UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS DE TELECOMUNICAÇÕES

MARCELO VEIGA PEREIRA

SISTEMA DE COMPARTILHAMENTO DE ARQUIVOS DESENVOLVIDO EM LINGUAGEM JAVA

TRABALHO DE CONCLUSÃO DE CURSO

CURITIBA
2019

MARCELO VEIGA PEREIRA

SISTEMA DE COMPARTILHAMENTO DE ARQUIVOS DESENVOLVIDO EM LINGUAGEM JAVA

Trabalho de Conclusão de Curso de Graduação, apresentado ao Curso Superior de Tecnologia em Sistemas de Telecomunicações, do Departamento Acadêmico de Eletrônica – DAELN, da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Dr. Kleber Kendy Horikawa Nabas

CURITIBA
2019

TERMO DE APROVAÇÃO

MARCELO VEIGA PEREIRA

SISTEMA DE COMPARTILHAMENTO DE ARQUIVOS DESENVOLVIDO EM LINGUAGEM JAVA

Este trabalho de conclusão de curso foi apresentado no dia 25 de Fevereiro de 2019, como requisito parcial para obtenção do título de Tecnóloga em Sistemas de Telecomunicações, outorgado pela Universidade Tecnológica Federal do Paraná. O aluno foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Profa. Dra. Tânia Lúcia Monteiro
Coordenadora de Curso
Departamento Acadêmico de Eletrônica

_____
Prof. M.Sc. Sérgio Moribe
Responsável pela Atividade de Trabalho de Conclusão de Curso
Departamento Acadêmico de Eletrônica

BANCA EXAMINADORA

_____
Prof. M.Sc. Omero Francisco Bertol
UTFPR

_____
Prof. Dr. Joilson Alves Junior
UTFPR

_____
Prof. Dr. Kleber Kendy Horikawa Nabas
Orientador - UTFPR

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

RESUMO

PEREIRA, Marcelo Veiga. **Sistema de compartilhamento de arquivos desenvolvido em linguagem Java**. 2019. 71p. Trabalho de Conclusão de Curso (Curso Superior de Tecnologia em Sistemas de Telecomunicações), Departamento Acadêmico de Eletrônica, Universidade Tecnológica Federal do Paraná. Curitiba, 2019.

A linguagem de programação Java dispõe de classes que possibilitam a comunicação de computadores em rede, utilizando sockets. Classes auxiliares para a leitura e escrita em fluxo de dados permitem comunicação em formato de texto e binário. Este trabalho explora a utilização de sockets através do desenvolvimento de uma série de programas cliente e servidor com diferentes formas de comunicação até a transferência e compartilhamento de arquivos binários utilizando sockets protegidos por criptografia.

Palavras chaves: Java. Socket. Servidor. Protocolo.

ABSTRACT

PEREIRA, Marcelo Veiga. **File sharing system developed in Java language**. 2019. 71p. Trabalho de Conclusão de Curso (Curso Superior de Tecnologia em Sistemas de Telecomunicações), Departamento Acadêmico de Eletrônica, Universidade Tecnológica Federal do Paraná. Curitiba, 2019.

The Java programming language has classes that enable the communication of computers in a network, using sockets. Auxiliary classes for reading and writing in data stream allow communication in text and binary format. This work explores the use of sockets through the development of a series of client and server programs with different forms of communication until the transfer and sharing of binary files using sockets protected by cryptography.

Keywords: Java. Socket. Server. Protocol.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

CRC	*Cyclic Redundancy Check*
FTP	*File Transfer Protocol*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
IP	*Internet Protocol*
ISO	*International Standards Organization*
JDK	*Java Development Kit*
JVM	*Java Virtual Machine*
LAN	*Local Area Network*
MAN	*Metropolitan Area Network*
OSI	*Open Systems Interconnection*
PNG	*Portable Network Graphics*
RTP	*Real-Time Transport Protocol*
SMTP	*Simple Mail Transfer Protocol*
SSL	*Secure Socket Layer*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
WAN	*Wide Area Network*

SUMÁRIO

1 INTRODUÇÃO

O protocolo *File Transfer Protocol* (FTP), ocupava a maior parte do tráfego de transferência de arquivos entre computadores e e-mail (TANENBAUM, 2011). Posteriormente o compartilhamento de arquivos pessoais passou a ser realizado também por e-mails, enquanto o FTP era utilizado para manter arquivos centralizados ou de uso em comum.

Com o advento da Web, arquivos também podiam ser disponibilizado para download na web via protocolo *Hypertext Transfer Protocol* (HTTP). Arquivos poderiam ser distribuídos, a partir de um servidor web central, diversos usuários. Posteriormente formas mais elaboradas para compartilhamento foram desenvolvidas e ficaram populares como, por exemplo, os softwares de compartilhamento Napster e eMule, que independem de um servidor central e utilizam uma arquitetura pier-to-pier para realizar o compartilhamento não centralizado de arquivos.

Atualmente têm-se disponíveis sistemas de compartilhamento de arquivos como DropBox, OneDrive e GoogleDrive, entre vários outros, que oferecem serviço de armazenamento virtual em nuvem e interface de gerenciamento Web, assim como software para sincronização de pastas compartilhadas e aplicativos para smartphones.

Soluções de armazenamento em nuvem, geralmente, oferecem espaço limitado reduzido gratuitamente, mas também oferecem opções de planos pagos anuais ou mensais que oferecem quantidade muito maior de espaço de armazenamento.

Qualquer destas ferramentas se torna bastante útil para a realização de cópia, compartilhamento entre computadores, entre diferentes dispositivos e ainda e backup de arquivos.

Embora existam diversas opções para transferir e compartilhar arquivos, eventualmente surge a necessidade da customização de suas funcionalidades e a possibilidade de integração com softwares proprietários.

2 PROBLEMA

A transferência de arquivos é uma operação de rotina quando se trabalha fazendo uso de computadores. São diversas alternativas para realizar a transferência de arquivos entre computadores.

Transferir arquivos entre computadores na rede utilizando as ferramentas disponibilizadas pelos sistemas operacionais é relativamente simples. Os sistemas operacionais oferecem funcionalidades de cópia de arquivos através de interface gráfica e também através de linhas de comandos.

A transferência de arquivos utilizando a interface gráfica é muito prática para o usuário, que pode selecionar os arquivos visualmente e efetivar a transferência utilizando movimentos do mouse. Mas transferir arquivos dessa forma periodicamente exige que o usuário repita a operação diversas vezes manualmente.

Transferir arquivos por linha de comando não é tão trivial, mas oferece uma série de parametrizações não disponíveis através interface gráfica. Através da linha de comando também é possível a utilização de *scripts* de automação que podem fazer uso de toda a flexibilidade das parametrizações por linha de comando.

Outra opção para transferir arquivos é a utilização de softwares proprietários, como *Rsync*, por exemplo, que podem ou não oferecer simultaneamente interface gráfica e parametrização por linha de comando.

Mesmo com a disponibilidade de softwares específicos e bibliotecas de terceiros, inclusive de código aberto, as bibliotecas das linguagem de programação disponibilizam vários componentes que, quando usados em conjunto permitem o desenvolvimento de funcionalidades mais complexas e completas.

O desenvolvimento de ferramentas que realizem a transferência automatizada e a devida checagem de integridade dos arquivos pode ser um desafio.

Este trabalho propõe o desenvolvimento de uma aplicação de transferência e compartilhamento de arquivos explorando as bibliotecas da linguagem Java.

3 JUSTIFICATIVA

Embora a linguagem Java não ofereça bibliotecas nativas para transferência de arquivos, existem disponíveis muitas bibliotecas de terceiros. Existem também vários programas que podem ser utilizados para transferência de arquivos, independente da tecnologia estes também podem, de forma mais limitada, ser integrados à outras linguagem de programação.

