Boas (e más) práticas de desenvolvimento *web*

Carlos Duarte
cgd[ ]sdf-eu.org
Maio 2003, rev 2

Introdução

Este texto foca aspectos de desenvolvimento de aplicações orientadas para a *web*, como segurança e desempenho (*performance*). Os temas são abordados nessa perspectiva: essencialmente *software* que recebe dados através de uma ligação de rede, executa algum processamento e envia uma resposta pela mesma ligação.

Actualmente, a maior parte do desenvolvimento faz-se sobre plataformas. Uma plataforma consiste numa base de código que implementa uma série de acções comuns às aplicações e um conjunto de componentes, módulos, etc, que podem ser usados para facilitar o desenvolvimento.

As plataformas, ao encarregarem-se de uma parte significativa do código que corre por cada pedido, "transferem" para si uma boa parte das responsabilidades (e problemas) tanto de *performance* como de segurança. Apesar disso, a abordagem aqui feita é genérica, inerente à arquitectura *web*, podendo, no entanto, haver referências a determinadas plataformas.

Ao longo do texto são exibidos pedaços de pseudo-código. Trata-se de uma forma clara[1] de expor uma ideia, de forma precisa, mas sem o fardo típico de uma implementação real. Na realidade, o pseudo-código usado é muito semelhante a uma linguagem de programação real: *python*[2]. As regras são muito simples: os blocos de instruções agrupam-se pelo nível de espaçamento entre a margem esquerda e o texto (*indent*[3]), deste modo não há chavetas nem `begin-ends`; uma variável precisa ser iniciada (atribuir-se-lhe um valor) antes de poder ser usada; e o seu tipo fica implícito, ou seja, é inferido pelo valor que lhe foi atribuído. As instruções de controlo (`if`, `while`, ...) terminam com o símbolo "`:`".

Segurança

Porque é que a segurança importa?

A humanidade tem algo de mau e perverso na sua essência. As pessoas dão-se a esforços incríveis para terem o prazer de estragar algo, só para provar que o conseguem fazer; divertem-se com desgraças alheias e tentam sistematicamente tirar vantagem de forma ilícita, mesmo que isso lhes dê mais trabalho do que teriam por vias "normais". É este espírito demoníaco que aumenta a importância da segurança.

Corolário às leis de Murphy: se um sistema tiver uma falha, essa falha vai ser explorada.

No caso de aplicações *web*, uma falha pode ser explorada em muitas variedades:

- *defacements*: mudam o visual da página. O bom dos *defacements* é que dá sempre para aprender asneiras novas.
- Ganhar acesso às máquinas.
- Obter informação que permita ganhos directos, como números de cartões de crédito.
- Tornar os serviços indisponíveis.

---

[1] supostamente :-)

[2] http://www.python.org

[3] a palavra "*indent*" não tem tradução directa em português

•

[este espaço foi intencionalmente deixado em branco]

Com a massificação e generalização da tecnologia, cada vez mais as pessoas vão querer serviços dos quais possam depender e confiar, tal como já acontece com a água, electricidade e telecomunicações. Para que isso aconteça, é necessário que os serviços — *web* neste caso — sejam seguros, fiáveis e disponíveis. Tendo em conta as considerações sobre a natureza humana expostas atrás, a solução passa forçosamente por investir em segurança.

Três máximas a não esquecer

- *Never, ever, trust user input*.
  Nunca se deve confiar em algo que venha de fontes exteriores. Esses dados não devem ser usados directamente em aplicações, pois podem conter código satânico que estrague muitas coisas. Se uma dada sequência de caracteres for enviada numa pesquisa a uma base de dados, essa sequência pode conter caracteres que terminem a pesquisa e apaguem informação, por exemplo.

```
"select * from tabela where name = '" + user_string + "'"
user_string="xpto"   OK
user_string="foo' from dual; delete tabela; -- "
                     OOPS... ver 3° ponto
```

