Universidade Federal de São João del-Rei

Servidores web iterativos, concorrentes, utilizando fork, threads e filas de tarefas: Explorando a linguagem C para programação com sockets.

Isabella Vieira Ferreira
Lucivânia Ester da Costa
Mônica Neli de Resende

3 de novembro de 2015

Sumário

Lista de Figuras

1 Introdução

A implementação de servidores web é utilizada ultimamente para diversas finalidades, destacando-se entre elas para hospedagem de sítios eletrônicos e para processamento de informações que necessitam de passar pela internet. A atual arquitetura de software passou por uma evolução no seu projeto: antigamente o usuário interagia com um programa monolítico que continha o código para gerenciar a aplicação, os dados, a interface do usuário e a comunicação.Agora, existe uma programação totalmente organizada, no que se refere às funcionalidades de um servidor, a qual possibilita uma divisão bem clara dessas partes no projeto de um software. Com isso, é possível realizar implementações mais eficazes de transferência de grandes quantidades de dados a partir de um servidor. As principais estratégias que serão abordadas ao longo deste trabalho são servidores iterativos, utilizando uma thread ou fork, servidores com filas de tarefas ou com concorrência.

Para a implementação de servidores é importante saber o conceito de sockets, que são arquivos que permitem a transferência dos dados de um cliente (aplicação), podendo ser o browser ou outra aplicação que gera requisições a um servidor.

Os servidores ditos iterativos são os modelos mais simples de implementação desse tipo e que servem apenas para uma noção de como funciona as requisições em um servidor. Cada requisição é feita separadamente, uma por vez. o socket é criado, espera a requisição e a executa. logo em seguida ele volta a esperar uma requisição novamente. Tudo sequencial.

Já os servidores com thread ou fork geram um processo paralelo, sendo o fork gerando um processo filho, dependente do processo principal (pai) e a com thread gera um processo independente que trata as requisições enquanto o processo principal aguarda outra chamada.

Os servidores que utilizam a ideia de produtor-consumidor necessitam de mais de uma thread, as quais são selecionadas pelo sistema operacional e então os semáforos funcionam para alternar entre a produção e consumo das requisições.

As aplicações que utilizam a chamada de sistema select já possuem uma característica de verificar requisições de acordo com um timeout especificado, abrindo diversos sockets para ”esperar”as requisições.

Tendo em vista esses conceitos, daremos continuidade com os detalhes ao longo deste trabalho, assim como uma análise mais aprofundada dos resultados das execuções destas estratégias. Determinaremos por fim qual estratégia se mostra mais eficaz em determinados cenários.

2 Descrição das rotinas utilizadas

2.1 Implementação com sockets

A programação de servidores utilizando a linguagem C necessita da implementação de sockets para realizar a conexão entre processos independente do protocolo utilizado (COSTA,2012). Ainda segundo COSTA (2012), os sockets podem ser caracterizados da seguinte forma:

- **Orientado à conexão** ou **não orientado à conexão**: Pode ser estabelecida uma conexão antes da comunicação ou então cada pacote pode descrever o destino.

- **Orientado ao pacote** ou **orientado ao fluxo**: Pode existir um limite de mensagens ou então pode existir um fluxo de mensagens, como no caso, um *stream*.

- **Confiável** ou **não confiável**: As mensagens podem ser perdidas, duplicadas, reordenadas ou então podem estar corrompidas.

Para realizar a conexão e enfim realizar requerimentos e respostas, é necessário fazer a implementação adequanda de acordo com o protocolo utilizado e o tipo da conexão. Para este trabalho, o protocolo utilizado é o TCP, orientada à conexão, orientada ao pacote (não estamos trabalhando com *streaming* de dados), e utilizando conexão confiável.

Nesse contexto, para implementar um servidor simples, é necessário seguir os passos para criar um socket, utilizá-lo e fechá-lo, como segue:

1. **Comando socket** : Define a criação do socket para sua utilização.
2. **Comando bind** : Define e atribui o endereço do socket no servidor.
3. **Comando listen**: Especifica o número máximo de pedidos de conexões que podem estar em espera para este processo.
4. **Comando accept** : Estabelece conexão com o cliente especificado.
5. **Comando send, recv** : Equivalente ao à escrita e leitura, mas baseado nos pacotes.
6. **Comando shutdown** : Estabelece o fim de escrita ou leitura.
7. **Comando close** : Libera as estruturas de dados do kernel.