Recomenda-se o uso de biblioteca de terceiros quando não se dispõe de muito tempo para o desenvolvimento. Mas o tempo dedicado ao desenvolvimento de algumas funcionalidades é justificável quando surgem as necessidades de customização, ou correção de bugs.

O uso de bibliotecas de terceiros exige domínio de suas classes de objetos e métodos mas não exige que o desenvolvedor tenha conhecimento aprofundado do funcionamento da funcionalidade seus componentes internos. Isso pode se tornar um problema quando surge a necessidade de uma atualização, seja para corrigir um problema ou para implementação uma nova funcionalidade.

Embora bibliotecas de terceiros de código aberto sejam mantidas por comunidades online, seus desenvolvedores podem não fornece a solução para o problema em tempo hábil.

O código fonte dos componentes internos de uma biblioteca de código aberto podem não estar em conformidade com as boas práticas de programação, padrões de projetos e documentação do restante do projeto em que será utilizada.

Da mesma forma, a correção de um eventual problema em uma biblioteca cujos componentes internos não sejam de profundo conhecimento do desenvolvedor pode tomar muito mais tempo que o disponível para este possa estuda-lo todo minuciosamente.

Mesmo sendo possível o uso de biblioteca de terceiros, a linguagem Java disponibiliza as classes fundamentais para manipulação de arquivos e comunicação via rede que, quando utilizadas em conjunto, podem ser utilizadas para a transferência de arquivos via rede.

4 OBJETIVOS

4.1 OBJETIVO GERAL

Desenvolver um sistema de compartilhamentos de arquivo em linguagem Java utilizando *Secure Socket Layer* (SSL), realizando a checagem da integridade dos arquivos após a transferência dos arquivos.

4.2 OBJETIVOS ESPECÍFICOS

Para atender o objetivo geral neste trabalho de conclusão de curso os seguintes objetivos específicos serão abordados:

- Desenvolver um servidor de sockets que atenda um cliente de cada vez;
- Desenvolver um cliente em Java que faça uso da biblioteca de sockets;
- Realizar testes básicos de comunicação entre o cliente e o servidor;
- Implementar tratativas de erros de comunicação entre cliente e servidor;
- Implementar funcionalidades de transferências simples de arquivos;
- Realizar testes de transferências arquivos;
- Desenvolver um servidor capaz de atender diversos clientes simultaneamente;
- Elaborar um protocolo simplificado para transferência de arquivos;
- Realizar testes utilizando o protocolo de transferência de arquivos;
- Implementar funcionalidades de compartilhamento de arquivos;
- Realizar testes de compartilhamento de arquivos;
- Implementar configurações para o uso de com criptografia SSL;
- Inspecionar a comunicação entre sockets para verificar a criptografia;
- Implementar a checagem da integridade dos arquivos transferidos;
- Realizar testes complementares.

5 PROCEDIMENTOS METODOLÓGICOS

O projeto se será desenvolvido pelo método iterativo a partir de uma aplicação bastante simples que será incrementada em funcionalidades à cada iteração.

Para a implementação correta das bibliotecas e classes disponibilizadas pela linguagem Java será feito uso extensivo da documentação oficial da linguagem Java oferecido pelo site da Oracle.

Os equipamentos utilizados para desenvolvimento e testes iniciais serão computadores pessoais e a arquitetura de redes para testes de transferência de arquivos serão maquinas virtuais utilizando o software *Oracle VM VirtualBox*.

6 FUNDAMENTAÇÃO TEÓRICA

6.1 PROGRAMAÇÃO ORIENTADA A OBJETOS

A programação orientada a objetos procura aproximar o código desenvolvido, ou a solução de um problema, ao problema em si. Através da modelagem de objetos, e da inter-relação entre eles, tenta-se representar em linguagem de programação o objeto real que faz parte do problema que está se procurando resolver.

Os objetos são descritos a partir de classes, que são conjuntos de códigos que representam um objeto do domínio do problema. Classes não são objetos, são descrições de objetos, bem como suas propriedades e comportamentos e suas relações com outros objetos (SANTOS, 2014). A partir do projeto da classe pode-se criar diversos objetos de mesma estrutura e comportamento, que podem se diferenciar por suas propriedades.

Os objetos podem ter outros objetos como parte de suas propriedades ou podem ter variáveis do tipo nativo da linguagem, semelhante às variáveis da programação estruturada. O comportamento dos objetos são descritos em seus métodos que são semelhantes a funções da programação estruturada mas são inerentes aos objetos.

Na orientação a objetos é importante que essas propriedades não possam ser acessadas diretamente mas sim através de métodos que realizam as devidas verificações antes de acessa-las, garantindo que seus valores não sejam configurados fora do limite estabelecido pela modelagem do projeto. Chame-se encapsulamento a capacidade de ocultar e restringir o acesso às propriedades de objetos a partir de seus métodos e esta é uma das mais importantes características da programação orientada a objetos (SANTOS, 2014).

Outra importante característica da programação orientada a objetos é que seus objetos podem ser estendidos com novas propriedades e métodos através de um mecanismo chamado de herança. Essa característica permite que parte do código possa ser reaproveitada e reutilizada, não sendo necessário que objetos parecidos tenham que ser implementados do zero (SANTOS, 2014).

6.2 LINGUAGEM DE PROGRAMAÇÃO JAVA

Java é a linguagem de programação orientada a objetos, desenvolvida pela *Sun Microsystems*. Em 2009 a Sun foi adquirida pela Oracle que hoje é responsável pela manutenção e atualizações da linguagem Java, bem como diversos outros produtos.

A linguagem Java não dá liberdade para que sejam criadas funções ou variáveis globais fora de classes. Todas as variáveis e métodos precisam obrigatoriamente estar contidos dentro de classes (SANTOS, 2013).

Quando um programa desenvolvido em Java é compilado ele é convertido de linguagem Java para *bytecode*, que se aproxima do código de máquina mas não são específicos para serem executados por uma ou outra arquitetura de processador ou Sistema Operacional, mas sim específicos para serem executados na máquina virtual Java (*Java Virtual Machine* - JVM). A JVM interpreta o código compilado em *bytecode* e repassa para o sistema operacional para que sejam executados como se tivessem sido desenvolvidos para a arquitetura sobre o qual está sendo processado.

Como o *bytecode* não é especifico para uma arquitetura de hardware mas para a Máquina Virtual Java, o código Java compilado pode ser executado em qualquer sistema através da Máquina Virtual Java apropriada para este sistema. Um programa Java não precisa ser recompilado para cada sistema, basta que o código seja compilado uma vez em qualquer sistema e poderá ser executado em qualquer outro. Esta propriedade da linguagem Java é chamada de portabilidade (SANTOS, 2013).

Diferente de C e C++ em que o programador precisa se preocupar com a alocação dinâmica de memória e sua liberação, para que seu programa não consuma memória desnecessariamente, Java implementa um recurso de coleta (*Garbage Collector*) que é responsável para identificar e liberar espaços de memória não utilizada pelo programa em execução. Esse recurso simplifica muito o desenvolvimento de programas porque o programador não precisa alocar ou liberar memória manualmente (SANTOS, 2013).

Java também dispõe nativamente de diversas bibliotecas como bibliotecas para manipulação de entrada e saída, acesso à Internet, manipulação de texto, estruturas de dados e outras de utilidades diversas (SANTOS, 2013).

6.3 PROCESSOS E THREADS

6.3.1 Processo

Um programa de computador em execução é um processo. A diferença entre programa e processo é bastante sutil, basicamente um programa é chamado de processo somente enquanto está em execução no sistema operacional (TANENBAUM, 2016), então fora do contexto do sistema operacional não há processos, somente programas.

Programas como *Word*, *Excel* e *Paint* são exemplos de programas de computador. Quando estes programas são executados, eles são iniciados como processos no sistema operacional.

Um processo consome ou reserva parte da memória do computador para sua execução. A memória reservada é utilizada como armazenamento temporário durante a execução deste processo.