- *Assumption is the mother of all... failures*![4]
  Outro erro é assumir. Isto é, partir do princípio que determinadas coisas são aquilo que podem não ser. Uma aplicação *web* recebe informação através de uma ligação de rede (*network socket*), antes de começar a realizar as suas tarefas propriamente ditas. Se uma ligação cliente nunca terminar de enviar dados e a aplicação assume que esses dados têm pequenas dimensões, coisas más vão acontecer...

```
cliente:
    s = socket_connect(victim_host, port)
    s.send("POST / HTTP/1.0\r\n")
    s.send("\r\n")
    while 1:
        s.send("a=1&") # lista de parâmetros sem fim

servidor:
    s = socket_accept(port)
    while 1:
        t = s.recv()
        if t == '':  # OOPS, nunca vai acontecer... ver 3° ponto
            break
        parse(t)
```

- ... Já estava assim quando cá cheguei!

Validar o *input* do utilizador

A falta de validações dos dados fornecidos pelos utilizadores, constitui uma das grandes brechas de segurança das aplicações *web*. O princípio é sempre o mesmo: qualquer tipo de dados ou conteúdos que sejam provenientes de fonte insegura, devem ser filtrados antes de serem usados. Isso significa passá-los por filtros de aceitação ou por filtros de transformação. A variedade de casos a validar é imensa, apresentando-se apenas um resumo minimalista:

---

[4] Versão original censurada: *Assumption is the mother of all fuckups*.

- definir padrões a aceitar, e rejeitar tudo o resto (em vez de definir padrões a rejeitar, e aceitar tudo o resto)
- *strings*: usar expressões regulares
- números: definir valores mínimos e máximos
- *e-mail*: validar um sub-conjunto de todas as formas possíveis e apenas deixar passar essas
- nomes de ficheiros: validar existência de caracteres perigosos ou inválidos, ter em atenção problemas de *encodings* (*utf-8* num *latin1 filesystem*, por exemplo)
- *cookies*: validar o valor do *domain*

Limitar utilização de recursos

Basicamente todo o tipo de recurso que a aplicação usa deveria ser limitado — *upper bound*. Isso nem sempre é possível. Outras vezes esse trabalho é da responsabilidade das plataformas em que a aplicação corre (um *application server* como o *weblogic*, por exemplo).

A imposição destes limites justifica-se porque os recursos são limitados, e um consumo abusivo de um deles pode provocar uma falha da prestação de serviço. A isso chama-se um ataque *DoS — denial of service*. Os *DoS* também se podem verificar na vida real e são comuns em situações de pânico generalizado.

- crise da Argentina: pessoas acorreram em massa aos bancos para levantar dinheiro — *DoS* financeiro.
- catástrofes naturais, como terramotos: pessoas usam telemóveis para contactarem familiares e amigos — *DoS* das telecomunicações.

Exemplos de recursos que devem ser limitados, tanto por pedido (i.e. limites que não podem ser ultrapassados por pedidos individuais), como a nível absoluto:

- CPU: tempo de utilização efectiva do processador.
- disco: dimensão de ficheiros criados.
- memória: consumo máximo de memória física (RAM)
- tempo: duração real que a aplicação demora a correr (de notar que este tempo é muito superior ao tempo de CPU)

Timeout everything!

Definir *timeouts* sempre que puderem ser definidos, especialmente em operações de *networking*. Deste modo evita-se ficar bloqueado com "bogus" clientes que não terminam as ligações, por exemplo.

Que se note bem que os *timeouts* devem ser definidos sempre que se puder, não apenas no estabelecimento de uma ligação de rede. De facto, tanto as leituras como as escritas num *socket* podem bloquear. Estas últimas são usualmente descuradas, não só na questão dos *timeouts*, como na confirmação do resultado da sua execução (se realmente escreveu todos os dados submetidos).

