa sobreviver a perdas de conectividade mais ou menos longe de servidores particulares, mantendo a sua disponibilidade permanente. Por fim, o NTP também procura fornecer proteção contra interferências com o serviço de tempo, quer de um modo malicioso como accidental, sendo que o serviço usa mecanismos de autenticação para verificar a fonte de cada dado temporal, proveniente de cada fonte, e os endereços de retorno das mensagens enviadas para esta.

A arquitetura deste serviço é bastante clara, sendo uma hierarquia bem estruturada constituída por um conjunto de servidores primários como conjunto de raízes, que obtêm um primeiro valor de tempo através de captadores de sinais rádio, onde os pulsos são emitidos ao tempo de um segundo UTC. Abaixo dos servidores primários temos um conjunto de servidores secundários, que acertam os seus relógios locais com servidores pertencentes ao estrato imediatamente acima na hierarquia. Note-se que os sistemas computacionais em cada estrato também podem coordenar a sua informação temporal com outros servidores no mesmo estrato para fornecer uma informação globalmente mais estável e robusta. [7]

Agora, o facto de estarmos perante uma estrutura hierárquica em que a informação é propagada das raízes para as suas folhas, a propagação será mais morosa quanto mais baixo estivermos. Isto faz com que o grau de incerteza da informação temporal aumente, uma vez que os erros introduzidos em cada etapa de sincronização são cumulativos. Para corrigir esta situação o NTP inclui um mecanismo de sub-árvore de acerto, que é reconfigurável de forma totalmente dinâmica. Neste cenário, sempre que um servidor primário deixa de poder aceder à sua fonte UTC, então passa para um estrato 2, tornando-se um servidor secundário. Por outro lado, sempre que o servidor usado para acerto de um sistema computacional de um dado estrato deixa de estar disponível, então outro será procurado ao invés deste.

Com o NTP as trocas de mensagens são realizadas sobre canais UDP através da Internet. Cada mensagem transporta dentro de si os *timesteps* dos eventos de mensagens mais recentes: são eles os tempos locais de quando a última mensagem NTP entre o par foi enviada e recebida e o tempo local de quando esta é transmitida. Através da Figura 3.6 podemos ver um exemplo de trocas de mensagens neste protocolo em particular.

Na Figura 3.6 podemos então ver a realização das operações entre um par de nós S_A e S_B para acerto do relógio local do nó S_B , se este pertencer a um estrato de valor mais elevado, ou para mútuo ajuste dos relógios locais de ambos os nós, se pertencerem ao mesmo estrato. Aqui, para além das marcas que já identificámos, ao receber uma mensagem, o nó destinatário toma nota também do tempo local da sua receção $t(B, i)$. Os quatro tempos

$t(B, i-3)$, $t(A, i-2)$, $t(A, i-1)$ e $t(B, i)$ são usados simultaneamente para se obter uma estimativa do deslocamento e da sua incerteza.

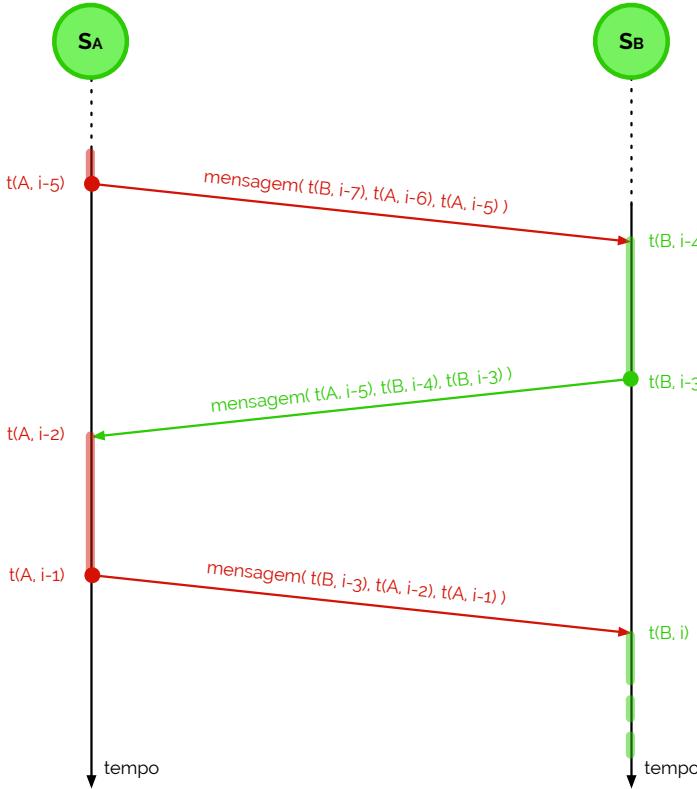


figura 3.6

Na Figura 3.6 também podemos verificar que o nó S_B supõe que os desvios do seu relógio local e do relógio local do nó S_A originam variações nos intervalos de tempo, respetivamente $t(B, 1) - t(B, i-3)$ e $t(A, i-1) - t(A, i-2)$ que podem ser consideradas desprezáveis. Exprimindo o deslocamento entre os tempos locais do nó S_A e do nó S_B como $\sigma = Ck_A(t) - Ck_B(t)$, a sua estimativa é modelada por (3.3).

$$\sigma_{\text{estimado}} = \frac{t(A, i-1) + t(A, i-2)}{2} + \frac{t(B, i) + t(B, i-3)}{2} \quad (3.3)$$

A incerteza associada com este valor no pior caso possível, Δ_{estimado} , resulta de se supor que o tempo de transmissão de uma das duas mensagens temporais sucessivas é mínimo, como podemos verificar em (3.4).

$$\Delta_{\text{estimado}} = \frac{t(A, i-2) - t(B, i-3) + t(B, i) - t(A, i-1)}{2} - t_{\min} \quad (3.4)$$

Os servidores NTP, depois, aplicam um algoritmo de filtragem estatística a pares sucessivos (σ_{estimado} , Δ_{estimado}), que estima o deslocamento σ e calcula a qualidade desta estimativa como uma quantidade estatística denominada por **filtro de dispersão**. Note-se que um filtro com valores relativamente altos toma-se por conter dados não fiáveis.

filtro de dispersão

Sincronização de processos através de relógios lógicos

Por fim, dos vários tipos de relógios que abordámos ao início houve um tipo que ficou por esclarecer: é ele o relógio lógico. Um **relógio lógico** é, assim, um relógio cujo tempo é percebido em termos do fluxo informativo.

relógio lógico

Do ponto de vista de qualquer processo, os eventos são ordenados unicamente pelos tempos atribuídos pelo relógio local. Contudo, e tal como Lamport constatou, uma vez que não podemos sincronizar com perfeição relógios inseridos num sistema distribuído, não podemos usar o conceito de tempo físico para tentar identificar a ordem de qualquer par arbitrário de eventos que nele ocorrem.

De uma forma geral, podemos usar um esquema que em tudo é semelhante ao modelo de causalidade física, mas aplicado a sistemas distribuídos. Nós, Humanos, usamos constantemente o modelo de causalidade, uma vez que sabemos que há eventos que ocorrem porque outros ocorreram por si só ou por vias de um outro. A esta relação, Lamport deu o nome de relação **ocorreu-antes** e definiu-a algebricamente tendo em conta dois aspectos: se dois acontecimentos ocorreram num mesmo processo p_i (com $i = 1, 2, \dots, N$), então estes ocorreram numa determinada ordem com que p_i os observa — a ordem algebricamente designada de \rightarrow , como definiremos mais abaixo; sempre que uma mensagem é enviada entre processos, o evento de enviar a mensagem ocorre sempre antes de receber a mesma.

Lamport acaba por chamar a este procedimento **ordenação parcial**, este, que é obtido pela generalização destes dois relacionamentos da relação ocorreu-antes. Também é vulgar identificar este procedimento como sendo a relação de ordenação causal ou de potencial ordenação causal.

A relação ocorreu-antes, \rightarrow , poderá então ser identificada da seguinte forma:

- se \exists processo $p_i : e \rightarrow e'$, então $e \rightarrow e'$;
- para qualquer mensagem m , envio(m) \rightarrow receção(m), onde envio(m) é o evento de envio de uma mensagem m e receção(m) é o evento de receber uma mensagem m ;
- se e, e' e e'' são eventos tais que $e \rightarrow e'$ e $e' \rightarrow e''$, então $e \rightarrow e''$.

