***Concorrência em sistemas distribuídos e concorrência em sistema centralizado ou shared-memory concurrency***

A concorrência em sistemas distribuídos apresenta problemas diferentes da concorrência *shared-memory* que é a que existe entre os processos e *threads* (fios de execução) em execução no mesmo computador. Nesta, os múltiplos *threads* em execução no mesmo processo têm acesso ao mesmo espaço de endereçamento e, portanto, os dados podem facilmente passar de um *thread* para outro. Uma variável ou ponteiro que é válido para um *thread* também o é para outro.

A concorrência também existe nos sistemas distribuídos porque computadores distintos podem estar a executar programas em paralelo. Como cada computador, neste caso, está a executar o seu sistema operativo, com um espaço de endereçamento próprio, não existe memória partilhada. Computadores diferentes só podem comunicar enviando mensagens, de uns para os outros, através da rede.

***Confiabilidade relacionada com sistema envolvendo múltiplos computadores (nós) distribuídos***

Um sistema envolvendo múltiplos nós distribuídos (sistema distribuído), se for bem desenhado, pode ser mais fiável que um sistema que consista de um único computador. Um computador (nó) pode falhar e precisar de ser reiniciado de vez em quando mas, utilizando múltiplos nós, um nó pode continuar a fornecer o serviço aos utilizadores enquanto outro está em baixo ou a reiniciar.

Por outro lado, essa fiabilidade só é maior se o sistema distribuído for bem desenhado e os desafios são maiores porque passam também a existir mais pontos potenciais de falha.

***A falha, no paralelo entre centralizado (de RAM) e distribuído***

Quando há uma falha de um componente num computador, normalmente não se espera que ele continue a funcionar apesar disso, simplesmente bloqueia ou desliga-se. O *software*, neste caso, habitualmente não é escrito para poder lidar com falhas, por exemplo de RAM.

Ao contrário, num sistema distribuído, frequentemente, prevê-se que algumas partes do sistema falhem e, ainda assim, o resto continue a funcionar, e o *software* é escrito explicitamente a contar com essas situações.

Por exemplo, se há uma falha de um nó num sistema distribuído (será uma falha parcial), os restantes nós deverão, ainda assim, ser capazes de continuar a disponibilizar o serviço: *fault tolerance*.

***Latências introduzidas na comunicação (interação) entre nós***

Em sistemas distribuídos, há um atraso variável entre o envio e a recepção de uma mensagem (latência) que está relacionado com o método e o meio em que se processa a comunicação entre nós.

**Chamada local e RPC, aspetos relevantes em cada um dos modelos**

Na chamada local de uma função, espera-se que a resposta seja imediata e que a execução esteja sempre disponível. O RPC (*Remote Procedure Call*) é um método que procura dar a aparência que a função chamada se está a executar localmente quando, na realidade, o código dessa função não existe localmente. Essa função pertence a outro programa que está em execução num nó diferente e a chamada é feita através de um *stub* disponibilizado pelo *framework*. O processo de codificação/descodificação entre os dois nós, efectuado pelo mecanismo RPC, é designado *marshalling*.

**A insolubilidade do problema dos dois generais e a sua abordagem no modelo RPC**

O problema dos dois generais, aplicado no contexto das comunicações em sistemas distribuídos, pode resumir-se como: uma acção que tenha que ser executada por dois intervenientes remotos que necessitem de estar coordenados, nunca o poderá ser com total confiança de ambos de que o outro também executará a sua parte. Isso acontece porque nas trocas de mensagens entre os dois intervenientes há sempre a necessidade de um *acnowledge* de cada mensagem e de um *acnowledge* do *ackowledge*, e por aí fora… Caso se perca uma dessas mensagens e não chegue ao destino, a acção terá que ser iniciada por uma das partes, assumindo o risco de que a outra parte não esteja preparada.

06-05-2021