2.2 Servidor iterativo

2.2.1 Fluxo de vida de socket dentro do servidor iterativo

O fluxo de vida de sockets no servidor iterativo segue o seguinte fluxograma:

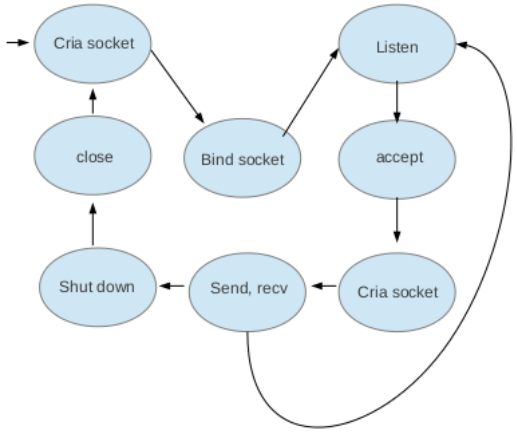


Figura 1: CICLO_SOCKET

A ideia de imlementação do servidor iterativo é basicamente criar, conectar e colocar para escutar as resquisições. Um socket por vez é aberto, não havendo, portanto, paralelismo implementado no primeiro momento.

2.2.2 Descrição geral do programa de servidor iterativo

Primeiramente, o socket que ficará escutando as requisições é **criado** utilizando a função socket, com os parâmetros AF_INET, SOCK_STREAM, e 0. AF_INET é o domínio que deve ser utilizado. AF_INET é o mais comum utilizado para implementação em socket. SOCK_STREAM é definido para estabelecer um circuito virtual para o fluxo de dados. E o último parâmetro é o protocolo, que por padrão recebe zero. A definição do socket, portanto, vem dos dois primeiros parâmetros. Caso o retorno da criação do socket seja -1, isso quer dizer que houve um erro no momento da criação, e então um retorno ao usuário deve ser realizado e então o programa deve ser finalizado.

Feito isso, é realizada a conexão (bind) do socket com o servidor. Para isso, é utilizada a estrutura de dados *sockaddr* que referencia a estrutura *servaddr*, onde é atribuída o domínio a ser abrangido, a conexão com todas as interfaces disponíveis (*htonl(INADDR_ANY)*) e a porta utilizada (*htons(port)*). A porta padrão em nosso programa é a porta 2002. Mas é possível entrar com as portas mais comuns, como 8080, 8060, 8090.

Tendo em vista que a conexão foi feita com sucesso, o próximo passo é verificar se o socket pode realizar a escuta (listen). Caso esteja tudo certo, é colocado então um laço infinito cujo possiuiá o accept e então realiza a leitura e escrita do arquivo da requisição feita pelo novo socket que é criado.

Logo após essas modificações o socket criado é fechado.

Rotina read_file

A rotina *read_file* é responsável por ler o conteúdo do arquivo que está sendo requisitado. Primeiro é verificado se há algum arquivo para ler. Se houver, então é verificado o tamanho do arquivo para armazenar espaço no buffer para realizar a transferência. É necessário também a confirmação de que o arquivo é comum (ordinary file). Se não for, o arquivo será fechado sem nenhuma leitura feita. Caso o arquivo seja comum, então um buffer é alocado com o tamanho do arquivo para realizar a leitura tuilizando o comando read(arquivo,buffer,tamanho). Após essas verificações e ações no código o arquivo é fechado e o buffer é retornado para posterior manipulação.

Rotina read_request

A rotina *read_request* realiza a leitura do buffer como bytes para a variável bytes_read. Caso o buffer esteja vazio, não é realizada nenhuma leitura da requisição. Caso contrário, é adicionada uma flag "\0"ao fim do vetor buffer para determinar o fim dos dados no mesmo. Logo em seguida é feita a leitura com sscanf passando por parâmetro o buffer, o método (no caso deste trabalho é o método GET), a url da requisição e a versão do protocolo utilizado. Após a leitura do socket é feita uma interpretação da requisição do usuário. O retorno da função é o nome do arquivo que deve ser buscado pelo servidor.

