Relatório - Trabalho 1 - Sistemas operacionais 2

Athos Lagemann
Fernando Spaniol
Gabriel Conte
Ricardo Sabedra

Este relatório tem como finalidade expor um trabalho realizado no Instituto de Informática UFRGS durante o período dos dias 07 de abril de 2017 até dia 8 de junho de 2017. O objetivo desta atividade consiste na implementação de um serviço semelhante ao Dropbox, este trabalho foi dividido em duas partes. Neste relatório está presente informações somente referenciando a parte inicial, a que se propõe a criação de um servidor (Dropbox) responsável por gerenciar arquivos de diversos usuários (clientes).

Foram utilizados como ambientes de testes, dois modelos principais, utilizando diferentes tipos de computadores, especificados a seguir:

MacBook Pro Retina

- Processador: 2.7GHz dual-core Intel Core i5 processor (Turbo Boost up to 3.1GHz) with 3MB shared L3 cache
- Memória RAM: 8GB of 1866MHz LPDDR3 onboard memory
- SO: Mac OS X
- Compilador: GCC 4.2.1

Notebook HP Pavillion X360

- Processador: 1.9GHz Intel M-Y510c
- Memória RAM: 4GB
- SO: Linux Ubuntu 16.04 - 64bits
- Compilador: GCC 5.4.0

Notebook Acer Aspire 5742Z

- Processador: 2.13GHz Intel Pentium P6200
- Memória RAM: 6GB
- SO: Linux Ubuntu 16.04 - 64 bits
- Compilador: GCC 5.4.0

Concorrência no servidor

Com o objetivo de realizar diferentes modelos de testes, utilizamos uma estrutura em rede local, com dois computadores utilizando a versão Mac OS. Configuramos um computador principal como o servidor e outro subsequente como cliente, este que pode fazer mais de uma conexão com o server para avaliarmos a consistência do software. Essa infraestrutura nos propiciou uma evolução constante e estável do trabalho em si. Procuramos também utilizar um mesmo computador como servidor e cliente próprio, assim utilizamos o modelo de localhost para fazer acesso local. Por definição, o servidor não

aceita mais do que 10 conexões simultâneas, de maneira que quando a vulga décima primeira conexão tente se conectar, seja automaticamente desligada, quando algum cliente se desconecta, um novo slot se abre.

A implementação do servidor foi criada de maneira que possam ser executados processos concorrentes. Para realizar este “feature”, utilizamos threads do sistema para dividir cada conexão do servidor, assim possibilitando que haja mais de um processo trabalhando ao mesmo tempo. Vale ressaltar que a concorrência se deve ao nível lógico do programa e não ao nível físico, caso fosse feito com bibliotecas de processamento paralelo. Tendo em mente que o protocolo de camada de transporte usado foi TCP e que o modelo TCP exige que se crie uma conexão para cada cliente, estas threads foram criadas assim que cada cliente foi conectado e obteve seu socket.

Sincronização:

O único ponto do programa que foi necessário garantir a sincronização foi na parte do cliente referente a disputa entre a thread principal e a thread apelidada aqui de “Daemon”. Quando a daemon estava tentando realizar a sincronização dos arquivos, a thread principal precisava estar bloqueada, sendo assim, foi utilizada uma variável de exclusão mútua de calibre global. Então quando alguma thread gostaria de usar a seção crítica do sistema, fazia-o por meios de atualizar o valor de tal variável, quando deixava a seção crítica, modificava a variável novamente. No servidor não foi preciso tal ferramenta pois os arquivos dos clientes se encontram em pastas diferentes e os sockets são individuais.

Como utilizar o dropBox:

Servidor:

Para a compilação foi usado GCC. Compila-se assim:

- gcc -o server dropboxServer.c

Em algum computador via terminal executar com o seguinte comando:

- ./server <ip>

Se um ip não for fornecido, o ip 127.0.0.1 será usado como default. A porta usada é a 53000 e não pode ser alterada.

Cliente:

Para compilação foi usado GCC. Compila-se assim:

- gcc -o client dropboxClient.c

Em qualquer computador da rede, usa-se o seguinte comando para executar:

- ./client <usuario> <ip>

Não existe maneira default neste caso, tem que ser o ip correto. O cliente, bem como o servidor, sempre se conectam na porta 53000000. Se um usuário ainda não existir, o servidor criará uma nova pasta para ele.

Ao se conectar com o servidor o usuário tem 4 opções de ações:

- Digitando 1, ele forçará a sincronização do seu folder. (Algo que a thread "daemon" já realiza a cada 10 segundos.
- Digitando 2, ele terá a opção de enviar um arquivo para o servidor, tendo então que fornecer o nome do mesmo.
- Digitando 3, ele receberá um arquivo do servidor, tendo também que informar o nome.
- Digitando 4, ele receberá uma lista dos arquivos que se encontram presentes em sua pasta no servidor.
- Para se desconectar do servidor, basta selecionar a opção 0.

Importante salientar que a porta utilizada pelo nosso programa é a 53000, tal opção não foi disponibilizada para mudança.

