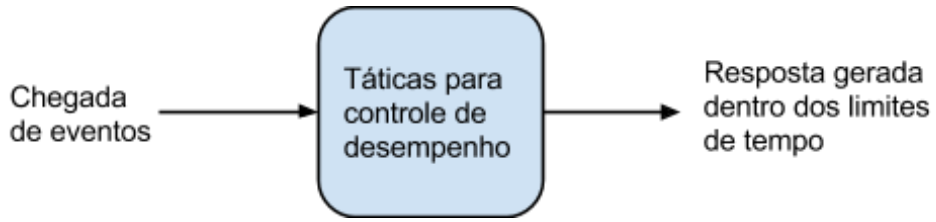
Tradução de trechos do Livro:

BASS, Len, CLEMENTS, Paul, KAZMAN, Rick. Software Architecture in Practice. 3. ed. Addison-Wesley, 2012

Tradução realizada por Torsten Nelson

Táticas de desempenho

No capítulo 4 vimos que o objetivo das táticas de desempenho é gerar uma resposta a um evento que chega ao sistema dentro de alguma restrição de tempo. Pode ser um único evento ou uma sequência. Ele é o gatilho para se realizar alguma computação. O evento pode ser a chegada de uma mensagem, o término de um intervalo de tempo, a detecção de uma mudança significativa no estado do ambiente do sistema, e assim por diante. O sistema processa os eventos e gera uma resposta. Táticas de desempenho controlam o tempo entre a chegada de um evento e a geração de uma resposta a ele.



After an event arrives, either the system is processing on that event or the processing is blocked for some reason. This leads to the two basic contributors to the response time: resource consumption and blocked time.

Após a chegada de um evento, o sistema estará processando o evento ou o processamento estará bloqueado por algum motivo. Isto leva aos dois fatores principais que contribuem para aumentar o tempo de resposta: uso de recursos e tempo bloqueado.

1. *Uso de recursos*. Recursos incluem CPU, armazenamento de dados, banda de comunicação em redes e memória, mas podem também incluir entidades definidas pelo sistema sendo projetado. Por exemplo, buffers devem ser gerenciados e o acesso a seções críticas deve ser sequencial. Os eventos podem ser de tipos variados (conforme enumerado anteriormente) e cada tipo passa por uma sequencia de processamento. Por exemplo, uma mensagem é gerada por um componente, é enviada pela rede, e chega a outro componente. Ela é então colocada em um buffer; transformada de alguma forma; processada de acordo com algum algoritmo; transformada em saída; colocada em um buffer de saída; e enviada para outro componente, para outro sistema, ou para o usuário. Cada uma dessas fases contribui para a latência total do processamento daquele evento.

2. *Tempo bloqueado*. Uma computação pode ser impedida de usar um recurso devido à concorrência por ele, ou devido ao recurso estar indisponível, ou porque a computação depende do resultado de outras computações que ainda não terminaram.
    - *Disputa por recursos*. A figura acima mostra eventos chegando no sistema. Estes eventos podem estar em um feixe único ou em múltiplos feixes. A existência de múltiplos feixes disputando o mesmo recurso ou eventos diferentes no mesmo feixe disputando o mesmo recurso contribui para a latência. Em geral, quanto maior a disputa por recursos, maior a chance de se introduzir latência. No entanto, isso depende de como a disputa é arbitrada e de como solicitações individuais são tratadas pelo mecanismo de arbitragem.
    - *Disponibilidade de recursos*. Mesmo na ausência de disputas, a computação não pode prosseguir se um recurso não estiver disponível. A indisponibilidade pode ser causada pelo fato do recurso estar offline, por falha do componente, ou por algum outro motivo. Qualquer que seja o caso, o arquiteto deve identificar lugares onde a indisponibilidade do recurso possa causar uma contribuição significativa à latência.
    - *Dependência em outras computações.* Uma computação pode ter que esperar porque precisa sincronizar com os resultados de outra computação ou porque está aguardando pelos resultados de outra computação que ela própria iniciou. Por exemplo, pode estar lendo informação de duas fontes diferentes. Se essas duas fontes forem lidas sequencialmente, a latência será maior do que se forem lidas em paralelo.