Rotina send_response

A rotina basicamente lê uma imagem específica escrevendo o cabeçalho http e enviando a imagem no corpo. Se houver uma resposta do servidor com conte údo, é então montado o cabeçalho HTTP.

Cabeçalho HTTP - Mensagem de sucesso:

ARQUIVO ENVIADO COM SUCESSO! HTTP/1.0 200 OK
Conection: close
Data: Sun, 25 out. 2015 17:26:15 GMT
Server: Supergirls/ 1.0.0 (Windows)
Last Modified: Sun, 01 out 2015 17:26:15 GMT
Content_length: <file_size>
<nomedoArquivo.extensão>

Cabeçalho HTTP - Mensagem de erro:

ARQUIVO NÃO FOI ENVIADO!
HTTP/1.0 404 Not Found
Conection: close
Data: Sun, 25 out. 2015 17:26:15 GMT
Server: Supergirls/ 1.0.0 (Windows)
Last Modified: Sun, 01 out 2015 17:26:15 GMT
Content_length: <file_size>
<nomedoArquivo.extensão>

Juntamente com o cabeçalho é enviado o arquivo not_found do servidor. É padrão dos servidores enviar esse arquivo para não ter uma resposta à uma requisição vazia.

2.3 Servidor utilizando thread ou fork

2.3.1 Fluxo de vida de socket dentro do servidor utilizando thread ou fork

O fluxo de execução do servidor seguindo a estratégia de fork possui basicamente a seguinte estrutura:

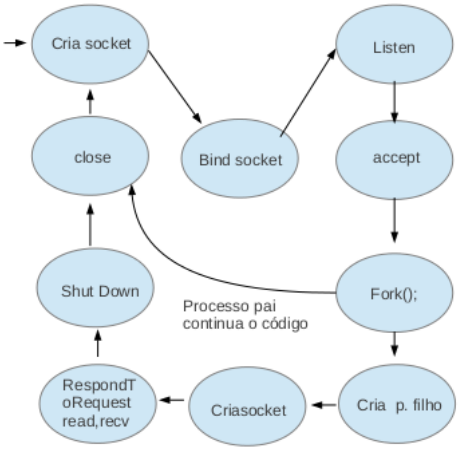


Figura 2: CICLO_FORK

Já o fluxo de execução do servidor seguindo a estratégia de disparar uma thread possui basicamente a seguinte estrutura:

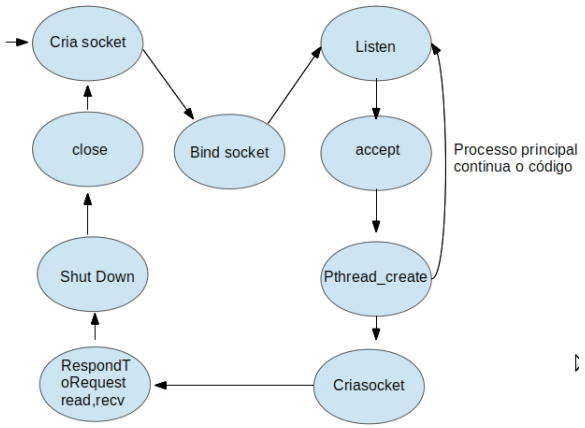


Figura 3: CICLO_THREAD

2.3.2 Descrição geral do programa de servidor utilizando thread ou fork

Estratégia utilizando fork para criar processo filho:

Para montar o servidor utilizando a estratégia de fork foi utilizado basicamente a função chamada *fork();*. Após a chamada desta função a função dispara um processo filho cujo trabalhará com a criação do novo socket para realizar as leituras e respostas das requisições enquanto o pai continua o código depois desse comando. Basicamente ele volta a escutar o servidor depois que fecha o processo filho.

Para isso, foi necessário um pid para verificar o status do processo filho. Caso o processo filho seja menor que zero, quer dizer que o fork obteve erro (ele não foi criado). Caso o pid seja zero, quer dizer que o fork retornou o processo filho com sucesso e então é possível atribuir a função respondToRequest para o processo filho responder às requisições.