Não assumir valores pequenos para nada

Especialmente ao interpretar dados fornecidos por clientes, como *upload* de ficheiros, devem ser impostos limites superiores para não provocar esgotamento de recursos, como encher a totalidade do espaço em disco. Neste caso trata-se de um limite por pedido. Há que ter em atenção os limites por sistema: se muitos milhares de clientes fizerem *upload* de ficheiros, ainda que pequenos, provocam também um esgotamento de "espaço em disco", pois apesar de continuar haver espaço propriamente dito, não existem entradas disponíveis para a criação de novos ficheiros, o que acaba por resultar num efeito semelhante (a diferença é que neste caso os ficheiros que já existem podem crescer).

Smart logging

Fazer *logging* "inteligente". Normalmente os registos são usados pelos programadores para efeitos **apenas** de *debugging*. Deve-se ter em atenção outro tipo de dados a registar, como início e fim das ligações, as suas durações, operações realizadas, etc... Estes dados até podem ser úteis para outras áreas, como *marketing* (*data mining*).

Normalmente este tipo de registos é da responsabilidade da plataforma.

Não revelar demasiada informação

Deve ser feito um esforço para se limitar ao máximo a informação a que o utilizador tem acesso. Existem dois casos típicos: comentários que se deixam embebidos em código a que o utilizador acaba por ter acesso (HTML, *javascript*, JSPs) e diferentes mensagens que caracterizem os diversos casos de erro.

No primeiro caso, trata-se de revelar informação que pode fornecer pistas a possíveis atacantes: normalmente os comentários são escritos pelos programadores para documentar ou clarificar uma situação, o que como é evidente pode ser revelador de informações mais sensíveis. Os comentários, especialmente nas páginas que têm código e HTML misturados (PHP, ASP, JSP) devem ser completamente removidos nos sistemas de produção! O caso mais grave e frequente, é quando se quer eliminar uma série de código, e, como é prática usual em desenvolvimento, se comenta todo esse código — que passa a estar disponível na íntegra a quem aceder à página — em vez de se o apagar.

O segundo caso é exemplificado nas páginas de *login*. Um utilizador coloca o seu identificador e *password*. Existem uma série de situações de erro diferentes que podem ocorrer: o utilizador não existe, existe mas a *password* está errada, houve erros de sistema, ... Se a mensagem de erro que se envia ao utilizador descreve o que se passou, esta página pode ser usada como um meio para ganhar informação: tentar descobrir utilizadores registados, *passwords*, se a máquina está sujeita a grande carga de utilização, etc... A simples mensagem "Erro, reintroduza os seus dados correctamente" deve ser suficiente.

Performance

Sensibilização

Algo profundamente errado que se ouve e lê frequentemente, assemelha-se a: "como as máquinas cada vez são mais rápidas, têm mais memória, mais espaço em disco, então... podemos fazer *software* mais lento, usar mais memória, ocupar mais espaço...".

Além de isto ser perigoso (se todos pensassem desta maneira — a evolução dos outros resolve os nossos problemas — ninguém fazia nada, ou fazia-as sempre da mesma maneira, nunca evoluindo), constitui uma grande falácia.

A "lei de Moore"[5] estabelece que a *performance* dos CPUs duplica a cada 20 meses. Só que o volume de dados/informação a processar (i.e. informática!) cresce muito mais rápido. Em 1998 estimava-se que os conteúdos na *internet* cresciam duas vezes a cada 50 dias. A este crescimento, junta-se o custo de processamento desses dados, já que muitos algoritmos têm custos acima do linear (exemplo de ordenação: $O(n \log_2 n)$). Isso significa que para continuar a manter o passo, temos que empregar melhores algoritmos e usar melhores técnicas/tecnologias. Não podemos descurar nem a *performance*, nem a "bondade" dos sistemas só com a desculpa de que as máquinas de amanhã nos vão resolver os problemas de hoje.

Optimizar o quê?