Considere a Figura 3.7 onde podemos ver um exemplo de eventos a ocorrerem em três processos, com ordenação causal.

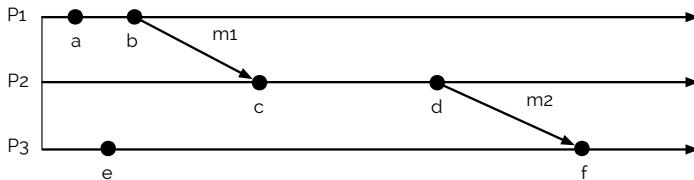


figura 3.7

Pela Figura 3.7 podemos verificar que existe uma relação de causalidade $a \rightarrow b$, dado que tais eventos ocorrem por esta mesma ordem no processo P_1 , da mesma forma que se valida $c \rightarrow d$. Mais, também podemos dizer que, entre processos, $b \rightarrow c$ é válido, identificando um envio de uma mensagem m_1 do processo P_1 para o processo P_2 , tal como $d \rightarrow f$. Combinando estas relações, por transitividade, temos que $a \rightarrow f$.

Contudo, na Figura 3.7 também podemos verificar que nem todos os eventos estão relacionados por relações de ocorreu-antes. Por exemplo, $a \not\rightarrow e$ e $e \not\rightarrow a$, uma vez que estes ocorrem em diferentes processos e sem qualquer tipo de cadeia de mensagens de interação entre ambos. Enquanto que nos outros casos dizemos os pares de acontecimentos são **sequenciais**, neste caso dizemos que os acontecimentos tais como a e e não são ordenados parcialmente, pelo que são **concorrentes**, algebricamente identificados da seguinte forma: $a \parallel e$.

Lamport, entretanto, criou um dispositivo a que designou de **relógio lógico** para explicitar numericamente a ordenação resultante da relação de ocorreu-antes. Um relógio lógico redefine-se assim como um contador local de acontecimentos, monotonamente crescente, que não tem qualquer associação direta com o tempo real (físico). [8]

© Leslie Lamport

ocorreu-antes

ordenação parcial

sequenciais
concorrentes

relógio lógico

Aqui, cada processo p_i mantém o seu próprio relógio L_i que usa para aplicar a marcação dos **timestamps de Lamport**. Denotemos assim o *timestamp* de Lamport por $L_i(e)$ para o evento e no processo p_i e, por $L(e)$ o *timestamp* do evento e em qualquer que seja o processo.

timestamps de Lamport

Este algoritmo funciona da seguinte forma: para capturar a relação de ocorreu-antes os processos atualizam os seus relógios lógicos e transmitem os seus valores dos seus relógios em mensagens. Primeiramente L_i é incrementado antes de cada evento ser lançado no processo p_i . Quando um processo p_i envia a mensagem m , esta carregará o valor $t = L_i$. Ao receber (m, t) um processo p_j calcula $L_j := \max(L_j, t)$ e aplica-lhe o incremento $L_j := L_j + 1$ antes de marcar o tempo no evento receção(m).

Consideremos novamente a mesma cadeia de eventos em três processos da Figura 3.7, agora na Figura 3.8, onde aplicamos os *timestamps* de Lamport.

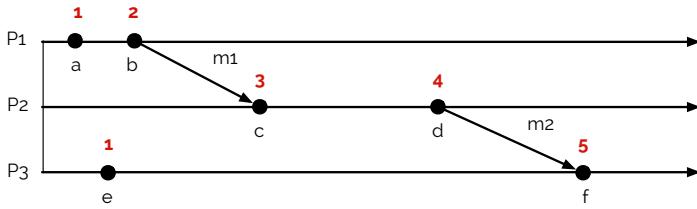


figura 3.8

Embora incrementemos os relógios por uma unidade, poderíamos ter escolhido qualquer outro valor positivo, sendo que para qualquer um verifica-se que a relação de $e \rightarrow e' \implies L(e) < L(e')$. Contudo, o contrário não é verdadeiro, uma vez que não podemos $e \rightarrow e'$ olhando simplesmente para $L(e) < L(e')$. Tendo isto em conta, na Figura 3.8 cada um dos processos possui o seu relógio lógico inicializado a 0. Os valores de relógio são atribuídos aos vários eventos que estão imediatamente à frente associados a estes (os quais são adjacentes). Note-se, por exemplo, que $L(b) > L(e)$, mas $b \parallel e$. [2]

Lamport mostrou, no entanto, que grupos de acontecimentos conexos podem ser sujeitos a uma operação de ordenação total e percebidos na mesma ordem por processos residentes em diferentes nós de uma máquina paralela, se relógios lógicos do tipo por ele definido forem usados na geração de marcas temporais incluídas nas mensagens trocadas.

Sejam assim e_j , com $j = 0, 1, \dots, K-1$, acontecimentos associados ao envio de mensagens m_j entre os processos p_i , com $i = 0, 1, \dots, N-1$. Dizemos que os acontecimentos e_j podem ser **totalmente ordenados** se e só se puder estabelecer uma correspondência biúnica entre cada acontecimento e um ponto da reta numérica através de uma propriedade que lhes está associada (como a marca temporal incluída na mensagem).

totalmente ordenados

Dado que, com tudo consolidado, nada impede que os *timestamps* associados a duas mensagens distintas sejam iguais, temos que $\text{timestamp}(m_p) = \text{timestamp}(m_q)$, com $p, q = 0, 1, \dots, K-1$ e $p \neq q$, Lamport definiu uma estrutura de dados que pode ser designado de **timestamp estendido**, como podemos ver em (3.5).

timestamp estendido

$$\left(\text{timestamp}(m_p), \text{id}(m_p) \right) \quad (3.5)$$

A representação em (3.5) consiste no par ordenado formado pela marca temporal da mensagem e pela identificação do seu remetente, impondo a regra de ordenação de (3.6).

$$\begin{aligned} \left(\text{timestamp}(m_p), \text{id}(m_p) \right) < \left(\text{timestamp}(m_q), \text{id}(m_q) \right) \Leftrightarrow \\ \Leftrightarrow \text{timestamp}(m_p) < \text{timestamp}(m_q) \vee \\ \text{timestamp}(m_p) = \text{timestamp}(m_q) \wedge \\ \text{id}(m_p) < \text{id}(m_q) \end{aligned} \quad (3.6)$$

Admita-se agora que, numa dada aplicação alvo para sistemas distribuídos, existem N cópias de uma mesma região de dados localizadas em regiões geograficamente distintas. Cada cópia é acedida por processo p_i (com $i = 0, 1, \dots, N-1$) específico. Cada processo p_i efetua sobre a sua cópia operações de escrita e de leitura que levam a uma eventual alteração do conteúdo da região de dados. Fica então a questão, como organizar as operações de modo a garantir que as diversas cópias se mantêm permanentemente sincronizadas, isto é, que apresentam sempre o mesmo valor?

Haver uma sincronização permanente das cópias significa que sempre que se pretenda modificar o valor de um registo da cópia local, o comando de alteração seja propagado primeiro aos processos que gerem o acesso a todas as outras cópias e que a ordem de execução dos comandos de alteração seja sempre a mesma em todo o lado. Para que tal suceda, tem que se supor que os processos p_i mantêm-se sempre em operação correta, isto é, que não ocorrem falhas catastróficas na sua execução e que não há perdas de mensagens.