Sistemas operacionais mais antigos, como o *MS-DOS*, executavam apenas um programa por vez; não sendo possível a execução de dois programas ao mesmo tempo (SILBERCHATZ, 2015). Para iniciar a execução de um novo programa era preciso encerrar a execução do programa anterior.

Num contexto monotarefa, como do MS-DOS, o termo processo não se aplica porque o carregamento do programa era mais simplificado pois somente um programa podia ser executado de cada vez.

Com a evolução dos sistemas operacionais, ainda antes do advento de processadores com mais de um núcleo, surgiram sistemas multitarefa que eram capazes de executar dois ou mais programas de forma, aparentemente, simultânea. Isso era possível porque o sistema operacional alternava a execução muito rapidamente, dando a impressão que estavam sendo executados ao mesmo tempo (TANENBAUM, 2016).

Um sistema multitarefa precisa implementar controle de acesso a recursos, como proteção de memória, para impedir que dois processos tenham acesso a um mesmo recurso ao mesmo tempo. O acesso irrestrito à estes recursos poderiam corromper os arquivos e áreas de memória.

6.3.2 Thread

Threads são processos leves que executam tarefas. Um processo com apenas uma thread executa uma tarefa de cada vez, de forma serial. Processos com duas ou mais threads podem executar diversas tarefas em paralelo (SILBERCHATZ, 2015).

A criação de threads é menos custosa (SILBERCHATZ, 2015) para o sistema operacional e, por isso, mais rápida que a criação de processos (TANENBAUM, 2016). Programas que precisam executar diversas tarefas simultaneamente podem criar várias threads para segmentar essas tarefas e dividi-las entre as threads.

Um exemplo de aplicação em que threads desempenham um papel importante são servidores web (SILBERCHATZ, 2015). Servidores web precisam tratar diversas requisições de clientes e se não fossem capazes de trata-las simultaneamente teriam que atender uma requisição de cada vez.

Além disso, o acesso a alguns recursos, como dispositivos de entrada e saída, exigem que o processamento seja interrompido temporariamente, porque o recurso pode não estar disponível no momento da tentativa de acesso, fazendo que um processo precise esperar pela sua disponibilidade (TANENBAUM, 2016). Em sistemas que não permite processamento paralelo o tempo de processador é desperdiçado enquanto o processo espera pelo recurso.

No exemplo citado, quando um servidor recebe uma requisição para uma página web estática, que contenha imagens, ele precisa ler do disco o arquivo que contém a página e, posteriormente, ler os arquivos de imagem. Um outros dois mil clientes realizassem novas requisições precisariam esperar que todas as tarefas das solicitações anterior fossem concluída para que pudessem ser atendidos.

Servidores antigos tratavam múltiplas requisições iniciando um novo processos para tratar cada uma delas (SILBERCHATZ, 2015). Mas isso exigia mais recursos do sistema e por isso o número de requisições simultâneas era mais restrito.

Servidores modernos criam novas threads para o tratamento das requisições (SILBERCHATZ, 2015). Como threads são mais leves e mais sua criação rápidas, um número maior de requisições pode ser atendido sem que seja necessário um computador com maior velocidade de processamento.

Threads também são importante para aplicações sendo utilizada para separar a interface gráfica da parte lógica. E um programa single-thread, uma operação de compactação de arquivo deixaria de poderia responder as entradas do mouse ou

teclado até que a compactação fosse concluída (TANENBAUM, 2016). Este problema não ocorreria utilizando uma thread separada para a compactação e outra para a interface de usuário.

Uma outra grande vantagem das *threads* é que o fato de compartilharem entre si e com o processo que as instanciou, os recursos reservados para esse processo, como a área de memória e arquivos abertos (SILBERCHATZ, 2015).

Há mais simplicidades ao transferir arquivos ou outros recursos entre *threads* e, com isso, fica mais fácil coordenar o processamento das tarefas porque não é preciso que arquivos sejam fechados para serem transferidos entre processos, por exemplo.

6.4 REDES DE COMPUTADORES

6.4.1 Redes

Uma rede de computadores pode ser descrita como um conjunto de computadores autônomos interconectados por uma mesma tecnologia (TANENBAUM, 2011). Hoje em dia, essa rede interconecta não somente computadores, mas diversos outros dispositivos como câmeras de vídeo, consoles de jogos e smartphones (KUROSE; KEITH, 2013).

Redes de computadores podem ser classificadas por sua dimensão como rede: *Local Area Network* (LAN), *Metropolitan Area Network* (MAN) e *Wide Area Network* (WAN). Uma LAN é uma rede local que se estende dentro de uma residência ou escritório, limitando-se há um único prédio. São utilizadas para compartilhamento de arquivos ou impressoras (TANENBAUM, 2011). Com a popularização dos smartphones as redes locais passaram a ser implantadas cada vez mais com tecnologia sem fio fazendo uso de roteadores sem fio ou ponto de acesso.

Redes MAN tem uma área de cobertura maior que a LAN. Enquanto a rede LAN se limita a apenas um prédio, como uma residência; as redes MAN se estendem a outros prédios fazendo a interligação entre uma matriz de uma empresa e suas filiais dentro de uma mesma cidade ou cidades próximas. Os locais de rede podem ser interligadas por diferentes tecnologias, desde antenas de comunicação sem fio até a utilização de serviços dedicados de provedores de serviço de internet.

Redes WAN abrangem uma área muito maior que uma rede MAN, como todo um país ou até mesmo um continente (TANENBAUM, 2011). Estas redes geralmente

são formatas pela interligação de diversas outras redes fazendo uso de diversas tecnologias diferentes fazendo com que possam ser bastante complexas. As sub-redes que compõem uma rede WAN podem ser administradas por diferentes pessoas devido a quantidade de equipamentos utilizados e a quantidade de clientes conectados à rede.

Essas redes podem ser interligadas por equipamentos proprietários ou ser interligadas por uma rede de serviços terceirizada como provedores de serviço de internet. Provedores de serviço de internet interligam diversos clientes entre si e a internet (TANENBAUM, 2011).

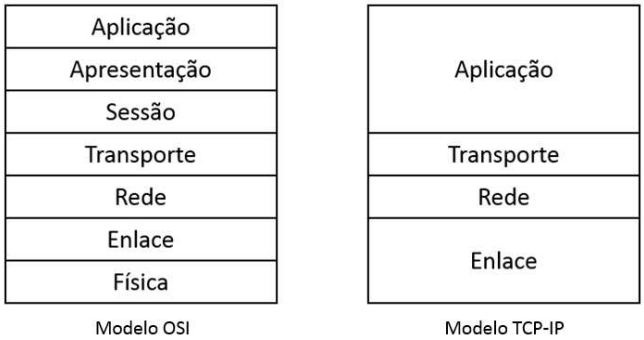
A internet não é considerada uma rede única mas um conjunto complexo formado por diversas redes interligadas através de diversos tipos de tecnologias e protocolos diferentes que opera sem um administrador específico e foi não foi criada a partir de um projeto específico (TANENBAUM, 2011).

6.4.2 Modelos de Referência

Para que fosse possível a interligação de redes por equipamentos de diferentes fabricantes, houve a necessidade de uma estrutura padronizada para o projeto de redes. Esse projeto dividiu a estrutura básica de rede em diferentes camadas de protocolos (KUROSE; KEITH, 2013).

A *International Standards Organization* (ISO) elaborou uma proposta para padronização internacional dos protocolos utilizados nas camadas de rede e desenvolveu o Modelo de Referência *Open Systems Interconnection* (OSI). O modelo OSI é formado por sete camadas bem definidas: Aplicação, Apresentação, Sessão, Transporte, Rede, Enlace e Física (Figura 1).

Figura 1 - Modelos de referência



Fonte: Autoria própria.

As camadas são organizadas umas sobre as outras e cada uma é responsável por um conjunto de funções e utiliza serviços da camada posicionada diretamente abaixo. Essa estrutura é conhecida também como Pilha de Protocolos (KUROSE; KEITH, 2013).