Estratégia com uma thread:

A estratégia com uma thread se assimila ao processo com o fork, mas uma diferença é que não é um processo filho que é gerado, e sim um novo processo independente.

A thread é disparada para executar a função respondToRequestThread, a qual cria o socket para realizar a leitura e resposta da requisição do cliente (brownser). A thread recebe como parâmetros o id do socket e o id da própria thread.

2.4 Servidor utilizando threads e filas de requisições (Produtor-consumidor)

2.4.1 Fluxo de vida de socket dentro do servidor utilizando filas de requisições (Produtor-consumidor)

O fluxo do produtor consumidor segue a linha de raciocínio conforme a imagem a seguir:

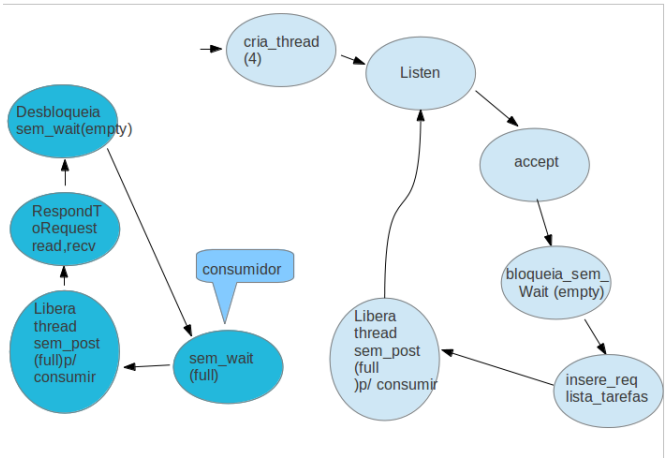


Figura 4: CICLO_PRODUTOR_CONSUMIDOR

2.4.2 Descrição geral do programa de servidor utilizando filas de requisições (Produtor-consumidor)

Para a implementação do produtor consumidor (ou lista de tarefas) foi necessário utilizar uma lista encadeada (filas de requisições) para manipular as diversas requisições existentes. O código necessita também de dois semáforos. O primeiro semáforo é o semáforo (sem_wait)para esperar até que uma requisição se complete (é inicializada como vazia e depois é preenchida com o consumidor para atender à requisição). o segundo semáforo (sem_post) é utilizado para alertar ao outro semáforo que já acabou a requisição, então ele é inicializado como cheio (full) e após o consumidor terminar a requisição ele é alterado para o estado vazio. Para que eles se comuniquem é necessário sempre passar a lista das requisições e o socket em si.

Por padrão o número de threads é 4. O sistema operacional escolhe qual thread será disparada no momento de atender a requisição do início da fila de tarefas.

Por padrão é necessário verificar se a lista de requisições está vazia. Se a lista não estiver vazia ele continua chamando a função de consumidor. Caso ela esteja vazia ela apenas termina a execução.

A função consumer implementa os atendimentos às chamadas quando uma thread necessita adormecer ou acordar. A variável de requisição para atender as chamadas recebe sempre o primeiro item da lista de requisições

(por isso o nome fila de tarefas). Caso a variável seja diferente de -1 então a função chama as funções de leitura e resposta e a função close de requisição descritas anteriormente no servidor iterativo.

2.5 Servidor concorrente utilizando função Select

2.5.1 Fluxo de vida de socket dentro do servidor utilizando função select (servidor concorrente)

O fluxo da implementação da estratégia select segue o fluxograma descrito a seguir:

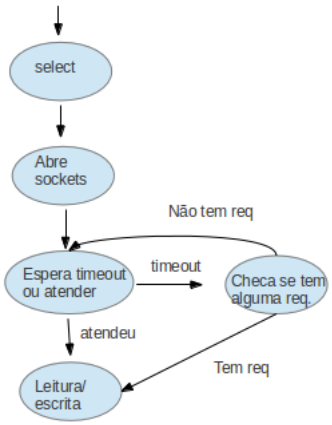


Figura 5: CICLO_SELECT

2.5.2 Descrição geral do programa de servidor utilizando função select (servidor concorrente)

o desenvolvimento do servidor utilizando a função select iniciou-se com a chamada select (assim como a chamada das funções padrões de criação de socket, bind e listen). A função select é uma chamada de sistema o qual é necessário passar uma série de sockets os quais a função seleciona a cada intervalo de tempo determinado (timeout determinado, em nosso caso cerca de 10 segundos). A quantidade de sockets selecionada inicialmente foi de 50 sockets. A função select abre os 50 sockets e permanece esperando as requisições. Caso haja a requisição ela é atendida pelo socket selecionado. Caso não haja, os sockets voltam a esperar.

3 Resultados

Os experimentos foram executados em uma máquina com as seguintes características: processador Intel® Core™ i5-3337U CPU @ 1.80GHz ×4, 8Gb de memória RAM e sistema operacional Linux Ubuntu 14.04 LTS 64 bits.

Para que os servidores implementados pudessem ser testados, utilizamos o Siege.

Siege é um teste de regressão http/https. É uma ferramenta utilizada para medir o desempenho do código, para que os desenvolvedores possam ver como o programa vai se comportar mediante um alto número de requisições. A duração de um Siege é medido em transações, ou seja, a soma do número de usuários simulados com o número de vezes que cada usuário simulado faz uma requisição ao servidor. Sendo assim, 20 usuários simultâneos e 50 vezes, equivale a 1000 transações (duração do teste). As medidas de desempenho que o Siege avalia são: tempo de execução, quantidade de dados transferidos (incluindo cabeçalho), tempo de resposta do servidor, taxa de transação, rendimento, número de conexões simultânea e o número de conexões encerradas com sucesso. Neste trabalho, avaliaremos apenas algumas dessas medidas.

3.1 Análise dos resultados

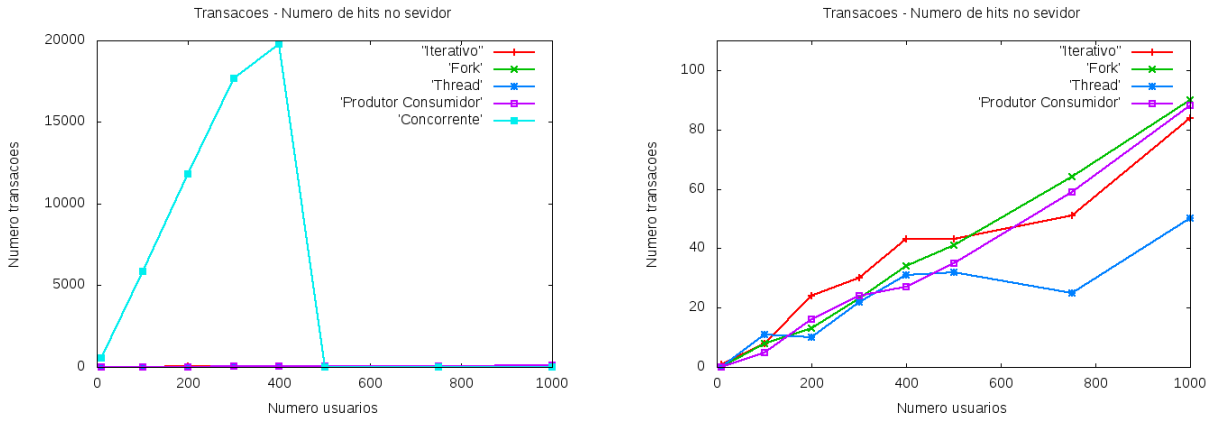
Para os nossos testes, variamos o número de usuários simultâneos para fazer as requisições, sendo: 10, 100, 200, 300, 400, 500, 750, 1000 usuários concorrentes e 30 repetições por requisição. Ressaltamos que o servidor concorrente não foi capaz de controlar as requisições de 500, 750 e 1000 usuários concorrentes, recusando as mesmas. Da mesma forma, o servidor com threads deu problema para requisições de 500 e 750 usuários.

3.1.1 Número de transações

O número de transações é o número de requisições pedidas ao servidor. Como por exemplo, colocamos para executar 100 usuários concorrentes e 30 vezes, o equivalente a 3000 transações.