O algoritmo proposto por Lamport consiste em cada processo p_i , ao verificar que tem que realizar um comando de alteração, começar por construir uma mensagem com toda a informação sobre a operação onde inclui uma marca temporal com o valor do seu relógio local. Uma vez enviada a mensagem a todos os membros do grupo (incluindo ao próprio), quando esta se recebe, cada processo p_i acerta o seu relógio local segundo as regras de acordo de Lamport e coloca a mensagem recebida numa fila de espera ordenada por ordem crescente da marca temporal estendida. Logo de seguida é enviada uma mensagem de *acknowledgment* a todos os membros do grupo, incluindo ao próprio, sendo que os comandos residentes em cada fila de espera local são executados por cada processo p_i na ordem estabelecida quando e só quando houver o consentimento de todos os processos do grupo, isto é, quando tiverem sido recolhidas a totalidade das mensagens de *acknowledgement* associadas. [8]

Relógios lógicos vetoriais

Um outro tipo de relógio lógico foi introduzido por Mattern e Fidge com o objetivo de resolver a limitação existente no relógio lógico escalar proposto por Lamport, em que a partir de $L(e) < L(e')$ não conseguimos inferir que $e \rightarrow e'$. Um **relógio vetorial** para um sistema de N processos é um array de N inteiros. Cada processo mantém o seu próprio vetor-relógio V_i que é usado para gravar a marca temporal local dos eventos.

De forma análoga às marcas temporais de Lamport, os processos carregam os vetores de *timestamps* nas mensagens que carregam entre processos. O algoritmo funciona da seguinte forma: inicialmente $V_i[j] = 0$, com $i, j = 1, 2, \dots, N$; mesmo antes de p_i marcar temporalmente um evento, marca-se $V_i[i] := V_i[i] + 1$; p_i inclui o valor $t = V_i$ em cada mensagem que marca; quando p_i recebe um *timestamp* t numa mensagem, marca $V_i[j] := \max(V_i[j], t[j])$, para $j = 1, 2, \dots, N$.

Pelo que podemos concluir, a ideia desta implementação foi manter, além do próprio relógio local, toda a informação possível, ainda que desatualizada, sobre a evolução dos potenciais relógios locais dos outros processos do sistema e capturar desta forma a causalidade potencial que possa existir entre acontecimentos que se desenrolam em processos residentes de uma máquina paralela.

Em suma, sejam V e V' marcas temporais vetoriais, a sua comparação é realizada de acordo com as regras seguintes: $V = V'$ se e só se $V[j] = V'[j]$ para um $j = 1, 2, \dots, N$; $V \leq V'$ se e só se $V[j] \leq V'[j]$ para um $j = 1, 2, \dots, N$; $V < V'$ se e só se $V[j] \leq V'[j]$ e $V[j] \neq V'[j]$. Note-se ainda que agora $e \rightarrow e' \implies V_i(e) < V_j(e')$ também é válido no seu converso $V_i(e) < V_j(e') \implies e \rightarrow e'$.

Na Figura 3.9 podemos ver o mesmo exemplo que da Figura 3.8 agora com a implementação de relógios lógicos vetoriais. Aqui podemos ver exemplos os de $V(a) < V(f)$, onde se reflete o facto de que $a \rightarrow f$. De forma semelhante, podemos dizer que dois eventos são concorrentes através da comparação dos seus *timestamps*. Por exemplo, $c \parallel e$ pode ser comprovado pelo facto de nem $V(c) \leq V(e)$ nem $V(e) \leq V(c)$.

- Friedemann Mattern
- Colin Fidge
- relógio vetorial

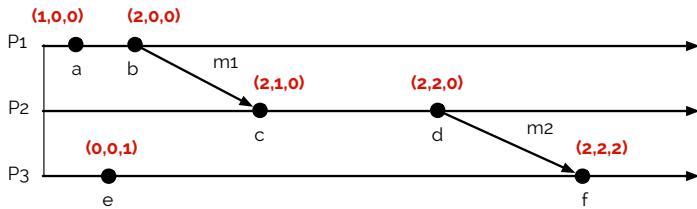


figura 3.9

Estas marcas temporais têm as desvantagens de, em comparação às de Lamport, de ocuparem mais espaço de memória e um *payload* de mensagem que é proporcional a N , o número de processos.

4. Comunicação entre Pares

Nesta nova secção introduzimos um conjunto de algoritmos cujo objetivo é fundamental em sistemas distribuídos: coordenar um conjunto de processos em termos de ações ou arranjar concordância entre um ou mais dados. Introduzimos também uma nova topologia de implementação lógica em sistemas distribuídos, que foi popular nos anos '90 e que será uma revisita da disciplina de Fundamentos de Redes (a3s1) — o *token-ring*.

Autorização de acesso centralizada

Consideremos que estamos perante uma arquitetura em que um conjunto de processos pretende escrever num objeto partilhado — como é que podemos garantir que os vários processos não se “atropelam” uns aos outros em busca do tão precioso recurso?

Uma forma de ponderar sobre este assunto é através da aplicação de um **processo guardião**, isto é, um processo específico que controla o acesso ao objeto partilhado. Basicamente, neste tipo de topologia, sempre que um dos processos-par p_i com $i = 0, 1, \dots, N-1$, pretender aceder ao objeto partilhado, envia uma mensagem de pedido de acesso ao processo guardião p_N , solicitando autorização e aguardando por uma mensagem de permissão de acesso garantida por este último, que lhe será enviada como resposta.

processo guardião

Neste cenário um outro efeito poderá ocorrer. Se na altura nenhum outro processo-par estiver a aceder ao objeto partilhado, o processo p_N responde de imediato, caso contrário insere o pedido numa fila de espera de pedidos de acesso. Quando um processo termina o que tem a fazer com um determinado recurso larga o seu acesso exclusivo. Por fim, o processo guardião, depois desta última ação, atribui o recurso ao primeiro processo que o pediu e se encontra em fila de espera.

Podemos ver uma representação destas interações na Figura 4.1.

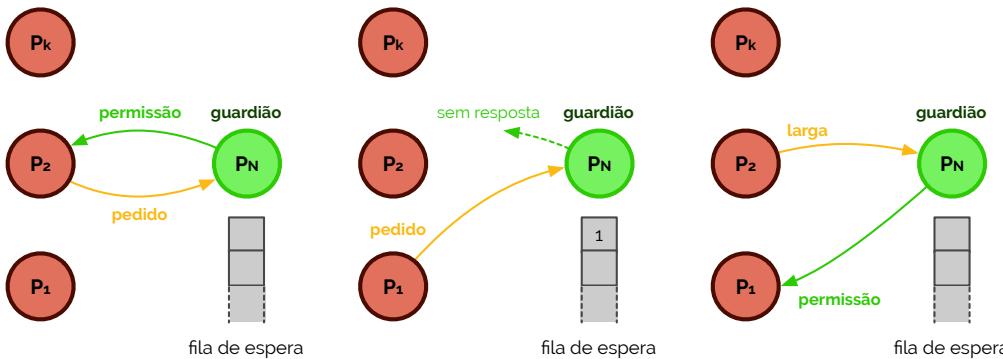


figura 4.1

O que tentamos fazer com este tipo de procedimento, no fundo, é criar algoritmos para exclusão mútua. Assumindo que o nosso sistema é assíncrono, que os processos não falham e que as entregas de mensagens são fiáveis, então qualquer mensagem é eventual-

mente enviada puramente intacta, exatamente uma vez. Como requisitos para as nossas ações pretendemos ir de encontro ao estabelecimento de três condições importantes:

- **segurança** – pelo menos um processo poderá executar dentro de uma região crítica de cada vez;
- **vida** – pedidos para entrar e sair de uma região partilhada eventualmente são bem sucedidos;
- **ordenação causal** – se um pedido para entrar numa região partilhada ocorrer antes de outro, então a entrada na região é garantida na mesma ordem.

No caso da segunda condição (condição de vida), note-se que esta implica estar livre de *deadlocks* e de situações de adiamento indefinido. [2]

Algoritmo com base em anéis lógicos (token-ring)

Uma das formas mais fáceis de aplicar e distribuir critérios de exclusão mútua entre os N processos sem que processos adicionais surtam efeito é através da aplicação de modelos de arquitetura de **aneis lógicos**. Estas implementações requerem que cada processo p_i tenha um canal de comunicação estabelecido com o próximo processo presente no anel $p_{(i+1) \bmod N}$.

A ideia neste caso passa pelo facto da exclusão ser realizada e conferida através da obtenção de um **testemunho** sob a forma de uma mensagem que é passada de processo em processo numa única direção dentro do anel. A topologia em anel poderá ser descartada fisicamente nas interligações entre os vários computadores, mas em termos lógicos está presente.