As informações que trafegam na rede passam por todas as camadas da mais alta até a mais baixa e posteriormente, vão da camada mais baixa até a mais alta em outro dispositivo. As camadas de mesmo nível devem implementar o mesmo protocolo para que possam se comunicar (TANENBAUM, 2011).

Os protocolo de rede estabelecidos como padrão é o *Internet Protocol* (IP) e *Transmission Control Protocol* (TCP). O IP é baseado em datagrama de até 64 KB, que pode ser dividido em pacotes menores, mas não garante a entrega. Esses pacotes trafegam pela rede e pela internet pro diversas rotas, possivelmente diferentes e são remontados quando chegam ao destino.

Como pacotes podem se perder ao longo da rede e como o protocolo IP não garante a entrega, é necessário que um recurso confiabilidade de entrega seja implementado em outra camada. O TCP, que pertence a camada de transporte e se posiciona logo acima da camada de Rede (Figura 1) implementa essa funcionalidade (TANENBAUM, 2016).

O protocolo TCP é responsável por entregar os dados na ordem correta e garantir a entrega usando mensagens de confirmação, enquanto o protocolo IP é responsável por identificar os computadores envolvidos na comunicação. Finalmente, a camada de enlace é responsável por realizar a entrega dos sinais para o meio físico de transmissão.

As detalhes da implementação desses protocolos é transparente para os processos envolvidos na comunicação. Os processos simplesmente enviam suas mensagens para os sockets e os sockets realiza as devidas chamadas ao sistema operacional para realizar o envio através da pilha de protocolos.

No modelo TCP/IP não inclui as camadas de sessão e apresentação. Por esse motivo, as funcionalidades dessas camadas devem ser incluídas na camada de aplicação quando necessário (TANENBAUM, 2011).

6.4.3 Protocolos da Camada de Aplicação

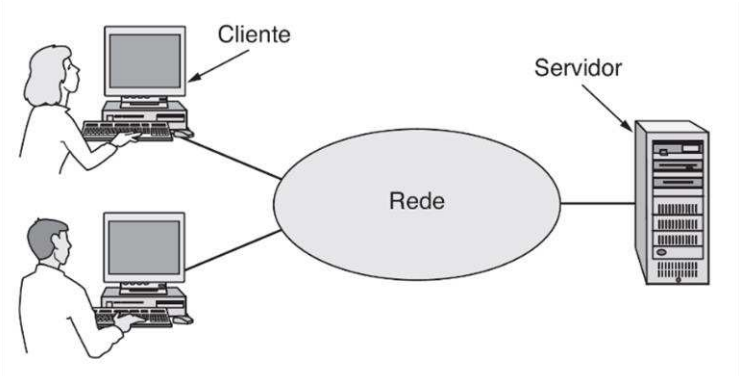
A camada de aplicação contém os protocolos de nível mais alto e comumente necessários para os usuários como: HTTP, usado para entrega de páginas Web; FTP, usado para transferência de arquivos; *Simple Mail Transfer Protocol* (SMTP) de correio eletrônico; e *Real-Time Transport Protocol* (RTP), usado para entrega mídia de voz ou vídeo em tempo real; são apenas alguns exemplos (TANENBAUM, 2011).

6.5 ARQUITETURA CLIENTE SERVIDOR

Em redes empresariais é muito comum que arquivos fiquem centralizados em servidores de arquivos para que possam ser acessados de outros computadores. Estes servidores, geralmente, possuem mais recursos que computadores pessoais para que possam responder a diversas requisições concorrentes com maior eficiência. Servidores devem ficar em área restrita e não devem ser utilizados por pessoas não autorizadas (TANENBAUM, 2011). Os arquivos disponíveis nos Servidores podem ser acessados pelos funcionários através de seus computadores que, nesse contexto, são chamados Clientes, que podem ser configuração mais simples.

Esse tipo de arquitetura é chamado de arquitetura Cliente-Servidor (Figura 2) e é o modelo mais comum utilizado em redes de computadores. Computadores não se comunicam, ou trocam arquivos entre si diretamente, mas sim através de um servidor central. Servidores de arquivos, Servidores Web, Servidores de Banco de dados e Servidores de E-mail se baseiam nessa arquitetura (TANENBAUM, 2011).

Figura 2 - Arquitetura cliente servidor

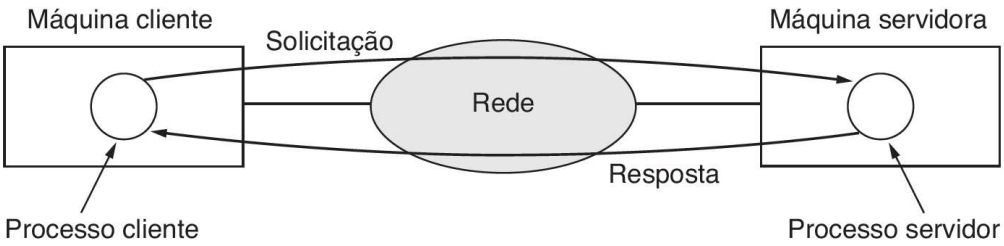


Fonte: Kurose e Keith (2013).

O fato de um computador ter maior poder de processamento não o classifica como um servidor; qualquer computador pode ser um servidor se estiver compartilhando algum recurso ou executando um processo que aguarda por requisições de outros computadores; mas é altamente recomendável que sejam computadores dedicados e apropriados para este fim.

Servidores ficam aguardando requisições de clientes; servidores Web, por exemplo, podem executar o Apache HTTP Server e aguardar requisições de páginas Web. Clientes são computadores que executam processos clientes que realizam requisições ao servidor (Figura 3), como o *Google Chrome* que realiza requisições de páginas a Servidores Web.

Figura 3 - Comunicação entre cliente e servidor



Fonte: Kurose e Keith (2013).

Servidores, normalmente, mantém um endereço IP fixo para facilitar seu acesso e o processo requisita ao sistema operacional uma porta de comunicação específica para cada serviço, servidores web utilizam a porta 80 (KUROSE; KEITH, 2013). O par formado pelo endereço IP e a porta de comunicação são chamados de socket (SILBERCHATZ, 2015) e são necessários tanto para o Servidor como para o Cliente. Todo o mecanismo de comunicação entre os processos do Servidor e do Cliente é realizado através de pares de *sockets*.

6.6 SOCKETS

Processos podem se comunicar através da rede enviando e recebendo mensagens através de uma interface de software denominada *socket* (KUROSE; KEITH, 2013).

Sockets representam extremidades na comunicação em rede e são identificados pelo endereço IP da interface de rede seguido do número da porta utilizada (SILBERCHATZ, 2015).

Servidores que implementam serviços específicos (FTP e HTTP) utilizam portas bem conhecidas (um servidor FTP utiliza a porta 21, um servidor Web ou HTTP, utiliza a porta 80). As portas abaixo de 1024 são consideradas bem conhecidas e devem ser usadas apenas para os serviços correspondentes (SILBERCHATZ, 2015). Todas as portas acima de 1024 são para uso geral e podem ser utilizadas por processos clientes ou para implementações de novos serviços.

Não é possível que dois processos no mesmo computador utilizem a mesma porta ao mesmo tempo. Quando um *socket* é aberto por um processo este deve ser único no computador e exclusivo para o processo que o abriu. Se outro processo iniciar um novo *socket* este obrigatoriamente terá uma porta diferente do processo anterior (SILBERCHATZ, 2015).

Na linguagem Java, *sockets* também são objetos e implementados na classe *Socket*. Basicamente as classes que implementam a manipulação de *sockets* pertencem a biblioteca *java.net*. Essa biblioteca contém classes para implementação de *sockets* TCP, orientados a conexão e *User Datagram Protocol* (UDP), não orientados a conexão (SILBERCHATZ, 2015).