Com esses conceitos, nos voltamos a nossas três categorias táticas: demanda por recursos, gerência de recursos e arbitragem de recursos.

Demanda por recursos

Feixes de eventos são a fonte da demanda por recursos. Duas características da demanda são o tempo entre eventos em um feixe de recursos (o quão frequentemente um pedido é feito em um feixe) e a quantidade do recurso consumido em cada pedido.
Uma tática para reduzir a latência é reduzir a quantidade de recursos necessária para se processar um feixe de eventos. Algumas formas de se fazer isso são as seguintes:

- Aumentar a eficiência computacional. Um passo no processamento de um evento ou mensagem é a aplicação de algum algoritmo. Melhorar os algoritmos utilizados em regiões críticas reduzirá a latência. Pode-se muitas vezes trocar um recurso pelo outro. Por exemplo, pode-se armazenar dados intermediários em um repositório, ou podem ser gerados novamente, dependendo da disponibilidade do tempo e espaço. Essa tática é normalmente aplicada ao processador mas também é eficaz quando aplicada a outros recursos, como o disco.
- Reduzir o overhead computacional. Se não há pedido por um recurso, as necessidades de processamento são reduzidas. O uso de intermediários (tão importantes para a modificabilidade) aumenta os recursos consumidos no

processamento do feixe de eventos. Dessa forma, removê-los melhora a latência. Este é um *tradeoff* clássico entre modificabilidade e desempenho.

Outra tática para se reduzir a latência é reduzir o número de eventos processados. Isto pode ser feito de duas formas:

- *Gerenciar a taxa de eventos*. Se for possível reduzir a frequência de amostragem das variáveis do ambiente, pode-se reduzir a demanda. Isso pode ser possível se o sistema foi superdimensionado. Em outros casos uma taxa de amostragem desnecessariamente alta é usada para estabelecer períodos harmônicos entre múltiplos feixes, isto é, algum feixe em um conjunto de feixes de eventos é superamostrado para que os feixes possam ser sincronizados.
- Controlar a taxa de amostragem. Se não há controle sobre a chegada de eventos gerados externamente, os pedidos enfileirados podem ser amostrados em uma taxa menor, possívelmente resultando em perda de pedidos.

Outras táticas para se reduzir ou gerenciar a demanda envolvem o controle do uso de recursos

- *Tempos de execução limitados*. Envolve colocar um limite máximo no tempo de execução usado para se responder a um evento. Às vezes isso faz sentido, às vezes não. Para algoritmos iterativos que dependem de dados, limitar o número de iterações é um método para se limitar os tempos de execução.
- *Filas de tamanho limitado*. Isto controla o número máximo de eventos enfileirados e, consequentemente, os recursos usados para se processar esses eventos.

Gerência de recursos

Apesar da demanda por recursos não ser controlável, a gerência desses recursos afeta os tempos de resposta. Algumas táticas de gerência de recursos são:

- *Introduzir paralelismo*. Se recursos puderem ser processados em paralelo, o tempo que se passa bloqueado pode ser reduzido. O paralelismo pode ser introduzido ao se processar diferentes feixes de eventos em *threads* diferentes, ou ao se criar *threads* adicionais para processar diferentes conjuntos de atividades. Uma vez que o paralelismo é introduzido, é importante alocar as threads aos recursos de forma apropriada (balanceamento de carga) para explorar ao máximo o paralelismo.
- *Manter múltiplas cópias dos dados ou computações*. Clientes em um padrão cliente-servidor são réplicas da computação. O propósito das réplicas é reduzir a contenção que ocorreria se todas as computações acontecessem em um servidor central. *Caching* é uma tática na qual dados são duplicados para se reduzir a contenção. Como os dados do cache são geralmente cópias de dados existentes, manter as cópias consistentes e sincronizados se torna uma responsabilidade que o sistema precisa assumir.
- Aumentar os recursos disponíveis. Processadores mais rápidos, memória adicional e redes mais rápidas têm o potencial para reduzir a latência. O custo é normalmente um obstáculo para essa tática, mas aumentar os recursos é certamente uma tática válida para se reduzir a latência.