Uma aplicação *web* consiste basicamente em três grandes blocos: aguardar os pedidos de clientes, executar a funcionalidade pretendida e enviar os resultados de volta aos clientes.

Não vale a pena fazer pequenas optimizações de código (embora nunca façam mal), porque os ganhos são completamente absorvidos pela estrutura da aplicação, nomeadamente nos tempos de latência de rede (receber o pedido e enviar as respostas).

As optimizações devem ser focadas a:

---

[5] Também aqui existem uma série de más concepções. Moore inicialmente previu que o número de transístores por *chip* duplicaria em cada ano. Mais tarde, Moore corrigiu a sua própria previsão para metade — a densidade duplicaria a cada dois anos. Na mesma altura em que fez essa correcção, outra pessoa da *Intel* disse que isso teria um efeito de duplicação do desempenho dos CPUs a cada 18 meses. É a esta afirmação a que se chama "Moore's Law", embora seja o próprio a admitir que não é dele. Na verdade, a *performance* dos CPUs tem estado a duplicar a cada 20 meses.

- nível da arquitectura: como se recebe os pedidos, como se os processa e como são enviados ao utilizador
- volume de dados: quantidade de *bytes* que se transferem pela rede; quanto menos, melhor
- diminuição de contenções ao nível do servidor: quanto menos uma aplicação tiver que esperar (literalmente), mais rápida será. Essas esperas normalmente resultam da concorrência a recursos partilhados, ou de dependências de sistemas externos.

A optimização de desempenho de um produto, como é de esperar, passa por diversos aspectos, desde a escolha do próprio *hardware*, configurações e qualidade, tanto do sistema operativo como das plataformas que se usam, arquitectura global do sistema, e, finalmente, da aplicação propriamente dita. As técnicas que se apresentam a seguir não são todas da responsabilidade do programador, algumas delas devem ser levadas a cabo pelos administradores de sistemas, por exemplo. No entanto, a sua compreensão é essencial para se escrever *software* eficiente.

Compressão

A melhor técnica de "optimizar" uma aplicação *web* é comprimir os resultados através do método de compressão *gzip*.

Muitos clientes (*browsers*) suportam conteúdo comprimido. Isso é determinado através dos *headers* que enviam no pedido original (`Accept-Encoding: gzip`). Uma aplicação deve detectar se o *browser* aceita conteúdo comprimido e se o resultado for suficientemente grande que justifique a sua compressão, fazê-lo, notificando isso através dos *headers* de resposta.

```
# escreve results em output
servidor(output, results):
    support_gzip = false
    # detecta: Accept-Encoding: gzip
    enc = get_header("Accept-Encoding")
    # versão simplificada-- há que tomar em conta o factor "q"
    if enc.contains("gzip"):
        support_gzip = true
    if len(results) > 4000 and support_gzip:
        # produz: Content-Encoding: gzip
        set_header("Content-Encoding", "gzip")
        gz_res = gzip(results)
        output.print(gz_res)
    else:
        output.print(results)
```

Esta é a mãe de todas as optimizações *web related*. Este trabalho pode ser feito ao nível do código da aplicação, ao nível da plataforma (preferido) ou alternativamente usando os chamados "filtering proxys" — os pedidos são enviados ao *proxy* que os despacha para a aplicação *web*. Esta envia a resposta normal ao *proxy*, que lhe aplica uma série de transformações. É uma forma "limpa" de optimizar aplicações existentes sem mudar uma linha de código.

HTML *cleanup*

A maior parte do *html* que vemos hoje em dia nas diversas páginas, é produzido por ferramentas visuais. Essas ferramentas normalmente têm duas coisas em comum:

(1) tornam o trabalho de criação gráfica mais fácil

(2) produzem uma grande quantidade de lixo desnecessário

(por razões semi-óbvias, não me pronuncio sobre o efeito dos *web-designers* neste processo :-)

Um truque para lidar com este problema é pós-processar o *html* produzido e "limpá-lo". As limpezas incluem coisas como:

- apagar a maior parte dos espaçamentos
- apagar a maior parte dos comentários (uma medida simultaneamente de segurança, aliás)
- apagar *tags* vazias (`<b></b>`, por exemplo)
- converter todos os nomes das *tags* e atributos para minúsculas, para aumentar a redundância e melhorar uma eventual compressão posterior.