6.7 CAMADA DE SOCKETS SEGUROS (SECURE SOCKET LAYER - SSL)

Quando informações trafegam na rede através dos protocolos TCP não recebem nenhuma codificação. As informações trafegam pela rede e são entregues ao socket do processo de destino exatamente da mesma forma que foram enviadas pelo socket do processo de origem. Como as informações não são codificadas, uma senha ou número de cartão de crédito poderão ser lidos por usuários mal intencionados posicionados em algum ponto da rede que permita o acesso ao fluxo de rede.

A necessidade de transferir informações de forma segura levou ao desenvolvimento aperfeiçoamentos sobre o protocolo TCP denominado Camada de *Socket* Seguros (*Secure Sockets Layer* - SSL). O SSL adiciona uma camada de criptografia ao protocolo TCP garantindo a segurança das comunicação entre processos enquanto as informações trafegam na rede (KUROSE; KEITH, 2013).

As bibliotecas de socket tradicionais não oferecem criptografia. Normalmente o conjunto de bibliotecas disponíveis para as linguagens de programação possuem bibliotecas próprias para a implementação de sockets seguros. Para que uma aplicação faça uso dos serviços disponíveis pelo SSL é preciso fazer uso de bibliotecas específicas que instanciam sockets com as funcionalidades do SSL (KUROSE; KEITH, 2013).

Sem criptografia, alguém que consiga acesso aos dados que trafegam na rede poderia ter acessos às senhas e qualquer outra informação sensível.

O SSL faz o intermédio entre a aplicação e o protocolo TCP. As informações enviadas pelo processo são criptografadas pelo SSL antes de serem entregues ao protocolo TCP, e por isso antes de serem enviadas à rede. Se alguém conseguir acesso à qualquer ponto da rede entre o cliente e o servidor e tentar acessar as informações da comunicação, não conseguirá acesso fácil aos dados devido a esses estarem criptografados (TANENBAUM, 2011).

Sites de compra, por exemplo, utilizam o SSL para dar segurança ao protocolo HTTP. Quando está utilizando SSL o protocolo o protocolo HTTP passa a ser chamado de *Hypertext Transfer Protocol Secure* (HTTPS) e, ao invés da porta 80, faz uso da porta 443. Essas diferenças são para não haver confusão entre os protocolos utilizados na conexão. O protocolo HTTPS tem as camadas adicionais referentes a criptografia, que também precisam ser tratadas no cliente. Utiliza-se outra porta para que não sejam iniciadas conexões sem criptografia à um servidor que espera conexões criptografadas (KUROSE; KEITH, 2013).

Como o SSL não é implementado na aplicação mas sobre o protocolo TCP, qualquer aplicação que utilize as bibliotecas de socket seguro pode utilizar os serviços de segurança do SSL. Deve ficar claro que o SSL oferece funcionalidades adicionais ao protocolo TCP, mas não o substitui como protocolo (TANENBAUM, 2011).

O SSL utiliza certificados para garantir a autenticidade do servidor e, em alguns casos, também do cliente. Estes certificados contém as chaves que serão utilizadas no processo de criptografia. No processo inicial de conexão, o servidor envia ao cliente o certificado que contem sua chave privada. O cliente deverá conferir se foi emitido por uma entidade certificadora ou se possui o certificado em sua base de certificados autorizados antes de dar continuidade a conexão.

A linguagem Java mantem uma base de certificados confiáveis, ou *truststore*, no arquivo *cacerts*. O certificado do servidor deve ser verificado contra esta base antes de estabelecer o fluxo de dados entre cliente e servidor. A implementação de *socket* SSL se responsabiliza por realizar estas verificações.

Novos certificados podem ser adicionados à *truststore*, como no caso de um certificado criado pelo próprio usuário e, nesse caso, não emitido por uma entidade certificadora. Java disponibiliza a ferramenta *keytool* para criação e gerenciamento de certificados.

Também é possível criar novas *truststores* e utiliza-las em Java através de linha de comando. Este recurso é interessantes para uso em desenvolvimento e para realização em testes, por não ser necessário alterar a *truststore* padrão do Java e por permitir, facilmente, o compartilhamento da *truststores* com outros computadores.

7 DESENVOLVIMENTO

Os desenvolvimento dos códigos disponíveis neste trabalho são apenas exemplos e tiveram como objetivo a simplicidade e facilidade de leitura e entendimento. Foram evitados códigos muito longos ou muito complexos e bibliotecas não nativas da linguagem Java, assim como excessivas checagens de erro. O tratamento de exceções foi mantido genérico e em blocos *try/catch* abrangentes. Na maioria das vezes, os programas, simplesmente, encerram sua execução quando ocorre uma exceção. O foco do código apresentado está na utilização de sockets de maneiras diferentes.

7.1 SERVIDOR SIMPLES

O desenvolvimento foi iniciado a partir do programa *SimpleServer* (Apêndice A), um servidor simples que, ao receber um texto, o exibe no terminal e retorna ao programa cliente entre parênteses.

Este programa utiliza um objeto do tipo *ServerSocket* como base para seu funcionamento. Objetos da classe *ServerSocket* criam um socket que ficam esperando conexões na porta informada como parâmetro ao construtor da classe. Foi utilizada a porta 9000 como padrão, mas outra porta pode ser utilizada, se passada como parâmetro ao programa.

O método *accept* da classe *ServerSocket* coloca o servidor em estado de espera por conexões e retorna um objeto do tipo *Socket* quando uma conexão é atendida. No servidor, o objeto *Socket* representará o cliente e qualquer informação escrita nesse socket será enviada ao programa cliente.

Este programa servidor foi desenvolvido para receber e enviar linhas de texto e, para isso, o objeto do tipo *InputStreamReader* foi utilizado em conjunto com um objeto *BufferedReader*, que dispõe do método *readLine* que permite a leitura de linhas de texto do *stream*.

Para o envio de dados para o cliente, foi utilizado um objeto do tipo *PrintWriter* que dispõe do método que permite o envio de linhas de texto para *streams*.

Este programa recebe uma linha de texto, envia uma linha de texto de resposta e então encerra a conexão.

Para testar o programa é necessário um programa cliente, que pode ser um cliente de *Telnet*. Para realizar testes básicos de comunicação foi utilizado o cliente de *Telnet* do Windows (Quadro 1). Foi necessária a configuração do parâmetro *localecho* para visualização dos caracteres enviados, caso contrário a linha 6 do Quadro 1 não seria visível. Outros clientes de *Telnet* podem não ter este problema.

Quadro 1 - Acesso ao servidor SimpleServer através de um cliente de Telnet

```
1  >telnet
2  telnet> set localecho
3  Local echo on
4  telnet>open localhost 9000
5  Connecting To localhost...
6  Teste
7  (teste)
8  Connection to host lost.
9  >
```

Fonte: Autoria própria.

É possível realizar conexões ao servidor a partir de vários programas cliente como, por exemplo, um navegador Web. O servidor pode ser acessado por um navegador Web através do endereço '*http://localhost:9000*'. Como o protocolo HTTP é baseado em texto o servidor receberá uma requisição de página Web e exibirá o texto recebido '*GET / HTTP/1.1'* no terminal.

Observa-se que apenas a primeira linha da requisição HTTP é recebida devido ao método *readline* retornar apenas uma linha de texto. É possível receber as demais linhas da requisição chamando o método *readline* mais vezes ou até que este retorne um texto vazio.

Quanto à mensagem exibida no navegador Web, dependerá do navegador utilizado. O navegador *Chrome* apresentou uma mensagem de erro de informando que a resposta HTTP era inválida. Já o *Firefox* exibiu o texto '(*GET / HTTP/1.1)'* exatamente como a resposta enviada pelo servidor.

O programa cliente desenvolvido, *SimpleClient* (Apêndice B), deve receber como parâmetros o nome do *host* e, opcionalmente, o número da porta do servidor. Diferente do programa servidor que usa um objeto da classe *ServerSocket* para receber conexões, o programa cliente *SimpleClient* utiliza diretamente um objeto do tipo *Socket* que é utilizado para iniciar conexões.

