Instituto Superior Técnico – Universidade de Lisboa
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Ano Lectivo 2013 / 2014
Lisboa, 20 de Maio de 2014
Programação de Sistemas

PROGRAMAÇÃO DE SISTEMAS

Relatório do Projecto

Servidor WEB

Grupo nº 2

Maria Margarida Dias dos Reis, nº 73099

Ricardo Filipe Amendoeira, nº 73373

Recepção de um pedido

Quando chega um pedido de um determinado cliente ao servidor, a *thread* principal chama uma função responsável por fazer o *accept*. Esta função devolve para o processo principal o *file descriptor* da *socket* desse cliente, que é depois escrito numa FIFO, em conjunto com o endereço IP do cliente, a data e a hora em que o pedido foi aceite e o início da contagem em milisegundos da duração da resposta ao pedido do cliente.

A *pool* de *threads* que é criada ao início tem a responsabilidade de ler da FIFO a estrutura enunciada anteriormente e chamar uma função que analisa o pedido feito. Quando um determinado pedido já acabou de ser atendido o *file descriptor* da *socket* desse cliente fica livre e será reutilizado futuramente.

Threads e processos

São usados processos extra para executar outros programas, como o ls para listar as directorias ou os CGI's. Há uma threads que armazena as estatísticas numa lista, uma threads de controlo que cria mais threads de atendimento quando necessário e essas mesmas threads de atendimento a clientes há portanto 3 tipos diferentes de threads.

Gestão de *threads* e processos

No início do programa é criada uma *pool* base de *threads* com 150 *threads*, sendo que este valor não se mantém constante ao longo do programa, sendo de facto crescente monotónico. É definido que o número de *threads* livres nunca pode cair abaixo de 10 e assim, quando há menos de 10 *threads* livres de atendimento aos clientes existe uma função de controlo e gestão de *threads* que cria o número necessário das mesmas para que voltem a ser 10. A ideia é, depois de criada a *pool* base criarem-se mais blocos de *threads*.

As *threads* são terminadas são terminadas quando se interrompe o servidor, com a excepção da thread de controlo, que termina se já não for possível abrir novas threads.

Existem *threads* de atendimento aos clientes, uma threads que armazena as estatísticas e uma thread de controlo que cria novas threads de atendimento quando necessário.

Controlo de tempo dos CGIs

Para a execução de CGIs é feito inicialmente um *fork*. O processo filho é responsável por executar o CGI e escrever o *output* deste na *socket* do cliente que efectuou o pedido. O processo pai começa por registar o tempo actual e entra depois num ciclo em que espera que o processo filho mude de estado dentro de 5 segundos. Se depois do ciclo se verifica que o processo filho não

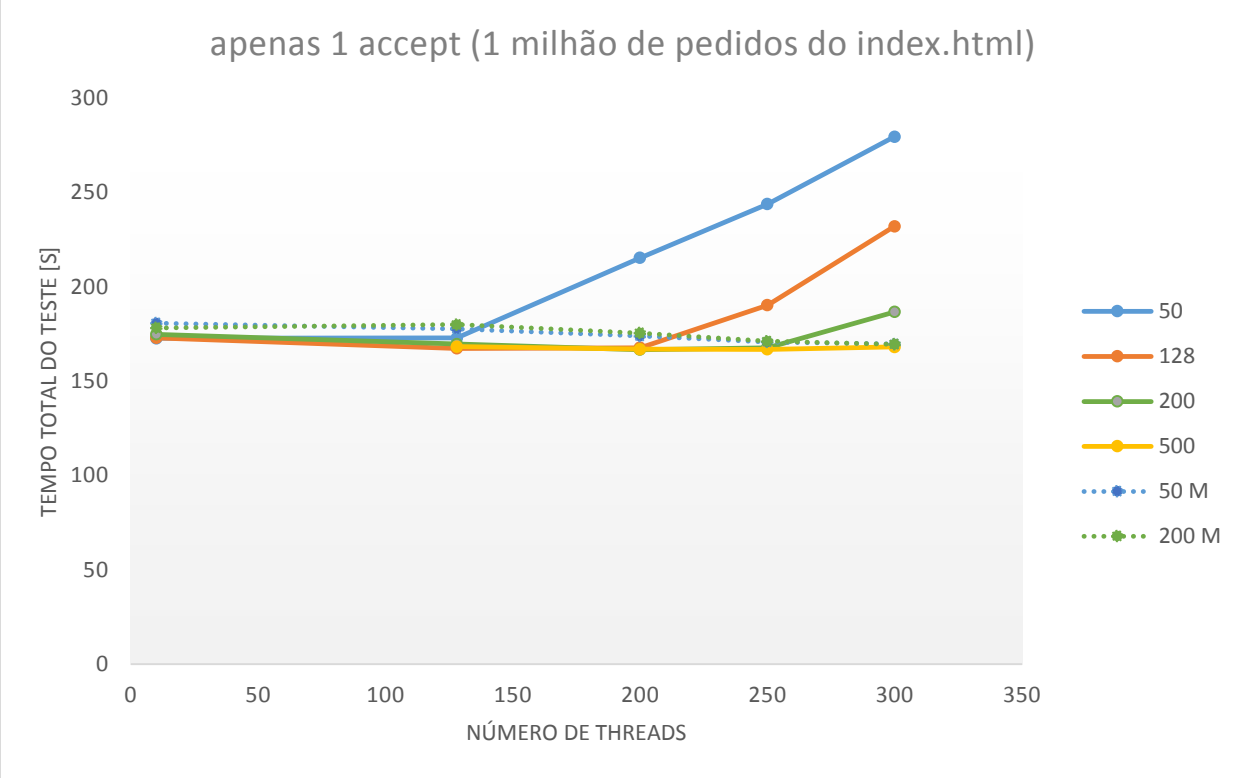
mudou de estado então é porque nunca se completou a execução do CGI dentro dos 5 segundos e mata-se o processo filho, ou seja, interrompe-se a execução do CGI.