Se um processo não requerer a entrada numa região crítica quando recebe o testemunho (vulgarmente denominado de **token**), então passa-o imediatamente para o processo seguinte (seu vizinho). Por outro lado, um processo que necessite do testemunho espera até o obter, mantendo-o quando o obtiver. Para sair da região crítica, o processo deverá, depois, voltar a enviar o testemunho para o anel, para que este possa seguir para o seu vizinho.

A organização dos vários processos pode ser representada conforme é visível na Figura 4.2.

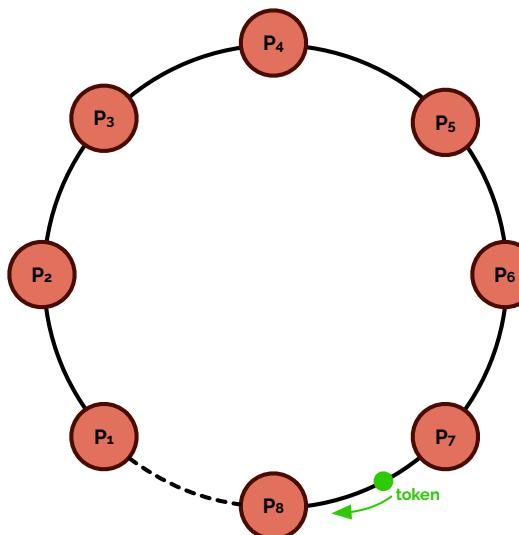


figura 4.2

Olhando para a Figura 4.2 podemos verificar que ambas condições de segurança e de vida, conforme as enunciámos anteriormente, são válidas neste modelo de arquitetura, sendo que, contudo, o testemunho não tem de ser obtido segundo uma relação causal, isto porque os processos podem trocar mensagens independentemente da rotação do testemunho.

Note-se que a implementação deste algoritmo possui uma desvantagem: consome muita largura de banda (à exceção do período em que um processo se encontra dentro de uma região crítica), sendo que os vários processos enviam mensagens dentro do anel mesmo quando nenhum processo necessita de acesso a uma região crítica. O atraso experienciado por um processo que pretende aceder a uma região crítica é sempre entre 0 mensagens (quando recebe o *token*) e N mensagens (quando o testemunho acaba de passar). Para sair de uma região crítica apenas há a necessidade de libertar uma única mensagem. O tempo de atraso de sincronização entre a saída de um processo numa região crítica e a entrada de um próximo, por outro lado, está sempre entre 1 e N transmissões de mensagens.

Ordenação total dos acontecimentos

Ricart e Agrawala propuseram um algoritmo que implementa o acesso com exclusão mútua a um objeto partilhado por parte de N processos utilizando o relógio lógico de Lamport para a ordenação dos vários pedidos de acesso. O objetivo é, então, garantir que todos os processos-par efetuam a mesma ordenação dos acontecimentos associados aos pedidos de acesso ao objeto partilhado, obtendo-se, portanto, um consenso geral.

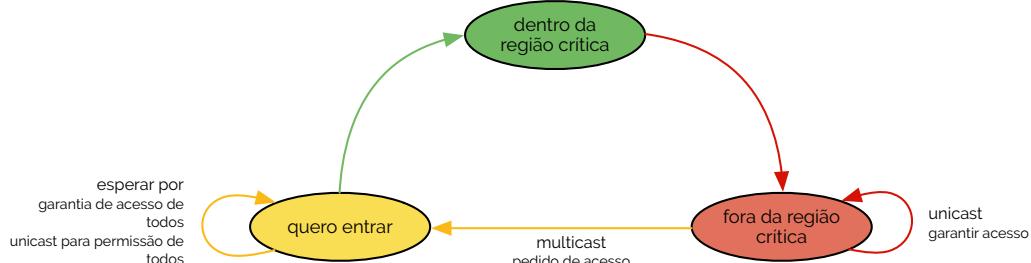
As mensagens trocadas entre os vários processos incluem um campo com a marca temporal do remetente, o que permite que, aquando da receção da mensagem, o destinatário acerte o seu relógio local e seja assim explicitado o nexo de causalidade entre pares de acontecimentos.

Para se impor a ordenação total dos acontecimentos é, entretanto, construída uma marca temporal estendida que contém como parte inteira o *timestamp* da mensagem associada e como parte fracionária, a identificação do remetente.

Na Figura 4.3 [9] podemos ver um diagrama ilustrativo da máquina de estados que permite descrever a construção deste algoritmo.

◎ Glenn Ricart
◎ Ashok Agrawala

figura 4.3



Em termos de modelação em código teríamos o seguinte representado no Código 4.1, para representar a ordenação total dos acontecimentos.

```

// inicialização
estado = fora_da_Regiao_critica;
mensagemDePedidoPronta = falso;

// processo i entra na região crítica
estado = quero_entrar;
numeroDePedidosAceites = 0;
minhaMensagemDePedido = multicast(pedidoDeAcesso);
mensagemDePedidoPronta = verdadeiro;
esperar até (numeroDePedidosAceites == N-1);
estado = dentro_da_Regiao_critica;

// processo i sai da região crítica
estado = fora_da_Regiao_critica;
mensagemDePedidoPronta = falso;
enquanto (!vazio(filaDePedidos)) {
    identificacao = getId(tirarDaFila(filaDePedidos));
    unicast(identificacao, acessoGarantido);
}

// processo i recebe pedido de acesso de processo j
se (estado == fora_da_Regiao_critica) {
    identificacao = getId(mensagemDePedido);
    unicast(identificacao, acessoGarantido);
} caso contrário, se (estado == dentro_da_Regiao_critica) {
  
```

código 4.1

```

    colocarEmFila(filaDePedidos, mensagemDePedido);
} caso contrário {
esperar até (mensagemDePedidoPronta);
se (getExtTimestamp(minhaMensagemDePedido) < getExtTimestamp(mensagemDePedido)) {
    colocarEmFila(filaDePedidos, mensagemDePedido);
} caso contrário {
    unicast(identificacao, acessoGarantido);
}

// processo i processa autorizações de acesso
numeroDePedidosAceites += 1;

```

Mecanismo eletivo e minimização do número de mensagens

Como tivemos oportunidade de verificar anteriormente, o mecanismo que expusemos anteriormente consome muita largura de banda, uma vez que os vários processos enviam mensagens dentro do anel mesmo quando nenhum processo necessita de acesso a uma região crítica.

Em 1985, Maekawa mostrou que o acesso aos objetos partilhados por parte que de qualquer um dos processos não requer a autorização de todos eles. Os processos, assim, podem ser organizados em grupos parcelares e obter apenas autorização dos processos pertencentes ao mesmo. Neste cenário, a exclusão mútua (portanto, a consequente eliminação de condições de corrida no acesso às regiões críticas) é garantida desde que esses grupos não sejam mutuamente exclusivos.

O princípio subjacente é garantir que um processo só accede ao objeto partilhado quando este tiver obtido autorização de todos os processos pertencentes ao seu grupo através de um processo de votação.

Maekawa associou, assim, um conjunto de votação V_i com cada processo p_i (com $i = 1, 2, \dots, N$), onde $V_i \subseteq \{p_1, p_2, \dots, p_N\}$. Os conjuntos V_i são escolhidos de forma a que, para todo $i, j = 1, 2, \dots, N$, $p_i \in V_i$ (o processo p_i faça parte do grupo de votação), $V_i \cap V_j \neq \emptyset$ (dois processos p_i e p_j fazem parte, pelo menos, de um mesmo grupo), $|V_i| = K$ (por questões de justiça, cada processo possui um conjunto de votação do mesmo tamanho) e cada processo p_j está contido em M grupos de V_i .

Foi mostrado que uma solução ótima, que minimiza K e permite que vários processos consigam exclusão mútua, tem um valor de K muito semelhante a \sqrt{N} e $M = K$ (de forma a que cada processo se encontre em tantos grupos de votação quantos elementos existir em cada um destes conjuntos). Contudo, note-se que não é nada trivial calcular os conjuntos ótimos R_i . Ainda assim, através de uma aproximação, uma simples derivação do conjunto R_i tal que $|R_i| \sim 2\sqrt{N}$ permite-nos que coloquemos os processos numa matriz $\sqrt{N} \times \sqrt{N}$, tendo V_i como sendo a união das colunas e das linhas que contêm p_i .