Como mostrado no gráfico 6(a), o servidor concorrente foi o que mais recebeu requisições para 10, 100, 200, 300 e 400 usuários concorrentes. Entretanto, para 500, 750, e 1000 usuários o servidor concorrente não conseguiu controlar as requisições, recusando as mesmas. Para 500 e 750 requisições, o servidor iterativo obteve o maior número de transações e com 1000 requisições foi o servidor implementado com fork.

Como mostrado no gráfico 6(b), os servidores com fork e produtor-consumidor apresentaram um desempenho similar quanto ao número de de transações. Já os servidores iterativo e com thread variaram e apresentaram uma queda no número de transações ao aumentar o número de usuários concorrentes.



(a) (b)

Figura 6: Gráfico de número de transações

3.1.2 Tempo de execução e tempo de resposta

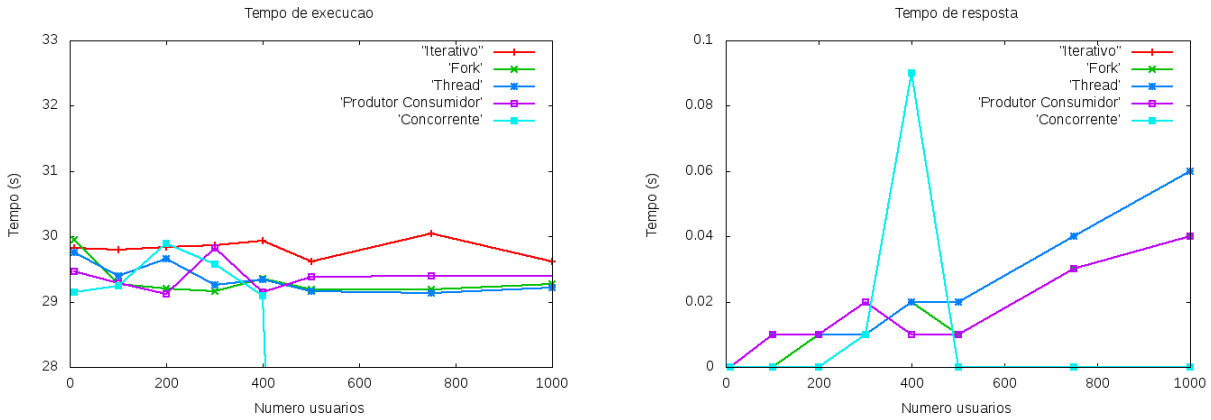
O tempo de duração é o tempo de execução de um teste. Essa métrica é medida a partir do tempo que o usuário executa um comando do Siege até a última transação simulada.

O tempo de resposta é a média de tempos que levou para responder cada requisição de usuário simulado.

Em nossos testes, o servidor iterativo gastou um maior tempo de execução para todas as variações de usuários concorrentes. Isso acontece pois o servidor iterativo atende uma requisição e enquanto isso as outras não são ouvidas, ou seja, correm o risco de não serem atendidas.

Como as demais estratégias recebem requisições e despacham a tarefa para que outro processo filho ou thread atenda a requisição, e assim esperar por novas requisições, ou seja, atender mais requisições por período de tempo.

O tempo de resposta por requisição variou entre 0.1s e 0.6s para todos os servidores exceto para o concorrente. Ressaltamos que o servidor concorrente não atendeu mais do que 500 requisições.