Buffering, cache

Guardar em memória todos os **pequenos** conteúdos que puderem ser guardados. Guardar em disco os de dimensão média. Fazer isto sempre que a fonte de dados original seja mais lenta que estes meios. Por exemplo, os resultados de um acesso a uma base de dados podem ser gravados em memória. Para disco também, mas convém fazer testes — muitas vezes o acesso ao dados em disco (que inclui o acesso aos dados propriamente ditos, mais a conversão para as variáveis que o programa vai usar) pode ser mais lento que um acesso à base de dados.

Evitar conversões de *charsets*

TCP funciona *byte* a *byte*. Portanto, lêem-se *bytes* e escrevem-se *bytes*. Muitas linguagens, como `java`, usam caracteres — `char` — que são dois *bytes*. Quando se escreve para um destino que queira *bytes*, é feita uma conversão desses caracteres para *bytes*. Quanto se lê de uma fonte que forneça *bytes*, é feita uma conversão para caracteres. No fim existem duas conversões recíprocas, logo redundantes. O ideal é evitar conversões: **sempre que possível**, ler *bytes* e escrever *bytes* em toda a parte. Notar bem o "sempre que possível", e nessa frase, notar ainda melhor o "possível".

Por exemplo, em java servlets/jsp, deve obter-se o objecto HttpServletResponse.getOutputStream() para se fazerem as escritas, sempre que possível, em vez do HttpServletResponse.getWriter().

***Pooling* de recursos**

Existem recursos que têm elevados tempo de latência, ou seja, custam a obter. Um exemplo típico são as ligações a bases de dados. Manter um conjunto de ligações abertas, ainda que não sejam usadas num dado período, para estarem sempre disponíveis, melhora o desempenho. Além das ligações às bases de dados, as *threads* constituem outro recurso onde se pode aplicar a mesma técnica.

Keep those databases working!

As bases de dados funcionam melhor quando têm um fluxo de saída dos resultados constante. Depois de um `select` por exemplo, os resultados devem ser obtidos pela aplicação de forma contínua.

Normalmente, o ciclo de acesso a uma base de dados é o seguinte:

```
db = do_query("select * from xpto")
while db.have_data():
    data = db.get_data()
    process(data)
db.close()
```

Se `process()` for suficientemente rápido, o ciclo de cima não provoca atrasos. Caso contrário o desempenho sofre atrasos que podem ser significativos. Existem duas formas para corrigir o problema. Se os dados forem pequenos (mas atenção com o que se disse acima sobre assumir coisas!), podem ser lidos na íntegra para uma estrutura de dados e usados depois:

```
db = do_query("select * from xpto")
data = [] # array de dados
while db.have_data():
    data.push( db.get_data() )
db.close()
for d in data:
    process(d)
```

A outra alternativa consiste em usar uma *thread* adicional para ler os dados, e manter a principal a executar o seu processamento. Essas duas *threads* comunicam entre si através de uma *queue* síncrona, por exemplo.

```
queue = new queue()
t1 = new thread()
t1.run(queue)
while 1:
    data = queue.dequeue()
    if data == DATA_ENDED:   # no more data to receive
        break
    if data == NO_DATA:  # queue is empty
        sleep_a_little()
        continue
```

```
        process(data)

    t1:
        run(queue):
            db = do_query("select * from xpto")
            while db.have_data():
                queue.enqueue( db.get_data() )
            db.close()
            queue.enqueue( DATA_ENDED )
```

Esta ideia, usada numa aplicação real, revelou um aumento de desempenho cerca de quatro vezes superior, embora ressalve-se que a aplicação em causa constitui um caso extremo de uso desta medida, dado que lê uma grande quantidade de dados em sequência e o processamento de cada registo é particularmente lento.