O algoritmo de Maekawa pode ser visto no Código 4.2, onde para obter a entrada numa região crítica, um processo p_i envia um pedido de mensagem para todos os membros K do seu grupo V_i (incluindo a si próprio). Note-se que p_i não poderá entrar na região crítica até receber todas as K mensagens. Quando um processo p_j em V_i recebe uma mensagem de pedido, este terá de enviar uma mensagem de resposta imediata, a não ser que o seu estado interno seja de “quero entrar” ou já tenha respondido com “voto” desde a última mensagem de largada. Caso contrário, a mensagem de pedido irá para uma fila de espera de pedidos de acesso (pela ordem de chegada das mesmas), não respondendo ainda. Quando um processo recebe uma mensagem de largada, então o elemento que está na cabeça da fila é removido (isto se a fila não estiver vazia) e envia um pedido (com um “voto”) na sua resposta. Para sair da região crítica, o processo p_i deverá enviar um pedido de largada para todos os membros K de V_i (incluindo-se a si próprio).

```

// inicialização
estado = fora_da_Regiao_Critica;
votou = false;

```

• Mamoru Maekawa

código 4.2

```

// processo i entra na região crítica
estado = quero_entrar;
numeroDePedidosAceites = 0;
multicast(pedidoDeAcesso) para todos os processos em Vi;
esperar até (numeroDePedidosAceites == #(Vi));
estado = dentro_da_Regiao_critica;

// processo i sai da região crítica
estado = fora_da_Regiao_critica;
multicast(pedidoDeLargada) para todos os processos em Vi;

// processo i recebe pedido de acesso de processo j
se ((estado != fora_da_Regiao_critica) && !votou) {
    identificacao = getId(mensagemDePedido);
    unicast(identificacao, acessoGarantido);
    votou = verdadeiro;
} caso contrário {
    colocarEmFila(filaDePedido, mensagemDePedido);
}

// processo i recebe indicação de terminação de acesso de processo j
se (!vazio(filaDePedido)) {
    identificacao = getId(removerDaFila(filaDePedido));
    unicast(identificacao, acessoGarantido);
    votou = verdadeiro;
} caso contrário {
    votou = falso;
}

// processo i processa autorizações de acesso
numeroDeProcessosAceites += 1;

```

Este algoritmo conforme exposto no Código 4.2 [9], porém, não é correto. Neste contexto há situações em que face a condições de corrida prevalecentes vai ocorrer *deadlock*. Consideremos três processos p_1 , p_2 e p_3 , com $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$ e $V_3 = \{p_3, p_1\}$. Se os três processos pedirem acesso à região crítica de forma concorrente, então é possível que p_1 responda a si próprio e bloqueie p_2 , p_2 responda a si próprio e bloqueie p_3 e p_3 responda a si próprio e bloqueie p_1 . Cada processo recebeu uma de duas respostas, e nenhum consegue avançar.

O algoritmo, contudo, poderá ser modificado de forma a tornar-se livre da ocorrência de *deadlocks*. Para que isto possa acontecer, a fila de espera de pedidos de acesso terá de receber os objetos tendo em conta a relação de ocorreu-antes, de forma a que a nossa condição anterior de ordenação total dos acontecimentos fica satisfeita. [2]

Há situações que exigem a seleção de um processo pertencente a um grupo de processos para realizar uma tarefa bem definida. Dado que todos os processos são conceptualmente idênticos, qualquer um de entre eles pode, em princípio, ser escolhido.

Para responder a esta questão temos **algoritmos de eleição**, que não são nada mais, nada menos, que algoritmos para escolher um processo único para a execução de um papel em particular. Por exemplo, numa variante de uma arquitetura de cliente-servidor em termos de exclusão mútua (como o representado na Figura 4.1), um “servidor” poderá ser escolhido entre os vários processos p_i (com $i = 1, 2, \dots, N$) que precisem de utilizar a região crítica. Um algoritmo de eleição é necessário neste caso, sendo mais-que-essencial que todos os processos concordem com qualquer decisão que seja efetuada neste contexto.

Dizemos que um processo pede uma eleição quando este inicia uma ação que indica um processo de eleição. Um processo por si só não poderá pedir mais do que uma eleição de cada vez, mas no próprio princípio os N processos poderão pedir N eleições concorrentes. Em qualquer instante de tempo, um processo p_i pode tanto ser um **participante** — o que significa que este faz parte de um processo de eleição — ou um **não-participante** — se não fizer parte de nenhum processo de eleição.

Um requisito importante que deverá ser cumprido é que a escolha do processo eleito deverá ser única. Por exemplo, dois processos poderão decidir independentemente que um processo coordenante falhou e pedir, os dois, eleições.

Sem perdas de generalização, pede-se também que o processo eleito seja escolhido como sendo aquele que possui o maior identificador. O **identificador** pode ser qualquer valor considerado útil, desde que seja único e totalmente ordenável. Por exemplo, poderemos eleger um processo com a menor carga computacional requerida, através do uso do par

algoritmos de eleição

participante

não-participante

identificador

$(1/carga, i)$ como identificador, onde $carga > 0$ e o índice i do processo é usado para ordenar identificadores com a mesma carga.

Cada processo p_i (com $i = 1, 2, \dots, N$) possui uma variável de nome $elected_i$, que conterá o identificador do processo elegido. Quando o processo se torna participante num procedimento de eleição, então marca a sua variável $elected$ para um valor especial \perp (este valor será usado para denotar que ainda não está definido).

Os nossos requisitos, concretamente, no decorrer de qualquer execução do algoritmo são os seguintes:

- **segurança** – um processo participante p_i tem $elected_i = \perp$ ou $elected_i = P$, onde P é escolhido como o processo que não saiu no fim da corrida com o maior identificador;
- **vida** – todos os processos p_i participam e, eventualmente, nenhum deles assumem o valor de \perp como a variável $elected_i$ ou saem.

Note-se que podem existir processos p_j que ainda não são participantes, pelo que o seu valor de $elected_j$ possuem o identificador do processo anteriormente elegido.

Medimos assim o desempenho de um algoritmo de eleição através da largura de banda total da rede despendida (que será proporcional ao número total de mensagens enviadas) e pelo tempo de ida e volta do mesmo: o tempo de transmissão do número de mensagens serializadas entre a inicialização e a terminação de uma única corrida.

Um dos algoritmos de eleição que nos será mais interessante aplicar e estudar é o algoritmo de aplicação em anéis lógicos. Este algoritmo, denominado de algoritmo de Chang e Roberts, é aplicável num conjunto de processos organizados num anel lógico, em que cada processo p_i possui um canal de comunicação com o processo seguinte em $p_{(i+1) \bmod N}$ e todas as mensagens são enviadas segundo o sentido dos ponteiros do relógio dentro do anel. Consideremos que o sistema é assíncrono e que não há falhas de transmissão de mensagens. O objetivo deste algoritmo é então eleger um único processo a que denominaremos de **coordenador**, sendo este o processo com o maior identificador.

coordenador

Inicialmente, todos os processos são marcados como não-participantes numa eleição. neste momento qualquer processo poderá dar início a uma eleição. Quando um pretende iniciar o procedimento, marca-se a si mesmo como participante, colocando o seu identificador numa mensagem que envia ao seu vizinho mais próximo, dentro do anel, no sentido dos ponteiros do relógio.

Quando um processo recebe uma mensagem de eleição, compara-a com o identificador que deixa na sua própria mensagem. Se aquele que chegar possuir um identificador maior, então reenvia a mensagem para o seu vizinho. Se a mensagem que chegar com um identificador menor e o recetor for não-participante, então substitui-se o identificador da mensagem e reencaminha-a; contudo, note-se que a mensagem não será reenviada caso o receptor seja participante. Num reenvio de uma mensagem de eleição, seja qual for o caso, o processo marca-se como participante.

Se, no entanto, o identificador recebido for o do recetor em si, então o identificador do processo deverá ser o maior, passando a ser o coordenador (ou líder). O líder marca-se, então, como não-participante uma vez mais e envia uma mensagem de elegido para o seu vizinho, anunciando a sua eleição e propagando a sua identificação.