Como o servidor e cliente utilizarão linhas de texto como base para sua comunicação, os *streams* de entrada e saída devem ser tratados da mesma forma.

O programa cliente não espera por conexões mas, nesse caso, espera que seja entrada uma linha de texto no terminal para que possa enviá-la ao servidor. Para isso, um outro objeto *BufferedReader*, recebe o stream do terminal *System.in* que, ao contrário de *System.out*, é utilizado para ler entradas de texto do terminal.

Um ponto interessante a considerar é o fato do programa Cliente esperar que uma linha de texto seja entrada no terminal, envia o texto ao servidor e, então espera por dados do servidor. Isso é importante porque o servidor espera por dados do cliente e, se o cliente esperar por dados do servidor ao mesmo tempo, ambos os programas estarão esperando dados. Nessa situação, os programas ficarão esperando até que sejam encerrados. O Quadro 2 exibe um exemplo de execução do programa Cliente.

Quadro 2 - Acessando o servidor SimpleServer usando o cliente SimpleClient

```
1  >java SimpleClient localhost 9000
2  Enviar: teste
3  Recebido: (teste)
4  >
```

Fonte: Autoria própria.

7.2 SERVIDOR SIMPLES COM CONEXÃO PERMANENTE

O programa *TextServer* (Apêndice C) recebe linhas de texto e às envia de volta entre parênteses, da mesma forma que o programa *SimpleServer* diferente deste, não encerra a conexão após a primeira requisição.

Este programa demonstra como é possível continuar atendo requisições durante uma conexão e mesmo após a requisição ser encerrada pelo programa cliente, o servidor volta a aguardar conexões de outros clientes.

Um cliente conectado pode continuar enviando texto até que encerre sua conexão. O servidor encerra a conexão quando recebe o texto “sair” e volta a esperar por novas conexões. O Quadro 3 exibe um exemplo de comunicação com o servidor através de um cliente de *Telnet*.

Quadro 3 - Conectando ao servidor TextServer por Telnet

```
>telnet
telnet> set localecho
Local echo on
telnet>open localhost 9000
Connecting To localhost...
enviando uma linha de texto
(enviando uma linha de texto)
enviando uma outra de texto
(enviando uma linha de texto)
sair
Connection to host lost.
>
```

Fonte: Autoria própria.

O programa *TextClient* (Apêndice D), assim como o *SimpleClient*, recebe uma linha de texto do terminal e envia ao servidor mas não encerra sua execução e volta a receber novas linhas do teclado e enviar ao servidor até que o servidor encerre a conexão remotamente. O Quadro 4 exibe um exemplo de conexão utilizando o programa *TextClient*

Quadro 4 - Conectando ao servidor TextServer com o cliente TextClient

```
>java TextClient localhost 9000
Enviar: uma linha de texto
Recebido: (uma linha de texto)
Enviar: outra linha de texto
Recebido: (outra linha de texto)
Enviar: sair
>
```

Fonte: Autoria própria.

7.3 SERVIDOR SIMPLES DE ARQUIVOS

Um protocolo baseado em texto é facilmente exibido no terminal, mas não possibilita a transferência de dados binários. Para transferir arquivos é preciso utilizar os *streams* de entrada e saída de sockets em conjuntos com outras classes diferentes das *BufferedReader* e *PrintWriter*, utilizadas até agora.

Um servidor de arquivos pode ter a mesma estrutura base do servidor *SimpleServer* mas precisa ler e transferir dados de forma binária. Utilizando objetos da classe *BufferedOuputStream* é possível fazer a escrita de dados binários em *streams*.

O programa *SimpleFileServer* (Apêndice E) foi desenvolvido para realizar o envio simplificado de arquivos.

O *stream* de entrada do socket pode ser lido normalmente por um objeto do tipo *BufferedReader* porque o programa espera receber o nome do arquivo em formato de texto.

Neste programa também faz-se o uso de objetos do tipo *File* para verificar a existência do arquivo e *FileInputStream* para abrir um *stream* de leitura do arquivo. O *FileInputStream* foi utilizado em conjunto com *BufferedInputStream*, evidenciando que classes que trabalham com *stream* são intercambiáveis.

Um ponto importante a salientar sobre *BufferedInputStream* e *BufferedOuputStream* é que estas classes podem fazer a leitura de *streams* para *arrays* de bytes. A leitura de *streams* utilizando essas classes é mais eficiente porque podem ser lidos ou escritos vários bytes por vez num buffer ao invés de um byte de cada vez em classes de nível mais baixo.

Embora ainda muito simples este programa tem um protocolo mais elaborado que os programas apresentados anteriormente. Ao receber uma conexão, o servidor espera que o cliente envie uma linha de texto que contenha o nome de um arquivo.

O servidor verifica se o texto recebido corresponde a um arquivo e, em caso afirmativo, envia o arquivo ao cliente. Caso o arquivo não seja encontrado o servidor envia a mensagem "*Nome de arquivo invalido ou arquivo nao encontrado*"

Assim como o programa *SimpleServer*, o programa *SimpleFileServer* (Apêndice E) pode ser testado com outros clientes, como *Telnet*. É preferível que o arquivo solicitado seja do tipo texto porque, no caso de arquivos binários, os clientes de *Telnet* podem interpretar certos caracteres especiais como comandos e apresentar comportamento inesperado.

O Quadro 5 exibe o exemplo de saída quando o servidor é acessado por um cliente de *Telnet*. No exemplo é solicitado o arquivo de nome 'Arquivo.txt' (linha 6) e exibido seu conteúdo recebido do servidor (linhas 7, 8 e 9).

Quadro 5 - Acesso ao servidor SimpleFileServer através de um cliente de Telnet

```
>telnet
telnet> set localecho
Local echo on
telnet>open localhost 9000
Connecting To localhost...
Arquivo.txt
Linha 1
Linha 2
Linha 3
Connection to host lost.
>
```

Fonte: Autoria própria.

Ao ser acessado pelo *Firefox* o programa responde com "*Nome de arquivo invalido ou arquivo nao encontrado*" Isso acontece porque o Firefox envia o comando '*GET / HTTP/1.1*' na primeira linha da requisição e não um nome de arquivo, como esperado pelo servidor.

Quadro 6 - Modificação de SimpleFileServer para aceitar requisições do Firefox

```
filename = filename.replace("GET /", "").replace("HTTP/1.1", "");
```

Fonte: Autoria própria.

A alteração apresentada no Quadro 6, remove parte do comando *GET* da primeira linha da requisição deixando apenas o nome do recurso requisitado. Acessando o endereço '*http://localhost:9000*' continua-se o erro "*Nome de arquivo invalido ou arquivo nao encontrado*" porque não informamos um nome de um arquivo.

Informando um nome de arquivo na requisição como '*http://localhost:9000/Arquivo.txt*', visualiza-se o conteúdo do arquivo normalmente através do Firefox sem comprometer o funcionamento anterior do programa.

É importante salientar que o Firefox não tentará realizar o download do arquivo porque o servidor não responde com o protocolo HTTP que contém diversas informações, entre elas, o tipo do arquivo.

É possível salvar o arquivo utilizando o recurso do Firefox que permite salvar páginas Web.

Para acessar o programa servidor *SimpleFileServer* foi desenvolvido o cliente *SimpleFileCLient* (Apêndice F) usando como base o programa *SimpleCLient*. A principais diferenças entre *SimpleFileCLient* e *SimpleClient* está no modo como o *stream* de entrada é tratado. Como os servidores associados à esses programas

esperam receber texto, ambos os programas utilizam *PrintWriter* para escrever no *socket* de saída. Como o programa *SimpleFileCLient* espera receber dados binários como resposta, este utiliza no um objeto da classe *BufferedInputStream* para receber dados do *socket*, contrastando com o *BufferedOutputStream* que é utilizado no servidor para enviar dados binários ao cliente.