Estruturas e funções adicionais implementadas

Funções no server:

- *conta_conexoes_usuario:* Realiza a contagem do número de conexões ativas do usuário naquele instante de tempo, portanto, possui a habilidade de limitar a quantidade máxima das mesmas.
- criaSocketServidor: Função auxiliar que cria o socket do servidor
- *cria_pasta_usuario:* Ao receber uma conexão do usuário, cria uma pasta para o mesmo caso ele não possua uma.
- *atendeCliente:* Recebe a conexão do cliente e faz os "setups" iniciais.
- *list_files_server:* Lista os arquivos que o usuário possui no servidor.
- *receive_file*: Função que realiza todo o processo de recebimento de arquivos do cliente.
- *receive_file_sync*: Analoga ao receive_file, mas ao final do seu tempo de execução, envia o tempo obtido atraves da estrutura stat, de modo que o sync do cliente saiba quando foi modificado o arquivo
- *send_file_servidor*: Envia um arquivo do servidor para o cliente.
- *send_time_modified*: Envia quando um dado arquivo foi modificado pela última vez no servidor.

Funções no cliente:

- *get_info*: Recebe a opção de ação a ser escolhida pelo cliente na interface do terminal.
- *send_file_sync*: Chamada quando o “daemon” enviar um arquivo ao server, assim não necessita pedir requisição de “input” para usuário como acontece na função, “send_file”.
- *daemonMain*: Inicializa a thread que irá sincronizar os arquivos do cliente com os do servidor.
- *connect_server*: Função que realiza a conexão com o servidor.
- *sync_client*: Função que recebe uma lista dos arquivos do cliente no servidor e compara se o arquivo no cliente ou no servidor foi modificado e toma as devidas providências.
- *send_file_cliente*: Faz o envio de um arquivo para o servidor.
- *get_file*: Recebe um arquivo do servidor.
- *list_files*: Imprime os arquivos presentes em sua pasta.
- *close_connection*: Fecha a conexão com o servidor.
- *get_file_sync*: Análogo ao get_file, mas recebe o tempo no qual o servidor recebeu o arquivo para comparações futuras.

Primitivas de comunicação

Para realizar a comunicação entre processos, escolhemos sockets como método principal, o qual, também preferimos por implementar o protocolo TCP/IP para realizar uma troca de mensagens mais estável e confiável entre os nossos clientes e o servidor. Abaixo na “Imagem 1”, representa o modelo esquemático de comunicação TCP.

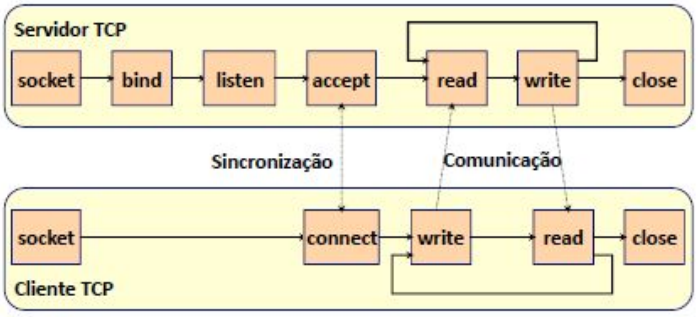

Imagem 1 - Modelo de comunicação TCP

Explicação das subdivisões do protocolo TCP:

- *Socket:* abstração de um ponto de comunicação (endpoint).
- *Agente de binding:* agente de amarração ou servidor de nomes, permite que o cliente obtenha um handler para acessar o serviço. Anexa um endereço local a um socket.
- *Listen:* Server fica disponível para aceitar conexões.
- *Accept:* Bloqueia o responsável pela chamada até uma tentativa de conexão ser recebida.

- *Connect*: Tenta estabelecer uma conexão entre cliente e servidor.
- *Send*: Envia dados através da conexão.
- *Receive*: Recebe dados através da conexão.
- *Close*: Encerra a conexão entre cliente e servidor.

Principais características do protocolo TCP/IP

- Permite comunicação bidirecional.
- Confiabilidade da comunicação.
- Baseada em abstração de fluxo (stream).
- Não define limites de tamanho para as mensagens, (tamanho variável).
- Garantia de entrega.
- Entrega ordenada.
- Não-duplicada.

Problemas encontrados

Ao longo do trabalho, encontramos vários problemas diversos que foram corretamente contornados para resultar na nossa solução final. Houve problemas de encontrar a melhor maneira de fazer a sincronização dos folders. Resolvemos utilizando a estrutura “stat” dos arquivos, comparando seus tempos e guardando um tempo local de quando a última tentativa de sincronização foi realizada. Caso não tivéssemos implementado essa solução, o cliente e o servidor acabavam por se alternarem no envio versões de seus arquivos. Pois toda vez que isto acontecia, a versão do que recebeu constava como mais nova. Outro problema encontrado foi de como garantir que a thread de sincronização, chamada de “Daemon”, não afetasse a thread principal sendo operada pelo usuário. Isto foi resolvido usando exclusão mútua.
Houve também outra situação com maior relevância de problemas que tivemos que contornar. Aconteceu que, indefinidamente, quando vários “sends” eram realizados em sequência, no momento em que o outro lado da conexão tentava executar um “recv”, reconhecia tudo como se fosse somente um. Para solucionar isto, foi realizado uma espécie de “ACK”, (acknowledge), como usado no TCP, para garantir que cada informação foi lida individualmente.

Conclusão

Este trabalho trouxe uma grande carga de conhecimento, de como na prática é possível utilizar os conhecimentos adquiridos em aula sobre a primeira parte da matéria. Principalmente referente a utilização de “Threads”, “sockets”, protocolo TCP, comunicação no modelo cliente servidor e sincronização de processos.