Quando um processo p_i recebe uma mensagem de elegido marca-se como não-participante, colocando a sua variável $elected_i$ com o identificador na mensagem e, a não ser que seja o novo líder, reenvia a mensagem para o seu vizinho.

É fácil verificar que a condição de segurança que apontámos anteriormente se verifica. Todos os identificadores são comparados, sendo que um processo deverá receber o seu próprio identificador antes de enviar uma mensagem de elegido. Para quaisquer dois processos, aquele que tiver um identificador maior não passará o identificador do outro. Deste modo é impossível que ambos recebam a sua própria identificação de volta.

Por conseguinte, a segunda condição (de vida), é também válida, sendo que se garante a travessia do anel por completo. Note-se como os estados dos não-participantes e dos par-

ticipantes são usados de forma a que mensagens duplicadas quando dois processos iniciam o procedimento de eleição ao mesmo tempo são existentes o mais cedo possível, e sempre antes do resultado do “vencedor” ser anunciado.

Se um único processo iniciar o procedimento de eleger um líder, então o pior caso de desempenho ocorre quando um vizinho anterior possui um identificador mais alto de toda a rede de processos. Neste caso um total de $N - 1$ mensagens serão necessárias de ser enviadas até atingir este vizinho, o qual não irá anunciar a sua eleição até que o identificador tenha completado mais uma volta completa, ou seja, só depois de mais N mensagens. A mensagem de eleito é então enviada N vezes, cumprindo $3N - 1$ mensagens no seu todo. O tempo de volta é, também, $3N - 1$, uma vez que estas mensagens são enviadas de forma sequencial.

Na Figura 4.4 podemos ver um exemplo de eleição num anel lógico em progresso, onde a mensagem de eleição possui o identificador 24, mas o processo 28 irá substituir o seu valor quando a mensagem lhe chegar. Note-se que neste procedimento a eleição foi iniciada pelo processo 17, sendo os participantes na eleição apenas os processos a verde.

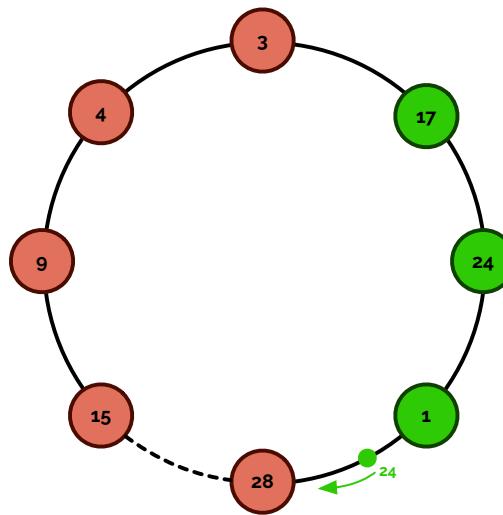


figura 4.4

Quando um processo não recebe qualquer mensagem de eleição (*acknowledge*) dentro de um período prescrito, considera-se o processo de identificação presentemente mais baixa no sistema e assume o papel de líder, enviando uma mensagem de eleito com a sua identificação a todos os processos para os informar do facto. Esta alteração ao algoritmo original foi uma modificação introduzida por Garcia-Molina, acabando o algoritmo por se denominar de **algoritmo de bully**. [2]

© Héctor García-Molina
algoritmo de bully

5. Consistência e Replicação

Num sistema distribuído uma das questões mais pertinentes que se podem colocar, dada a divisão lógica de um sistema simples numa arquitetura modulada por um conjunto de diversas entidades é a **consistência** dos dados que nele existem.

consistência

Por questões de tolerância a falhas e até mesmo por questões de disponibilidade do sistema, uma das formas de garantir eficácia no fornecimento de um serviço poderá passar por gerar mecanismos de redundância, sendo um deles a **replicação** de informação ao longo de um conjunto de máquinas distribuídas numa rede de computadores. Contudo, neste tipo de cenário aplica-se também a igual questão: como é que podemos garantir a consistência dos dados em locais disjuntos que deverão apresentar o mesmo tipo de informação em qualquer intervalo de tempo?

replicação

Regiões de armazenamento distribuídas

É precisamente por esta última questão que começamos a nossa análise. Para isso façamos, inicialmente, uma pequena introdução ao conceito da distribuição de uma região de armazenamento de dados de um sistema. Na Figura 5.1 podemos ver uma pequena representação deste tipo de arquitetura.

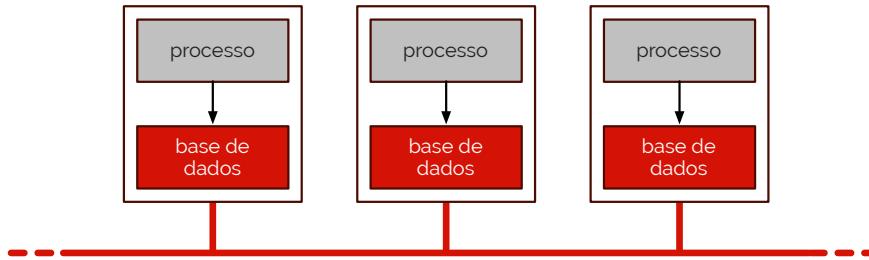


figura 5.1

Uma região de armazenamento distribuída pode ser entendida como uma memória partilhada, uma base de dados, ou um sistema de ficheiros, distribuídos sobre um conjunto de nós de processamento.

Neste conceito, assume-se que cada nó de processamento tem acesso direto a uma cópia local de toda a região e só existem dois tipos de operações, o tipo de escrita (que origina uma alteração dos dados) e o tipo de leitura (que constitui todas as operações restantes). Quando uma operação de escrita ocorre, esta é propagada a todas as cópias existentes. Mais, o acesso à região de armazenamento distribuída é efetuado em paralelo por parte dos processos residentes nos diferentes nós de processamento.

Modelos de consistência

Definimos como **modelo de consistência** um conjunto de regras que, se obedecidas pelos processos intervenientes, conduzem a que uma região de armazenamento distribuída funcione corretamente. O problema, contudo, coloca-se em definir o que se entende por “corretamente”, aplicando o conceito de uma forma sistemática.

modelo de consistência

Cada modelo que iremos abordar distingue-se por especificar, explicitamente, a gama de valores que uma operação de leitura pode devolver face a operações de escrita que tiveram anteriormente ocorrido. Quando se admite paralelismo de operações, os modelos de consistência pertencem a uma classe designada de **centrada nos dados**.

centrada nos dados

Numa primeira definição de critério de funcionamento correto, seja $op_{i,0}, op_{i,1}, op_{i,2}, \dots$ com $i = 0, 1, \dots, n$ uma qualquer sucessão de operações de escrita e de leitura efetuadas pelo processo i sobre a região de armazenamento distribuída. Cada operação é caracterizada pelo seu tipo, argumentos e valor de retorno, tomando o princípio de que deverá ser síncrona, ou seja, que a próxima operação só pode ser executada quando a imediatamente anterior tiver sido concluída.

Admita-se que os n processos, cada um com a sua sucessão de operações, acede em paralelo à região de armazenamento distribuída. Se se tratasse de uma região concentrada ao invés de uma região distribuída, as sucessões de operações dos diferentes processos interpenetrar-se-iam de algum modo em cada instância originando uma sucessão global de operações do tipo $op_{2,0}, op_{2,1}, op_{0,0}, op_{1,1}, op_{2,2}, op_{2,3}, op_{0,1}, op_{1,2}, \dots$ [10]

O critério de funcionamento correto da região de armazenamento distribuída é agora definido por referência a uma ou mais sucessões canónicas globais de interpenetração das sucessões parcelares que estabelecem o padrão de correção de funcionamento.

Note-se que esta sucessão padrão é meramente virtual, não tendo necessariamente que ocorrer nos acessos locais a uma dada réplica. O seu papel é servir meramente de certificação àquilo que se designa de correção de funcionamento para uma dada instância de acesso paralelo à região de armazenamento distribuída. Face aos resultados percebidos numa dada instância, a inexistencia de uma sucessão canónica que cumpra as pro-

propriedades do modelo de consistência considerado, permite concluir que o modelo não é válido.

