Backup Eficiente (Sobusrv / Sobucli) 2015/2016

Grupo: 28
Carlos Silva - A66553
Cláudio Ferreira - A48162
Gustavo Santos - A64385

Universidade do Minho
Escola de Engenharia

Universidade do Minho.
Projecto desenvolvido no âmbito da Unidade Curricular de “Sistemas Operativos” - MIEI
2015/2016

Problema Apresentado:

Neste projeto foi-nos apresentado o desafio de criar uma interface servidor-cliente que fosse capaz de realizar cópias de ficheiros e o restauro dos mesmos com um certo grau de segurança e confidêncialidade.

Método de ataque:

Para realizar o código referente a este projeto decidimos subdividir o códigos em módulos.

- **Cliente :** Programa que corre no processo cliente.
- **Server :** Programa que corre no servidor.
- **TreeA :** Estrutura AVL para salvaguardar dados de autênticação de utilizadores.
- **LigarServer:** Programa para iniciar o servidor.

Implementação dos módulos:

Cliente

Decidimos dar um menu e shell minimalista ao cliente, que só aceita certos comandos de modo a não poder "injetar" comandos prejudiciais à segurança dos dados no servidor. No entanto não pensamos que seja suficiente proteger neste módulo, por isso também criamos restrições no server de modo a seriar comandos fidedignos.

Temos a opção de fazer login ou registar no menu principal, e após um login bem sucedido procedemos à nossa shell que transmite as mensagens pelo FIFO para o server ler.

Os processos do cliente estão ainda preparados para receber os seguintes sinais:

- SIGUSR1 (10): para indicar que um ficheiro foi copiado.
- SIGUSR2 (12): para indicar que um ficheiro foi recuperado.
- SIGQUIT(3): para evitar que o cliente se desligue com este tipo de sinal.
- SIGTSTP(20): para indicar que a password inserida esta errada.
- SIGABRT(6): para indicar que utilizador ja está registado.

- SIGINT(2): para indicar registo bem sucedido.
- SIGTERM(15): para indicar login com sucesso.
- SIGFPE(8): para indicar que utilizador nao existe.

Server

No servidor implementa-mos métodos de autênticação de credênciais e um interpretador de comandos provenientes do FIFO. O server deveria dividir-se no máximo em 6 “processos”, 1 principal para analizar o FIFO, 5 consequentes para escalonar o trabalho das cópias entre si.

Esta abordagem permitiría a concorrência no servidor tal como reduziria o tempo de resposta aos diversos clientes que podem estar a aceder ao mesmo tempo.

Este servidor está preparado para receber alguns tipos de mensagens

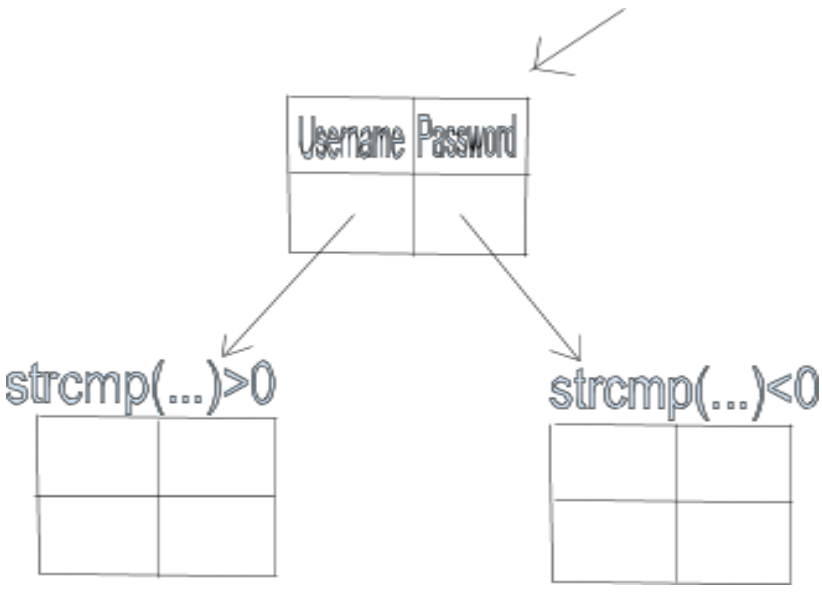
(char *) via FIFO, em que foi utilizado “:” como delimitador de campos, entre as quais:

- sobucli:reg:fifo:username:pass -> pedido de registar um utilizador
- sobucli:log:fifo:username:pass -> pedido de efetuar login
- username:sobucli:backup:fifo:filepath -> pedido de backup de ficheiro
- username:sobucli:restore:fifo:filepath -> pedido de restauro de ficheiro

Todas estas mensagens têm o 2º campo para indicar o tipo de mensagem no caso das mensagens para entrar no servidor, que no caso das mensagens para pedidos de backup e restauro é o 3º campo.

TreeA

Para guardar as credenciais dos utilizadores registados no sistema, utilizamos uma estrutura de dados AVL com os registos ordenados alfabeticamente.