Preload images

Se um *site* usa um conjunto relativamente limitado de imagens, e as usa com muito frequência (ícones, por exemplo), uma optimização para o cliente poderá passar pelo seu *browser* fazer *cache* dessas imagens logo na primeira página.

Eis um exemplo em que três imagens são pré-carregadas:

```
<html>
<head>
    <title>Preloading images with JavaScript</title>
<script>
<!--
    img1 = new Image();
    img1.src = "img/back.gif";
    img2 = new Image();
    img2.src = "img/next.gif";
    img3 = new Image();
    img3.src = "img/head.gif";
//-->
</script>
</head>
```

Keep alive connections

O protocolo *HTTP* é *stateless*. É feito um pedido, enviada uma resposta e fecha-se a ligação. Na versão 1.1 deste protocolo, surgiu o conceito de manter a ligação aberta, de forma a poder reusá-la em futuros pedidos. Um *browser* e um servidor chegam a esse acordo através do *header* "`Connection: Keep-Alive`".

A vantagem é não ter que restabelecer sistematicamente ligações de rede entre o cliente e o servidor. Para isto ser possível, é necessário que a dimensão do conteúdo enviado na resposta do servidor seja conhecida. Apenas desse modo o cliente sabe quantos dados pode ler da ligação antes de lá colocar os dados do novo pedido.

A gestão dessa informação normalmente é feita pela plataforma, mas para que funcione, é preciso que o programador não execute *flush*s no seu código. Isso provoca que os conteúdos actuais sejam colocados na ligação, incluindo o *header* da resposta. Só que é no header que vai a dimensão da resposta e esta ainda não se conhece na totalidade (o programador fez um *flush* do que já tinha, mas ainda pode continuar a enviar dados). Esta situação é prevista pelo modo *chunked* (`Transfer-Encoding: chunked`), mas não sei até que ponto esse técnica é suportada pelos *browsers* ou plataformas.

A ideia final é: evitar *flush*s.

Parâmetros de *networking*

Uma ligação *tcp/ip* passa por uma série de estados desde o seu estabelecimento ao seu fim. Pode haver *software* que não cumpra o protocolo com correcção.

Uma característica frequente é o estado `FIN_WAIT_2`. `FIN_WAIT_2` é o estado em que um dos lados notifica que vai fechar a ligação e fica à espera de resposta.

Se os clientes nunca derem essa resposta, esse porto vai ficar sempre ocupado o que provoca um DoS ao nível de portos disponíveis.

O comando `netstat`, em *Unix*, mostra todas as ligações existentes, bem como os seus estados.

Para resolver o problema do `FIN_WAIT_2`, a maior parte dos sistemas operativos define um *timeout* para libertar o porto, mesmo que não haja confirmação do *peer*.

Em *linux*, esse valor é configurável através do ficheiro: `/proc/sys/net/ipv4/tcp_fin_timeout`.

Esta responsabilidade, obviamente, é do sistema operativo e não da aplicação.

Parâmetros *I/O*

Ainda ao nível do sistema operativo, pode-se tentar melhorar o desempenho das nossas aplicações. Um caso de estudo é o *filesystem* do sistema operativo Solaris. Na maior parte dos *filesystems*, de sistemas operativos decentes pelo menos, é feita uma actualização da data do último acesso aos conteúdos de cada ficheiro. Em *Unix* isso chama-se `atime`. Numa aplicação *web* existem milhares de ficheiros, desde páginas *html* a imagens, a serem acedidos quase instantaneamente. Uma actualização — escrita — no disco por cada acesso, tem um impacto claramente negativo na *performance*. A partir do Solaris 7, existe uma opção do `mount` para que essa actualização não seja feita: `noatime`.

```
# exemplo do ficheiro /etc/vfstab
/dev/dsk/c0t3d0s0 /dev/rdsk/c0t3d0s0 / ufs 1 no logging
/dev/dsk/c0t3d0s6 /dev/rdsk/c0t3d0s6 /www ufs 1 no logging,noatime
```

Neste caso, qualquer acesso a ficheiros da partição `/www` é apenas *read-only*, não provocando actualizações no disco.