(a) Gráfico de tempo de duração

(b) Gráfico de tempo de resposta

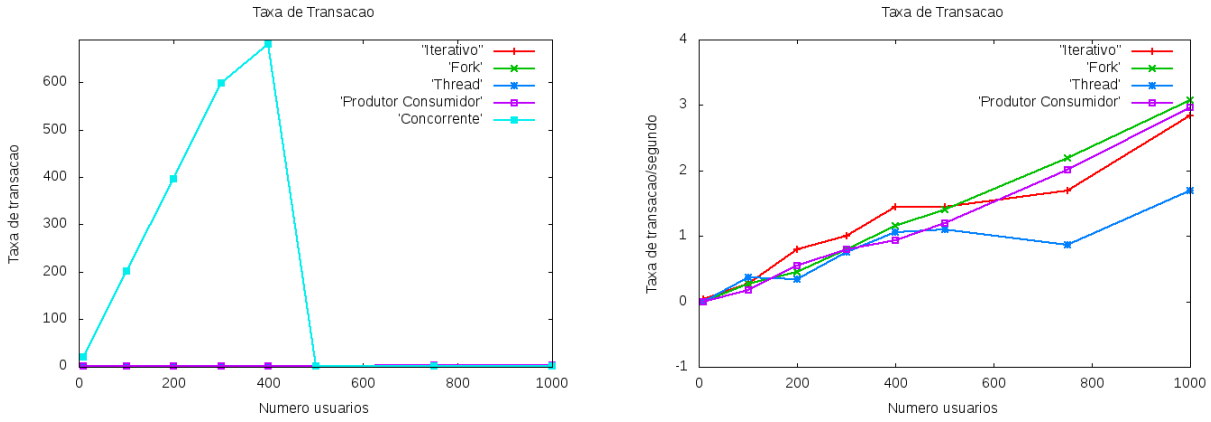
Figura 7: Gráfico do tempo de execução e de resposta

3.1.3 Taxa de transação

É a média do número de transações que o servidor atendeu por segundo.

Como mostrado no gráfico 8(a), o servidor concorrente possuiu maior taxa de transação por segundo. Enquanto os servidores iterativo, com fork, threads e fila de espera tiveram a taxa de transação variando de 0.1 a 3.0, o servidor concorrente possuiu taxas altissímas de transação variando de 200 a 680 para 100, 200, 300 e 400 usuários simultâneos. O pico no gráfico abaixo correspondente ao servidor concorrente obteve uma altíssima taxa de transação e nesse ponto o servidor parou de responder as requisições como mostrado no gráfico 6.

Como mostrado no gráfico 8(b), os servidores com fork e produtor-consumidor apresentaram um desempenho similar quanto à taxa de transação. Já os servidores iterativo e com thread variaram e apresentaram uma queda na taxa de transação ao aumentar o número de usuários concorrentes.



(a) (b)

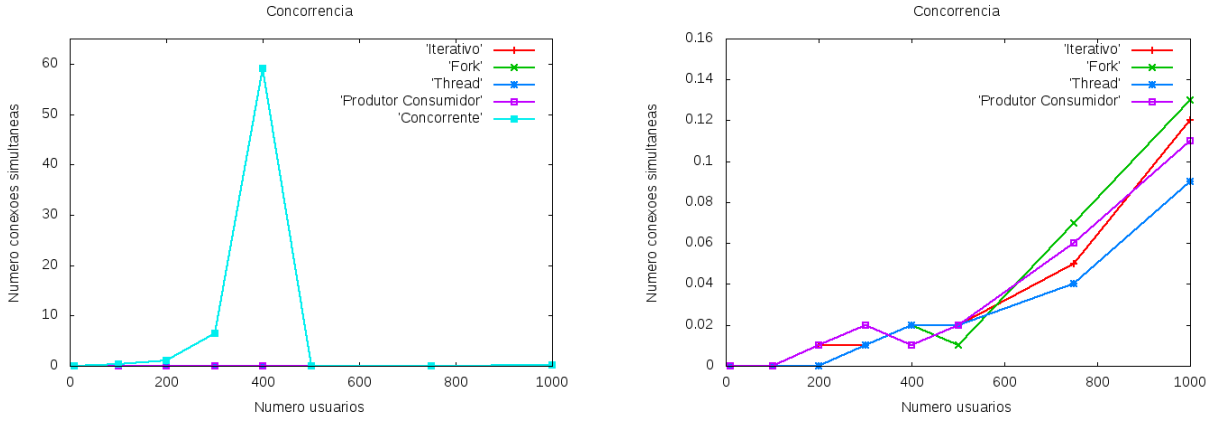
Figura 8: Gráfico de taxa de transação

3.1.4 Acesso simultâneo

É a média do número de conexões simultâneas, um número que aumenta a medida que o desempenho do servidor diminui.