Consistência estrita

Um dos tipos de consistência que pode ser implementado chama-se **consistência estrita**. Este método diz que toda a operação de leitura do registo x devolve um valor correspondente ao resultado da operação de escrita mais recente em x .

consistência estrita

Esta situação, sendo apenas verificada em processadores únicos, é uma situação que poderá ser considerada como ideal, sendo que não é introduzido qualquer tipo de efeito de diferença temporal nesta. Contudo, este modelo não é passível de ser implementado no contexto de sistemas distribuídos, sendo que a noção de acontecimento mais recente é ambígua, dada a velocidade finita da propagação do sinal imposta aos sistemas físicos.

Como consequência deste modelo temos que os relógio locais aos diferentes nós de processamento não podem ser sincronizados com uma precisão absoluta e, portanto, não se pode falar na existência de um relógio global que ordene os acontecimentos. Mais, o tempo de execução das operações de escrita e de leitura é necessariamente não nulo.

Nestas condições a ordenação de acontecimentos não pode ser estabelecida em termos de um padrão temporal único. Isto poderá ser visto na Figura 5.2, onde um processo P_0 inicialmente escreve um valor a numa variável x (no tempo t_0). Em t_1 , se quisermos ler o valor de a , como acontece por vias do processo P_1 , temos que o cenário é estritamente estrito se e só se conseguir ler o valor a aquando do acesso à variável x .

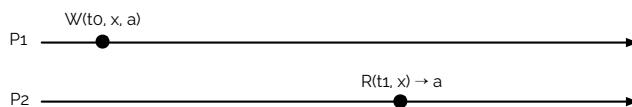


figura 5.2

Enquanto que na Figura 5.2 a região de armazenamento diz-se estritamente consistente uma vez que segue uma sucessão canónica em que, como se $t_0 < t_1$ então $W_0(t_0, x, a) - R_1(t_1, x) \rightarrow a$. Já na Figura 5.3 temos uma região de armazenamento que não se diz estritamente consistente, uma vez que não existe sucessão canónica que reflita os resultados. Isto acontece porque há uma primeira leitura do valor de x cujo retorno não coincide com o valor da sua última escrita, o que viola a condição da consistência estrita.

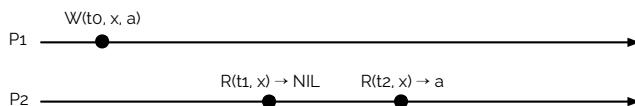


figura 5.3

Linearizabilidade

O modelo de linearizabilidade pressupõe que existe algum procedimento de sincronização dos relógios locais que possibilita uma ordenação total dos acontecimentos que ocorrem no sistema distribuído (aproximação à noção de relógio global). O objetivo é garantir que os processos intervenientes obtêm sempre a informação mais atualizada possível.

Diz-se assim que no modelo de linearizabilidade, o acesso paralelo a um registo x é visto por todos os processos envolvidos como se as operações efetuadas tivessem sido ordenadas numa sucessão única bem definida onde as operações mantêm a ordem cronológica de execução percebida localmente.

Transmite-se a ideia que, para todas as sucessões de operações efetuadas em paralelo pelos processos intervenientes, existe uma sucessão canónica de execução global das operações sobre uma execução única, consistente com a realização cronológica das operações, tal como esta é percebida localmente. [10]

Na Figura 5.4 podemos ver um levantamento de processos que foram executados.

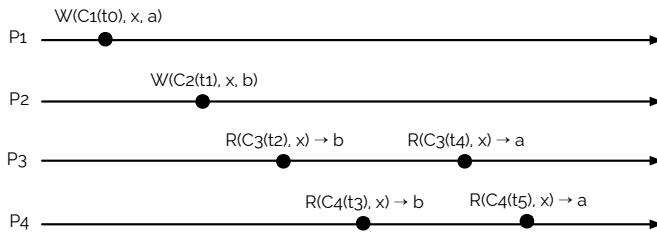


figura 5.4

Olhando para a Figura 5.4 podemos ver a execução de algumas operações sobre uma variável x . Podemos concluir, através desta, que podemos obter uma região de armazenamento que será considerada linearizável se considerarmos a seguinte ordem canónica de (5.1).

$$\begin{aligned} C_2(t_1) &< \{C_3(t_2), C_4(t_3)\} < C_1(t_0) < \{C_3(t_4), C_4(t_5)\} \Rightarrow \\ \Rightarrow W_2(C_2(t_1), x, b) - R_{3,3}(-, x) \rightarrow b - W_1(C_1(t_0), x, a) - R_{2,3}(-, a) \rightarrow a \end{aligned} \quad (5.1)$$

Pelo contrário, se o nosso evento inicial for $C_1(t_0)$, já não iremos conseguir obter uma sucessão canónica que refletirá os resultados obtidos, pelo que consideraremos que a região de armazenamento não é linearizável.

Consistência sequencial

Com o modelo de **consistência sequencial** temos que o acesso paralelo a um registo x é visto por todos os processos envolvidos como se as operações efetuadas tivessem sido ordenadas numa sucessão única bem definida onde as operações de cada processo individual mantêm a ordem por que foram localmente executadas.

consistência sequencial

A primeira constatação é que o tempo, quer global, quer local, não desempenha um papel explícito neste enquadramento. A segunda é que, em resultado disso, não é necessário estabelecer-se a ordenação total dos acontecimentos que ocorrem no sistema distribuído. A ideia que se pretende transmitir, assim, é que, para todas as sucessões de operações efetuadas em paralelo pelos processos intervenientes, existe uma sucessão canónica de execução global das operações sobre uma região concentrada virtual que fornece a cada processo uma imagem de execução única, consistente com a realização sucessiva das operações locais.

Na Figura 5.5 podemos ver um caso em que temos uma região de armazenamento que consideramos ser sequencialmente consistente, uma vez que conseguimos criar uma sucessão canónica: $W_1(x, b) - R_{2,3}(x) \rightarrow b - W_0(x, a) - R_{2,3}(x) \rightarrow a$.

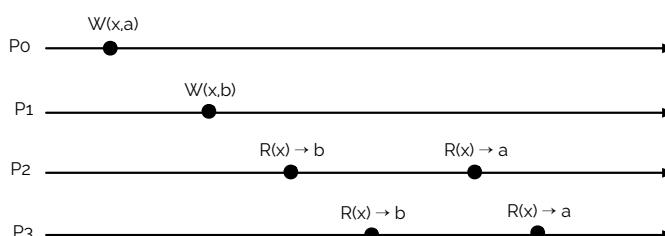


figura 5.5

Já na Figura 5.6 podemos verificar que não há como considerar uma região de armazenamento sequencialmente consistente, sendo que não conseguimos criar uma sucessão canónica que reflita os resultados obtidos da figura. Depois de executar uma operação de escrita da variável x com o valor b não há forma de lhe escrever a para ler a e voltar a ler b mais adiante, sendo que este já terá sido reescrito entretanto por a .

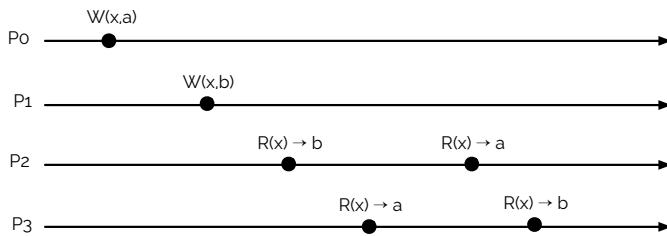


figura 5.6

Consistência causal e de fila

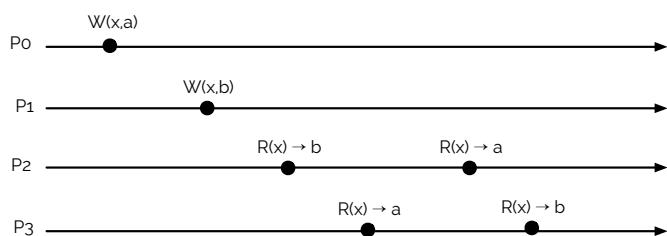
Por fim, existem mais dois modelos de consistência que nos são importantes referir. O primeiro, o modelo de **consistência causal** defende que o acesso paralelo a um registo x é visto por todos os processos envolvidos como se as operações efetuadas, que mantêm entre si uma relação causal, tivessem sido ordenadas numa sucessão única bem definida. As operações restantes, ditas paralelas, podem ser percebidas em ordem diferente pelos diferentes processos intervenientes.