Ao contrário do programa servidor *SimpleFileServer* que utiliza *BufferedInputStream* para ler dados de arquivos, o programa *SimpleFileClient* utiliza *FileOutputStream* associado a *BufferedOuputStream* para escrever os dados recebidos em arquivos.

Ocorre um problema com este programa quando o servidor não encontra o arquivo solicitado. O programa Simple*FileClient* cria um arquivo contendo a mensagem de erro "*Nome de arquivo invalido ou arquivo nao encontrado*" enviada pelo servidor. Para o programa identificar o erro seria necessário interpretar os primeiro bytes recebidos e verificar se estes se referem à mensagem de erro ou não.

Para tratar de problemas como esse é mais interessante utilizar um protocolo mais elaborado, que contenha um cabeçalho com *status* de sucesso e erro para as respostas das requisições.

Cabeçalhos são muito importantes quando se trabalha com *streams* binários. Não é difícil identificar um caractere de final da linha em um texto, mas não é possível saber onde fica o final de um *stream* binário sem receber essa informação previamente.

Esse problema não ocorre quando não é necessária uma confirmação após o envio de dados binários. O programa *SimpleFileServer* envia todos o arquivo binário e fecha o *stream*. O programa *SimpleFileClient* lê os bytes recebidos até que o *stream* seja fechado.

Sem a informação do tamanho do arquivo no cabeçalho, não há como o servidor saber o quanto deve ler do *socket e* poderá ler do *stream* até que este seja fechado, impossibilitando o envio de uma resposta.

Informando o tamanho do arquivo no cabeçalho, pode-se saber a quantidades de bytes que precisam ser lidos do *stream* para gravar no arquivo, possibilitando continuar o fluxo do programa e enviar uma resposta ao programa cliente.

7.4 PROTOCOLO DE TRANSFERÊNCIA DE ARQUIVOS

Como o programa cliente e servidor fazem uso algoritmos semelhantes para ler e processar os dados dos *streams* optou-se por criar uma classe *FileProto* (Apêndice G) para uso comum entre os programas, contendo métodos para o processamento das mensagens. A classe *FileProto* foi desenvolvida para ser utilizada como auxiliar por programas servidores e clientes.

A classe *FileProto* recebe um *socket* e instancia objetos *BufferedInputStream* e *BufferedOuputStream* que serão associados aos *streams* do *socket*.

Foi definido um cabeçalho simples semelhante aos cabeçalhos do protocolo HTTP ‘variável: valor’. Cada variável é seguida de um caractere ‘*n*’ (nova linha) e o cabeçalho termina com o mesmo caractere. O final do cabeçalho pode ser identificado quando são encontrados dois caracteres de fim de linha simultâneos um marcando o fim da última variável e outro marcando o fim do cabeçalho. Logo após o cabeçalho espera-se dados binários, quando existentes.

Definiu-se que os cabeçalhos deveriam ter obrigatoriamente uma variável *type* que indica o tipo de requisição. Para requisitar um arquivo do servidor o cliente deve enviar um cabeçalho do tipo *get* contento uma variável *filename* com o nome do arquivo.

O servidor responde à requisição *get* com um cabeçalho do tipo *get-response* contendo variáveis *filename, filesize e status*; contendo o nome o tamanho e o estado da requisição como *success* em caso de sucesso.

Caso o servidor não encontrasse ou não tivesse permissão de acesso ao arquivo a reposta deveria conter a variável *status* com o valor *failure* e uma variável message com a descrição do erro.

Para não ser mais necessário informar o *host* todas as vezes que o programa cliente é executado, foi desenvolvido o método *readHeaderFromFile* para ler configurações de arquivos. Como os arquivos de configuração seguem o mesmo formato dos cabeçalhos, as variáveis pode ser extraídas utilizando o método *parseHeader*.

7.5 SERVIDOR DE ARQUIVOS UTILIZANDO PROTOCOLO

O programa *ProtoFileServer* (Apêndice H) faz uso da classe *ProtoFile* nas funcionalidades de ler *streams* de sockets e arquivos. A estrutura básica do programa *ProtoFileServer* é muito parecida com as dos servidores anteriores. A maior diferença está no fato das leituras e escritas nos sockets serem realizadas através de um objeto do tipo *FileProto*.

O programa *ProtoFileServer* extrai o nome do arquivo do cabeçalho da requisição e envia de volta uma resposta contendo uma variável *status* indicando sucesso ou falha, e em caso de sucesso o nome e tamanho do arquivo seguido do conteúdo binário do arquivo.

O programa *ProtoFileCLient* (Apêndice I) também faz uso da classe *FileProto*. Este programa lê as configurações de endereço de servidor e porta de um arquivo de nome *config* utilizando o método *readHeaderFromFile* (linha 18).

Assim como o programa *SimpleFileClient*, este programa envia uma requisição de arquivo para o servidor e grava o arquivo recebido no disco.

O programa *ProtoFileServer* não tem a funcionalidade de receber arquivos. Acrescentar novas funcionalidades para poder receber outros tipos de requisições deixaria o laço principal do programa muito extenso. Optou-se por criar novos métodos na classe *FileProto* para atender as requisições.

7.6 COMPARTILHAMENTO BÁSICO DE ARQUIVOS

Modificando o programa *ProtoFileServer* foi *desenvolvido* o programa *ShareServer* (Apêndice K). As alterações no programa *ProtoFileClient* deram origem ao método *hangleGet* da classe *FileProto* (Apêndice J).

O método *handleGet* adicionado à classe *FileProto* incorporou parte do código do programa *ProtoFileServer*, com poucas modificações. Esse método recebe um *HashMap* com as variáveis do cabeçalho e utiliza o conteúdo para criar o cabeçalho de resposta e enviar o conteúdo do arquivo requisitado.

O métodos *handlePut* recebe um *HashMap* com as variáveis do cabeçalho e utiliza o conteúdo para criar o cabeçalho de resposta e ler a quantidade informada de bytes e gravar em um arquivo em disco.

Este novo programa é capaz de atender dois tipos diferentes de requisição *get* e *put*. Sendo requisição *get* é utilizada para baixar arquivos do servidor e a requisição *put* é utilizada para enviar arquivos ao servidor.

O programa *FileSender* (Apêndice L) é um programa cliente desenvolvido para enviar arquivos para o servidor. Como nos programas anteriores o tratamento dos cabeçalhos e dados de arquivos são realizados pela classe *FileProto*.

O conjunto dos programas *FileServer*, *ProtoFileClient* e *ShareServer* permitem o compartilhamento básico de arquivos. Com esses programas é possível enviar arquivos para o servidor e outros clientes podem baixar os arquivos disponíveis.

O programa *ShareServer* é capaz de atender diversas requisições ao longo de sua execução mas só consegue atender uma requisição por vez.

Durante a transferência de arquivos grandes, outros clientes ficam aguardando a transferência terminar para que suas requisições sejam atendidas.

Isso acontece porque esse programa tem apenas a uma *thread*. Para que seja possível a conexão simultânea de vários clientes é preciso que o programa seja alterado de forma que as conexões de entrada sejam atendidas por *threads* diferentes.

7.7 COMPARTILHAMENTO DE ARQUIVOS MULTITHREAD

A classe *ShareServerThread* (Apêndice M) foi desenvolvida a partir do programa *ShareServer*, mas não é um programa. Essa classe estende a classe *Thread* fazendo com que várias instâncias possam ser executadas em paralelo. O construtor da classe *ShareServerThread* recebe um *Socket* como parâmetro. O método *run* recebeu parte do código do programa *ShareServer*.

O método *run* de uma classe *Thread* é executado quando a thread é iniciada e, geralmente, é chamado pela thread principal que a instanciou.