Arquitectura...

Por vezes realizar o mesmo tipo de tarefas com pequenas variantes, pode produzir resultados bastante diferentes em termos de desempenho. Usando o mesmo conjunto de recursos, como servidores, podemos obter níveis de resposta muito superiores mediante determinado tipo de configurações. A seguir apresentam-se alguns casos reconhecidos como sendo mais eficientes em arquitecturas *web*.

Separar conteúdos dinâmicos e estáticos
: Os servidores de conteúdos dinâmicos (*application servers* em `java`, como o *weblogic*, por exemplo) tipicamente são mais lentos nas conectividades do que servidores *web* puros, como o *apache*. A ideia é usar os servidores dedicados para conteúdos estáticos (HTML e imagens) e os servidores de aplicações para os conteúdos dinâmicos (JSPs e *servlets*). Uma configuração com algum sucesso é *apache+weblogic*, em que o *apache* faz *proxy* para o *weblogic* todos os pedidos às extensões `.jsp` e, digamos, `/servlets/*`, servindo directamente todos os restantes ficheiros.

Imagens servidas à parte
: Este talvez seja um dos desenhos mais interessantes e menos usados em sistemas *web*. As imagens têm um grande impacto a nível de *network I/O*, porque uma só página referencia bastantes imagens, muitas das quais diferentes. Cada imagem referenciada provoca uma ligação HTTP extra ao servidor, para obter esse conteúdo binário. Um serviço de imagens dedicado tem como vantagem a eliminação de contenção *I/O* do servidor principal: enquanto fornece conteúdos HTML ao cliente, este vai abrindo ligações ao servidor de imagens, libertando o servidor principal para servir outros clientes. O fornecimento de imagens não é tão prioritário como servir um pedido, porque o *rendering* do *browser* vai-se processando enquanto a página está a carregar. Esta separação de serviços só faz sentido se correrem em máquinas distintas, pois o problema de contenção é ao nível de *sockets*, o que afecta todas as aplicações a correr na mesma máquina.

  Outra vantagem de manter as imagens em servidores dedicados, advém de eventuais ganhos em facilidade de gestão: como tudo está centralizado torna-se mais fácil globalizar operações, como mudanças de imagens, ou aplicação de técnicas de redução das suas dimensões.

Pipeline
: Em aplicações *web* é comum existirem *thread pools*, ou seja, um conjunto de *threads* que são

alocadas para processar cada novo pedido. Essas *threads* normalmente recebem o pedido já previamente tratado, nomeadamente com os parâmetros e *headers* disponíveis, e têm como responsabilidade fazer o processamento propriamente dito, bem como o envio da resposta.

Uma solução que optimiza a eficiência geral do sistema, consiste em destacar o trabalho de enviar a resposta a outra *thread pool*.

```
# modelo tradicional:
#  - recebe dados do pedido em request
#  - executa processamento e envia resposta em output
server(request, output):
    results = process(request)
    output.print(results)

# modelo pipelined:
server2(request, output):
    results = process(request)
    t = io_thread.get()
    t.add(output, results)

get_io_thread:
    t = nil
    new():
        # singleton
        if t == nil:
            queue = new synchronous_queue()
            t = new get_io_thread() on a separate thread
        return t
    get():
        return t
    add(output, results):
        queue.put(output, results)
    run():  # always running on a different thread
        while 1:
            output, results = queue.get()
            if output == nil:        # queue is empty
                sleep_a_little()
                continue
            output.print(results)
```

Este exemplo não gere uma *pool* de *threads I/O*, apenas gere uma *thread*. O interesse desta técnica é em casos onde a resposta a enviar ao utilizador tem algum impacto: páginas grandes ou ligações lentas. Nestes casos o rácio entre *worker threads* e *I/O threads* é de sensivelmente 20:1.