*Critério do strcmp(Node->Username, Username).

LigarServer

Como é explicitamente pedido no enunciado que o servidor deve ser iniciado com o comando sobusrv, o grupo decidiu fazer um terceiro programa (ligarServer) que trata de assegurar que o servidor se mantém ligado. Este programa é bem simples, e está preparado para ignorar sinais do tipo SIGINT e SIGQUIT (para evitar morrer facilmente). É usado também sinais do tipo SIGALRM para ir verificando de 3 em 3 segundos se o servidor está "vivo". A técnica para verificar isso é simples pois apenas se conta o número de linhas do comando "pidstat -urdh -C server" (se tiver menos de 4 linhas é porque o servidor está desligado).

Valorização:

- **Delete:** Para esta funcionalidade pensamos criar uma estrutura que armazenasse os meta-dados dos ficheiros que estaríam ligados a uma certa cópia de segurança, consequentemente fariamos um prompt de confirmação da eliminação no caso de se querer proseguir com tal operação.

- **Gc:** Neste caso consideramos uma busca dos ficheiros presentes na tabela dos metadados e salvaguarda das suas referências no processo que realizasse o comando. Posteriormente efectuarse-ia uma sequência de comandos:
    - "mkdir BACK"
    - "mv <ficheiroUsado> ./BACK"
    - "rm *"
    - "mv ./BACK/<ficheiroUsado> ."
    - "rm -fr BACK"

  Deste modo ser-nos-ia possível preservar os ficheiros constados em metadados e eliminar os restantes.

- **Diretorias:** Iríamos recorrer à criação na altura do backup de uma pasta com o $username na path do ficheiro. Posteriormente nos pedidos de recuperação utilizaríamos:
    - "find -type d $username"

  Deste modo poderíamos localizar a pasta do user em questão e procederíamos a operação na mesma.

- **<u>Sockets:</u>** Para este caso necessitávamos de uma reestruturação geral do programa. O servidor abria o socket ao abrigo do protocolo TCP (de modo a que o kernel trate da viabilidade das operações e fragmentação de ficheiros demasiado grandes para caberem no socket em si de uma só vez, ao contrário do UDP que não nos garante nenhum desses pontos). O cliente ligar-se-ia ao socket do servidor com o IP do mesmo e procederia a leitura de mensagens que fossem passadas no mesmo.

  No caso de enviar mensagem do Servidor -> Cliente utilizaríamos macros que contivessem valores de “saída” reconhecidos pelo Cliente de modo a poder interpretar se a operações foi concluída ou não. Isto seria feito no final da transmissão do Server -> Cliente e fecharíamos o socket do lado do Cliente (visto que não seria utilizado).

Conclusão:

No desenrolar deste projeto foi-nos possível perceber as dificuldades que existem na configuração de uma interface servidor-cliente que forneça a possibilidade de vários clientes interagirem com o mesmo.

A multiplexação poderá ser abordada de diferentes modos (neste caso com FIFOs no modo de ficheiros) de modo a o servidor poder gerênciar todos os comandos que lhe sejam impostos (idealmente) de um modo de verdadeira fila de espera que possa aceitar diversos comandos que não possa executar no momento mas que irá executar posteriormente (respeitando a metodologia do First In First Out).

Claro está que muitos pedidos poderão resultar em erros por isso o controle de erros é essencial para cuidar desses casos. Também terá que ser feito de modo a não permitir instruções ilegais no servidor que poderiam ser utilizadas de modo a recolher informações ou destruir informações que não deveriam ser acessadas (na mesma veia que o famoso “SQL Injection”).

Também temos em consideração que o maior bottleneck será na realidade a velocidade de escrita e leitura da memória neste projeto em concreto. O acesso lento a dados no servidor irá acrescentar delay ao resultado do cliente, o ideal seria utilizar hardware favorável a escrita rápida e leitura rápida de modo a otimizar esta interface.

O CPU também será um bottleneck a partir do momento que existam demasiados forks a serem criados (por isso temos que implementar um número limite de forks), outro elemento fulcral em projetos de larga escala (com mais de 1milhão de utilizadores) poderá vir a ser a RAM que poderia ser sobrecarregada pela estrutura que está em memória que gere a autênticação do cliente. Optamos por uma AVL porque seria a melhor em termos de tempos assimptóticos em casos médios ou piores casos de buscas na mesma, visto que o server carrega tudo de uma vez, a maioria do balanceamento da AVL decorrerá no boot do servidor e não durante a operação do mesmo (se formos a analizar em termos relativos).

Os recursos que temos disponíveis são importantes e por isso temos que otimizar o código de modo a evitar problemas que sairão caros no futuro (por exemplo “Memory Leaks”) por isso recorremos a uso de ferramentas como o “Valgrind” e “GDB” para analizar possíveis falhas no código, uma metodologia que no entanto nesta etapa da nossa formação não é muito abordada e por isso por vezes esquecida.