Para se compreender melhor o que é estabelecido por este critério, o mais fácil é definir o que se entende por causalidade. Consideramos assim que as operações executadas sucessivamente pelo mesmo processo formam entre si uma cadeia potencialmente causal, sendo uma operação de leitura sempre relacionada causalmente com uma operação de escrita que forneceu o valor que foi lido, independentemente do (ou dos) processo(s) que executa(ram) a(s) operação(s).

A ideia transmitida agora é que, para todas as sucessões de operações efetuadas em paralelo pelos processos intervenientes, a existência de uma sucessão canónica de execução global das operações sobre uma região concentrada virtual que fornece a cada processo uma imagem de execução única, consistente com a realização sucessiva das operações locais, está restrita ao subconjunto de operações que mantêm entre si uma relação causal.

Só para estas é imposto um critério de consistência sequencial. Todas as restantes operações podem ser percebidas em qualquer ordem pelos diferentes processos intervenientes.

Na Figura 5.7 podemos voltar a verificar um esquema, desta vez, onde mostramos um caso de exposição de uma região de armazenamento que é causalmente consistente, sendo que como as duas operações de escrita são paralelas, então a sucessão canónica é formada por zero operações.



consistência causal

figura 5.7

Já no caso da Figura 5.8 podemos ver um caso em que a região de armazenamento não é causalmente consistente, sendo que há uma relação causal entre as operações de escrita, fazendo com que não haja sucessão canónica que reflita os resultados.

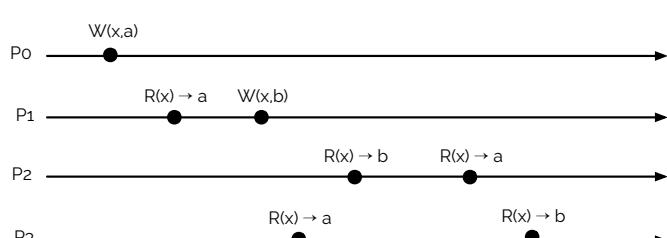


figura 5.8

Em termos do modelo de **consistência de fila** temos que aqui o acesso paralelo a um registo x é visto por todos os processos envolvidos como se as operações efetuadas por um mesmo processo mantivessem a ordem por que foram localmente executadas. Note-se que, neste caso, a existência de uma sucessão canónica de execução global das operações sobre uma região concentrada virtual que fornece a cada processo uma imagem de execução única, desaparece completamente.

Desde que a ordem de execução local das operações seja percebida por todos os processos intervenientes, e está-se naturalmente a falar apenas das operação de escrita, as operações de leitura deixam de ter qualquer importância porque não originam nenhum efeito. A interpenetração das sucessões parcelares pode ser qualquer e vista de um modo diferente por cada um dos processos.

Na Figura 5.9 podemos ver um caso em que a região de armazenamento é consistente em fila, dado que como as duas operações de escrita são executadas por processos diferentes, a ordenação local destas operações é trivial.

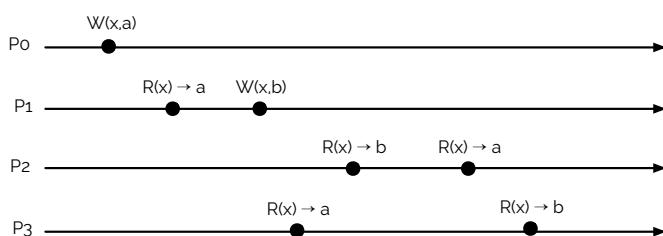


figura 5.9

Já na Figura 5.10 a região de armazenamento não é consistente em fila, uma vez que a ordenação local entre as operações de escrita do processo P_1 não é percebida enquanto tal pelo processo P_3 .

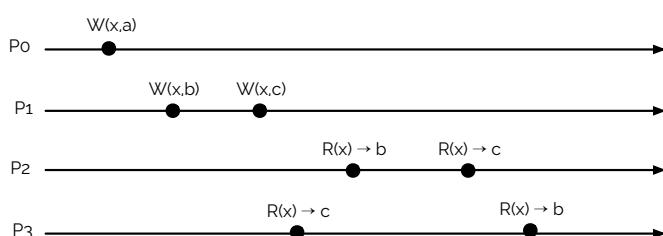


figura 5.10

E assim terminam os apontamentos de Sistemas Distribuídos (a4s2), onde se tentou elaborar a projeção de um sistema comum para múltiplas máquinas, com diferentes modelos de arquitetura próprios e onde se estudou o impacto das decisões em termos de tempo e consistência de dados.

1. Noções Básicas de Sistemas Distribuídos

Processos e fios de execução (threads)	2
Organização de um programa multithreaded.....	4
Implementação de threads em Java	4
Introdução ao conceito de sistemas distribuídos.....	7
Princípios gerais de concorrência	9

2. Modelos de Sistemas Distribuídos

Modelos de arquitetura (cliente-servidor, pares e publicador-subscritor)	14
Modelos fundamentais.....	20

3. Sincronização

Como é que medimos o tempo?	22
Sincronização de relógios locais.....	24
Método de Cristian para a sincronização de relógios	26
Algoritmo de Berkeley	27
Protocolo de tempo de rede (NTP)	28
Sincronização de processos através de relógios lógicos	29
Relógios lógicos vetoriais.....	32

4. Comunicação entre Pares

Autorização de acesso centralizada	33
Algoritmo com base em anéis lógicos (token-ring)	34
Ordenação total dos acontecimentos	35
Mecanismo eletivo e minimização do número de mensagens.....	36

5. Consistência e Replicação

Regiões de armazenamento distribuídas	40
Modelos de consistência	40
Consistência estrita	41
Linearizabilidade	41
Consistência sequencial.....	42
Consistência causal e de fila.....	43

As referências abaixo correspondem às várias citações (quer diretas, indiretas ou de citação) presentes ao longo destes apontamentos. Tais referências encontram-se dispostas segundo a norma IEEE (as páginas Web estão dispostas de forma análoga à de referências para livros segundo a mesma norma).

- [1] A. R. Borges, “Slides da disciplina de Sistemas Distribuídos - Introductory Concepts”, Universidade de Aveiro, 2018.
- [2] G. Coulouris et al., *Distributed Systems: Concepts and Design*, 5 ed. New Jersey: Pearson, 2013.
- [3] A. R. Borges, “Slides da disciplina de Sistemas Distribuídos - Concurrency I”, Universidade de Aveiro, 2018.
- [4] A. R. Borges, “Slides da disciplina de Sistemas Distribuídos - Concurrency II”, Universidade de Aveiro, 2018.
- [5] A. R. Borges, “Afternoon At The Races”, Universidade de Aveiro, 2018.
- [6] A. Tannenbaum et al., *Distributed Systems: Principles and Paradigms*, 3 ed. New Jersey: Pearson, 2015.
- [7] A. R. Borges, “Slides da disciplina de Sistemas Distribuídos - System Models”, Universidade de Aveiro, 2018.
- [8] A. R. Borges, “Slides da disciplina de Sistemas Distribuídos - Sincronização”, Universidade de Aveiro, 2018.
- [9] A. R. Borges, “Slides da disciplina de Sistemas Distribuídos - Comunicação entre Pares”, Universidade de Aveiro, 2018.
- [10] A. R. Borges, “Slides da disciplina de Sistemas Distribuídos - Consistência e Replicação”, Universidade de Aveiro, 2018.

Apontamentos de Sistemas Distribuídos

1^a edição - junho de 2018

sd

Autor: Rui Lopes

Agradecimentos: professor António Rui Borges

Todas as ilustrações gráficas são obra de Rui Lopes e as imagens são provenientes das fontes bibliográficas divulgadas.



apontamentos

© Rui Lopes 2018 Copyright: Pela Creative Commons, não é permitida a cópia e a venda deste documento. Qualquer fraude será punida. Respeite os autores e as suas marcas. Original - This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/deed.en_US.