Armazenamento dos pedidos

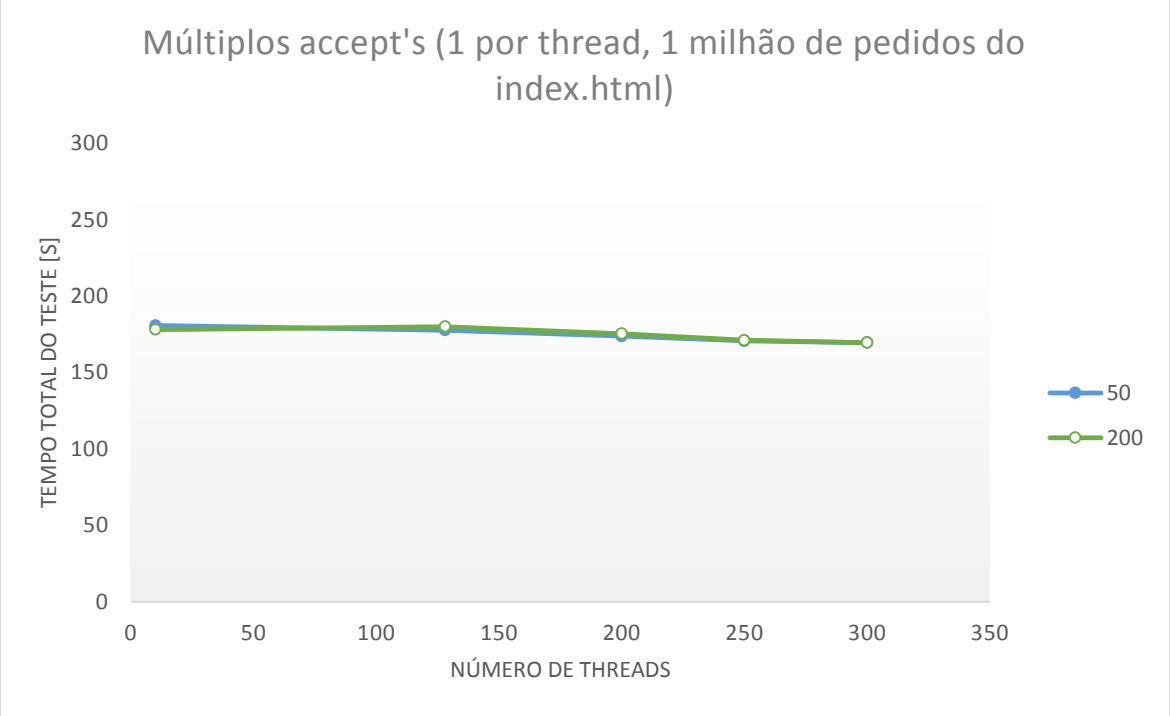
Testes de carga

Foram feitos diversos testes com o ab, não só para garantir a estabilidade do servidor mas também para descobrir a melhor forma de controlar a criação de threads e a forma como é feito o accept. Os gráficos seguintes são resultado de testes executados com o ab a correr na mesma máquina que o servidor com o objectivo de estudar diferentes implementações. Também se fizeram testes com o ab numa máquina diferente da do servidor mas esses resultados não foram apontados por serem apenas para garantir estabilidade.