Arbitragem de recursos

Sempre que há disputa por um recurso, o recursos deve ser escalonado. Processadores são escalonados, buffers são escalonados e redes são escalonadas. O objetivo do arquiteto é entender as características de cada recurso e escolher a estratégia de escalonamento que é compatível com seu uso esperado.
Uma política de escalonamento tem duas partes: uma atribuição de prioridades e o despacho. Todas as políticas de escalonamento atribuem prioridades. Em alguns casos a atribuição é simples, como primeiro-entra/primeiro-sai. Em outros casos, pode ser ligada à importância do pedido. Critérios que competem entre si pelo escalonamento incluem o uso ótimo de recursos, a importância dos pedidos, a minimização da latência, a maximização do throughput, entre outros. O arquiteto precisa estar ciente desses critérios possivelmente conflitantes e o efeito que a tática escolhida tem no seu atendimento.
Um feixe de eventos de alta prioridade pode ser despachado somente se o recurso para o qual ele está sendo atribuido está disponível. Às vezes isso depende em se preemptar o usuário atual do recurso. Opções possível de preempção são as seguintes: pode ocorrer a qualquer instante; pode ocorrer apenas em pontos específicos de preempção; e processos em execução não podem ser preemptados. Algumas políticas comuns de escalonamento são:

1. *Primeiro-entra/Primeiro-sai* (*first-in/first-out*, ou *FIFO*). Filas FIFO tratam todos os pedidos por recursos igualmente e os satisfazem em ordem de chegada. Uma possibilidade em filas FIFO é que um pedido ficará preso atrás de outro que leva muito tempo para gerar uma resposta. Desde que todos os pedidos sejam semelhantes, isso não é problema, mas se alguns pedidos tem prioridade maior do que outros, isso pode ser problemático.
2. *Escalonamento de prioridade fixa*. Nesse caso se atribui uma prioridade a cada fonte de pedidos por recursos, e se atribui os recursos nessa ordem de prioridade. Essa estratégia garante serviço melhor para pedidos de alta prioridade, mas admite a possibilidade de que um pedido de baixa prioridade mas importante levará um tempo arbitrariamente longo para ser atendido, porque está preso atrás de uma série de pedidos de alta prioridade. Três estratégias comuns de priorização são:
    - *importância semântica*. A cada feixe é atribuida uma prioridade de acordo com alguma característica de domínio da tarefa que o gerou. Esse tipo de escalonamento é usado em sistemas mainframe onde a característica do domínio é o momento do início da tarefa.
    - *monotônico por tempo limite*. Esse é um tipo de atribuição de prioridade que atribui prioridades maiores a feixes com limites de tempo de execução menores. Essa política é usada quando feixes de prioridades diferentes com limites de tempo real diferentes precisam ser escalonados.
    - *monotônico por taxa*. Esse um tipo de atribuição para feixes periódicos que atribui prioridade maior a feixes com períodos menores. Essa política em um caso especial da monotônica por tempo limite, mas é mais conhecida e mais suportada por diferentes sistemas operacionais.
3. Escalonamento de prioridades dinâmicas:

    - Round-robin. Essa é estratégia que ordena os pedidos e então, a cada possibilidade de atribuição, atribui o recurso ao próximo pedido em ordem. Uma forma especial de round-robin é um executivo cíclico onde as possibilidade de atribuição acontecem em intervalos de tempo fixos.
    - tempo limite mais próximo primeiro. Essa política atribui prioridades baseado no conjunto de pedidos pendentes com o tempo limite mais próximo.
4. Escalonamento estático. Um escalonamento executivo cíclico é uma estratégia onde os pontos de preempção e a sequência de atribuições do recurso são determinados offline.

As táticas para desempenho são sumarizadas na figura abaixo.