Os servidores iterativo, com thread, fork e fila de espera obtiveram acesso simultâneo próximo a 0.0 a 0.11 para todas as variações de usuários concorrentes, demonstrando que os servidores possuem um bom desempenho. O servidor concorrente, entretanto, possuiu alta taxa de acesso simultâneo, chegando a 59.07 com 400 usuários concorrentes, demonstrando que o servidor não possuiu um bom desempenho. O pico no gráfico abaixo correspondente ao servidor concorrente obteve uma altíssima taxa de acesso simultâneo que corresponde aos resultados apresentados nos gráficos 6 e 8.

Como mostrado no gráfico 9(b), o servidor com fork apresentou o maior número de acessos simultâneos e o thread o menor. Já os servidores iterativo e o produtor-consumidor ficaram na média.



(a) (b)

Figura 9: Gráfico de acesso simultâneo

3.1.5 Número de transações com sucesso

Número de transações com sucesso é o número de vezes que o servidor retornou um código menor que 400. Sendo assim, redirecionamento é considerado uma transação com sucesso.

O servidor com fork demonstrou um maior número de transações com sucesso e o servidor com fila de espera (produtor-consumidor) teve comportamento similar. O servidor concorrente obteve menor número de transações bem-sucedidas, falhando na maioria dos casos correspondendo aos resultados acima.

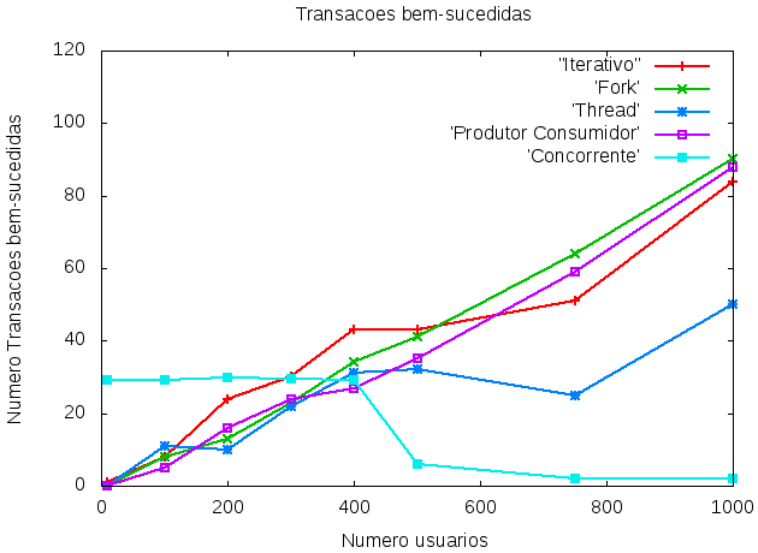


Figura 10: Gráfico de transações bem-sucedidas

4 Conclusão

O desenvolvimento deste trabalho nos permitiu desenvolver um sistema relativamente complexo, o sistema de servidor utilizando várias estratégias na linguagem C. Com ele pudemos aplicar as técnicas de desenvolvimento de servidores aprendidas em sala de aula, além de poder colocar em prática tanto o trabalho em equipe quanto o todo o processo do desenvolvimento de um sistema, que vai desde a análise à implementação.

Como resultados obtidos, observamos que o servidor com fork e fila de espera (produtor-consumidor) obtiveram melhor comportamento com relação às métricas avaliadas (descritas na seção anterior), principalmente com relação ao número de requisições atendidas para todas as variações testadas de usuários concorrentes. O servidor concorrente obteve o pior resultado e os demais ficaram na média.

5 Referências Bibliográficas

[1] http://www.martinbroadhurst.com/source/select-server.c.html
[2] https://www.joedog.org/siege-manual/
[3] http://cs.baylor.edu/ donahoo/practical/CSockets/textcode.html
[4] AEDS3: INTRODUÇÃO À PTHREADS(slides) - Isabella, Nicollas e Ramon
[5] Sistemas Operacionais: Processo(slides) - Rafael Sachetto Oliveira
[6] https://student.dei.uc.pt/ hpcosta/rc/rcMod7BAula2.pdf