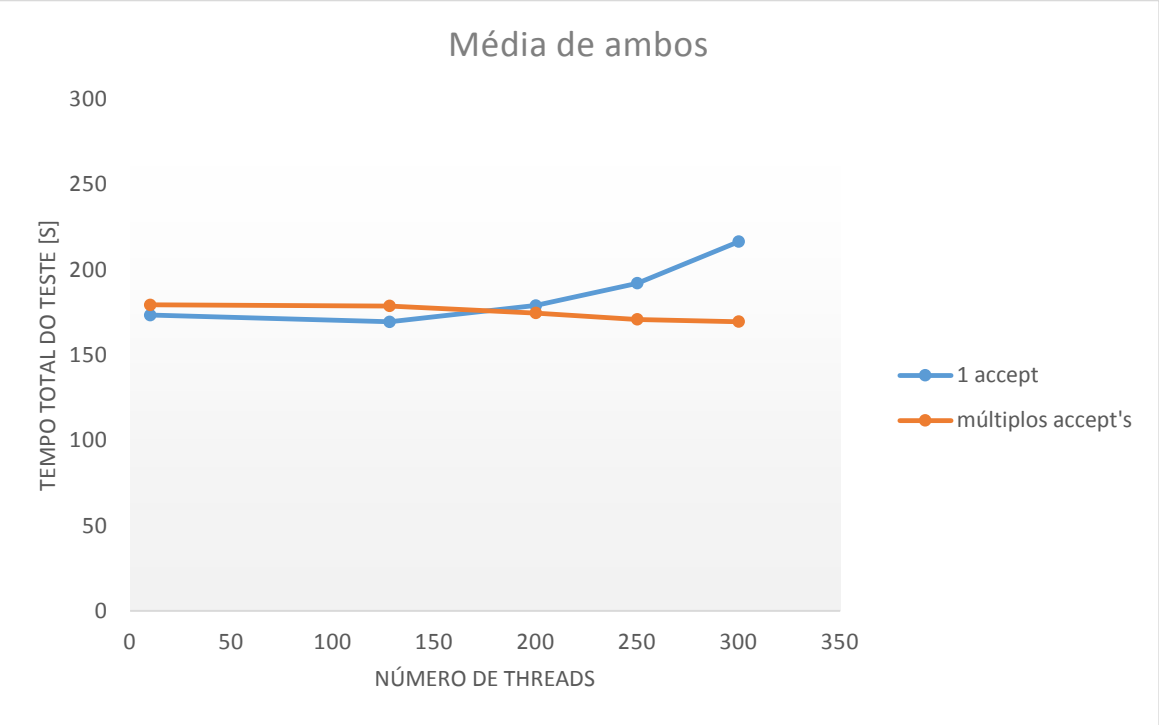
O seguinte gráfico compara a performance do servidor com um número fixo de threads e a responder a diferentes quantidades de pedidos concorrentes. Permitiu concluir que ter threads em excesso deteriora significativamente a rapidez das respostas.



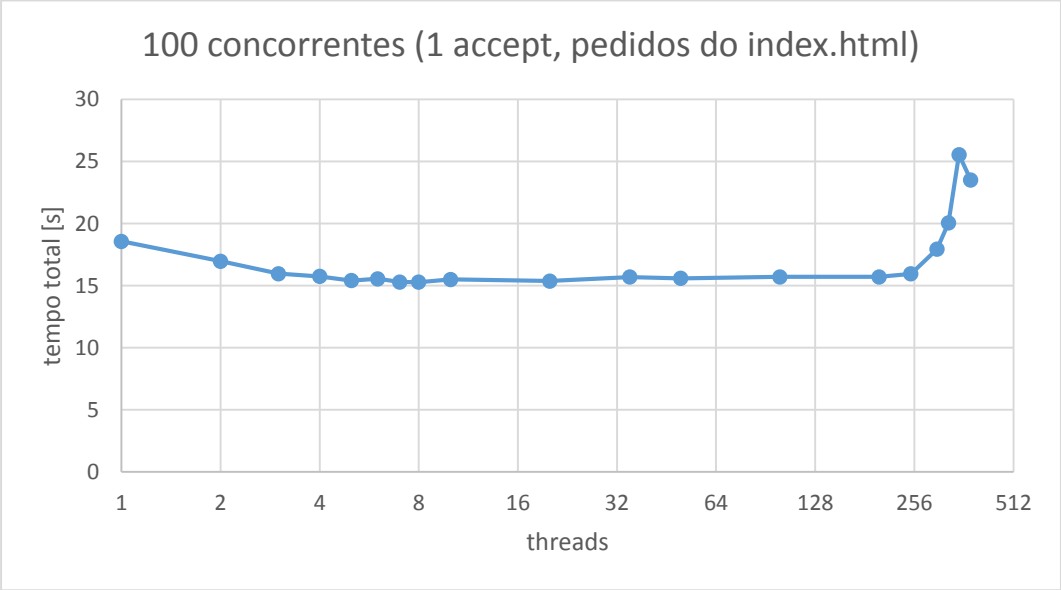
O gráfico seguinte mostra que com múltiplos accepts não há impacto negativo na performance do servidor se houverem threads em excesso, no entanto quando comparado ao método do gráfico anterior este método é ligeiramente mais lento (se o número de threads usado para o método anterior for adequado).