De notar ainda que muitas plataformas impossibilitam a implementação deste tipo de técnicas, como é o caso dos *java application servers*.

Problemas recorrentes

Evitar *timeouts* dos clientes

Uma situação típica não desejada, é haver *timeouts* por parte dos clientes (*browsers*). Isso acontece porque o processamento do lado do servidor demorou bastante tempo, não houve actividade na ligação de rede e o *browser* "desistiu".

Um truque para resolver esse problema, consiste em enviar páginas de espera sucessivas ao utilizador, enquanto o processamento final não termina. Exemplo em pseudo-código:

```
request():
    q = get_param("q")
    if q is null: # original request
        t = start new thread
        t.work()
        # o utilizador esta disposto a esperar ate 10 segundos
        sleep(10)
```

```
        if t.finished():
            send(t.results())
        else:
            q = gen_id()
            wait_page(q)
    else: # callback
        if t.finished():
            send(t.results())
        else:
            wait_page(q)

wait_page(q):
    send '''
Content-type: text/html

<html>
<head>
<META http-equiv="REFRESH" content="2; URL=myurl?q=%s>
</head>
<body> Please, wait a little... </body>
</html>''' % (q)

gen_id():
    str = "abcdefghijk"
    n = len(str)
    s = ''
    for i = 1 to 32:
        s = s + str[rand() % n]
    return s
```

Evitar *timeouts* II

Outro método para fazer o mesmo, consiste em usar técnicas de *dynamic html*. Neste exemplo, criam-se duas divisões, uma com o conteúdo de espera, e outra que referencia a fonte que gera a resposta interessante. A primeira é marcada como visível e a segunda como invisível. Quando o documento está completamente carregado, a visibilidade de ambas as divisões é trocada.

```
<html>
<head>
<style type="text/css">
    #waitpage { position: absolute; }
    #mainpage { position: absolute; visibility: hidden; }
</style>

<script language="JavaScript">
function init () {
    if (document.layers) {
        document.waitpage.visibility = 'hide';
        document.mainpage.visibility = 'show';
    } else {
        if (document.all) {
            document.all.waitpage.style.visibility = 'hidden';
            document.all.mainpage.style.visibility = 'visible';
        }
    }
}
</script>
</head>

<body onLoad="init();">

<DIV ID="waitpage">
    <center><i> Wait please ... </i></center>
</DIV>
```

```
<DIV ID="mainpage">
    <script language="JavaScript">
        location.href='http://host/path/servlet/myServlet';
    </script>
</DIV>

</body>
</html>
```

***Download* de ficheiros sem os abrir**

Problema: o ficheiro é enviado e o *browser* do cliente lança automaticamente uma aplicação para o abrir. A solução passa por definir os *headers* correctos na resposta a enviar. Neste caso, a resposta HTTP a enviar ao cliente deve ser a seguinte:

```
Content-type: APPLICATION/OCTET-STREAM
Content-Disposition: attachment; filename="xpto.zip"

[conteúdo binário de xpto.zip]
```

Em código, resume-se a usar os métodos que manipulam o valor dos *headers* de resposta. Exemplo em `java`:

```
// an HttpServletResponse response object is available here...
String name = "xpto.zip";
String path = "/myzips/";
response.setContentType("APPLICATION/OCTET");
response.setHeader("Content-Disposition",
  "attachment; filename=\"" + name + "\"");
FileInputStream in = new FileInputStream(path+name);
OutputStream out = response.getOutputStream();
byte [] buf = new buf[16384]; int n;
while ((n = in.read(buf, 0, buf.length)) > 0)
    out.write(buf, 0, n);
out.close();
in.close();
```