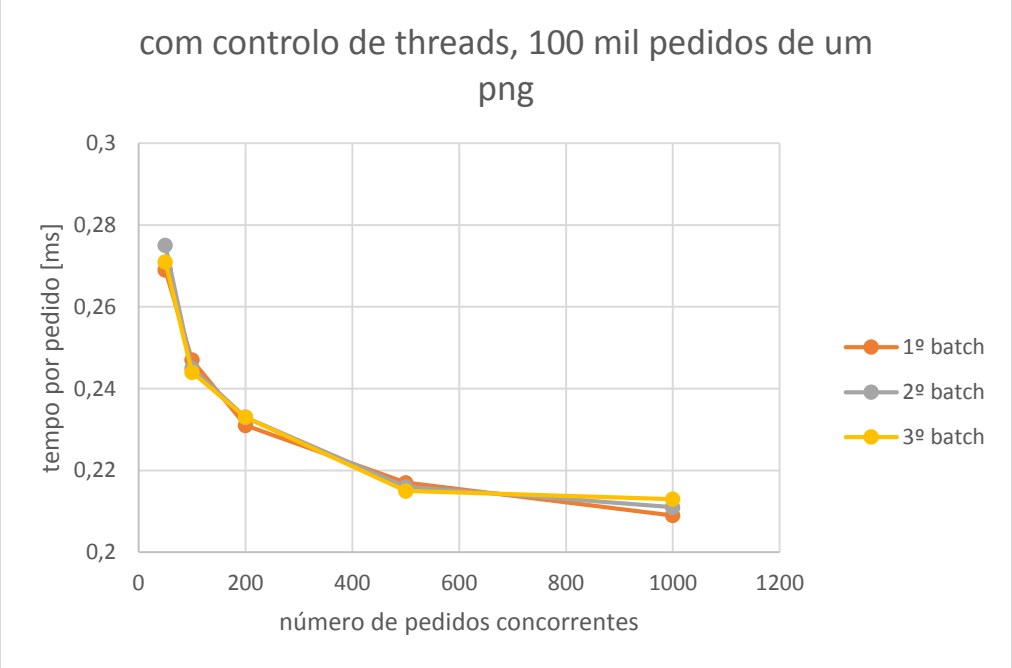
No gráfico seguinte é possível comparar mais facilmente ambos os métodos:



Nos testes do gráfico seguinte comparou-se o efeito do número de threads na rapidez das respostas. Conclui-se que após saturar os núcleos do processador (8 threads no caso do processador usado no teste) deixa de haver vantagem em termos de velocidade, podendo mesmo degradar (como visto no 1º gráfico) se o número de threads for excessivo. De notar que ter muito menos threads do que o número de pedidos concorrentes diminui a estabilidade do servidor, causando os erros 104, pelo que há vantagens em ter controlo de threads, para evitar ambos os extremos.



Estes testes permitiram chegar ao método usado na versão final do servidor, com os seguintes resultados:



Para concluir, a versão final do servidor foi submetida a diversos testes de carga, com vários pedidos diferentes de forma a testar todas as diferentes funcionalidades (enviar ficheiros html, listar uma directoria, pedir a lista de estatísticas, etc…).

Estes testes tiveram entre 100 mil e 1 milhão de pedidos totais cada, com diferentes níveis de concorrência (entre 50 e 1000) e o servidor manteve-se estável, embora com um número muito elevado de pedidos concorrentes (para cima de ~500 com o ab a correr na mesma máquina que o servidor) surjam ocasionais erros 104.