O programa *MultiShareServer* (Apêndice N) cria um *ServerSocket* e, a cada conexão recebida, instancia uma *thread* do tipo *ShareServerThread* para tratar a requisição. Este programa é muito próximo ao programa *ShareServer* tendo seu laço principal substituído pela criação de instâncias da classe *ShareServerThread*.

7.8 CHECAGEM DE INTEGRIDADE NA TRANSFERÊNCIA DE ARQUIVOS

Para adicionar o *Cyclic Redundancy Check* (CRC) ao cabeçalho foi criado o método *calcFileCRC* à classe *FileProto* (Apêndice O). Também foram feitas alterações nos método *handleGet* e *handlePut* assim como nos métodos que criam os cabeçalhos.

O envio do CRC é realizado em uma nova variável inserida no cabeçalho junto com o nome e o tamanho do arquivo.

Ambos, servidor e cliente, podem utilizar o método *calcFileCRC* para calcular o CRC do conteúdo recebido com e compara-lo com o valor do CRC recebido no cabeçalho da requisição.

7.9 DISTRIBUIÇÃO DE ARQUIVOS EM TEMPO REAL

Receber arquivos manualmente exige que o usuário execute o programa cliente de tempos em tempos para baixar o arquivo atualizado. Uma possiblidade seria colocar o programa cliente em um script que o executaria automaticamente.

Outra forma possível, seria manter um registro dos acessos dos usuários com informações sobre os arquivos já baixados mas, para isso, seria preciso um controle de contas de usuários que aumentaria muito a complexidade dos programas e ficaria além do escopo deste trabalho.

Uma solução possível, embora não ideal, é utilizar um recurso compartilhado entre as *threads ShareServerThread*.

A classe *BroadShareServerThread* (Apêndice P) e o programa *BroadMultiShareServer* (Apêndice Q), são versões de *ShareServerThread* e *MultiShareServer,* apresentados anteriormente, com modificações para enviar arquivos aos usuários conectados ao servidor.

O construtor da classe *BroadShareServerThread* passou a receber como parâmetro um objeto genérico do tipo *ArrayList<FileProto>* que é um vetor de objetos do tipo *FileProto*.

O programa *BroadMultiShareServer*, contém uma lista estática do tipo *ArrayList<FileProto>* e passa referência desta lista para as *threads* do tipo *BroadShareServerThread*, que instancia para o tratamento das requisições.

A *ArrayList<FileProto>* é utilizada para manter uma lista de objetos FileProto, que por sua vez, contém sockets e acessa seus *streams* para leitura e escrita.

Como a lista é compartilhada entre várias threads seu acesso precisa ser realizado dentro de um bloco *synchronized* para que não haja violação ou corrupção dos dados da lista.

Fazendo o uso desta lista de clientes conectados é possível enviar os arquivos recebidos para os demais clientes na lista. Para isso, foram realizadas alterações na classe *FileProto* (Apêndice R) principalmente no método *handlePut* que distribui o envio aos clientes da lista.

A classe *BroadShareServerThread* passou tratar requisições do tipo *subscribe* que deve ser enviada por um cliente que deseja receber uma cópia dos arquivos recebidos pelo servidor. Os programas clientes não foram modificados para esse fim. Um novo programa cliente foi desenvolvido com base no programa no programa *ProtoFileServer*.

Quando a *thread BroadShareServerThread* recebe uma requisição do tipo *subscribe* uma referência a essa conexão é inserida na lista compartilhada. Quando uma requisição *put* é recebida, o novo método *handlePut* salva o arquivo no disco e, em seguida, envia uma requisição *put* para cada cliente inscrito na lista.

O programa *FileReceiver* (Apêndice S) tem o comportamento muito parecido com o programa *ProtoFileServer* que foi convertido em um programa cliente. Este programa não envia requisições. Ao invés disso, fica aguardando requisições do tipo *put* que são tratadas pelo método *hanglePut* da classe *FileProto*.

É possível enviar arquivos normalmente para o servidor *BroadMultiShareServer* usando programa *FileSend*. Para receber publicações de arquivos de outros usuários é preciso estar executando o programa *FileReceiver*.

Estes três programas em conjunto permitem o compartilhamento de arquivos com publicação em tempo real para usuários que estejam executando o programa *FileReceiver*.

7.10 CANAL SEGURO DE COMUNICAÇÃO COM SSL

Embora bastante complexo em sua implementação, o uso de SSL em Java é bastante simples.

Para utilizar SSL na comunicação não foi necessária alteração significativa dos programas já desenvolvido. As únicas alterações necessárias nos programas foram a importação da biblioteca do objeto *SSLServerSocketFactory* (Quadro 7) e a substituição das classes que criam *Sockets* por chamadas de métodos classe *SSLServerSocketFactory*.

Quadro 7 - Importação da classe SSLServerSocketFactory

```
1  import javax.net.ssl.*;
```

Fonte: Autoria própria.

Para programas servidores, as linhas que instanciam objetos do tipo *ServerSocket* devem ser substituídas pelos códigos apresentados no Quadro 8.

Quadro 8 - Criando objeto ServerSocket com SSLServerSocketFactory

```
1  SSLServerSocketFactory ssf = (SSLServerSocketFactory)
     SSLServerSocketFactory.getDefault();
2  ServerSocket ss = ssf.createServerSocket(PORT);
```

Fonte: Autoria própria.

Para programas clientes, as linhas que instanciam objetos do tipo *Socket* devem ser substituídos pelos códigos apresentados no Quadro 9.

Quadro 9 - Criando objeto Socket com a classe SSLServerSocketFactory

```
1  SSLSocketFactory sf = (SSLSocketFactory)
     SSLSocketFactory.getDefault();
2  Socket s = sf.createSocket(hostname, port);
```

Fonte: Autoria própria.

Em relação ao código fonte dos programas essas são as únicas modificações necessárias.

Para que essas alterações funcionem de maneira correta é preciso criar uma *keyStore* com um certificado e passá-la por parâmetro para o Java na execução do programa.

O Kit de Desenvolvimento Java (*Java Development Kit* - JDK) disponibiliza a ferramenta *keytool* para a criação e certificados já inseridos em *keyStores*.

O Quadro 10, na linha 1 exibe o uso da ferramenta *keytool* para criar uma *keyStore* de nome *myKeyStore*.

Quadro 10 - Criando certificado e keystore para uso com para o servidor

```
01  >keytool -genkey -keystore myKeyStore –keyalg RSA
02  Enter keystore password:
03  Re-enter new password:
04  What is your first and last name?
05    [Unknown]:
06  What is the name of your organizational unit?
07    [Unknown]:
08  What is the name of your organization?
09    [Unknown]:
10  What is the name of your City or Locality?
11    [Unknown]:
12  What is the name of your State or Province?
13    [Unknown]:
14  What is the two-letter country code for this unit?
15    [Unknown]:
16  Is CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown,
      C=Unknown correct?
17    [no]:  y
18
19  Enter key password for <mykey> (RETURN if same as keystore password):
20  >
```

Fonte: Autoria própria.

É necessário informar e confirmar uma senha e responder a algumas perguntas conforme pode ser visto no Quadro 10.

Para utilizar a *keyStore* com os programa servidor é preciso passá-la por parâmetro para o Java conforme pode ser visto no Quadro 11, onde é passada a *keyStore* e a senha.

Quadro 11 - Executando BroadMultiShareServer com uma keystore

```
1  java -Djavax.net.ssl.keyStore=mySrvKeystore –
     Djavax.net.ssl.keyStorePassword=123456 BroadMultiShareServer
```

Fonte: Autoria própria.

O programa cliente utiliza uma *trustStore*, mas a diferença é apenas conceitual e pode-se utilizar o mesmo arquivo criado anteriormente utilizando a linha conforme o Quadro 12.

Quadro 12 - Executando FileServer com uma truststore

```
1  java -Djavax.net.ssl.trustStore=mySrvKeystore –
     Djavax.net.ssl.trustStorePassword=123456 FileSender
```

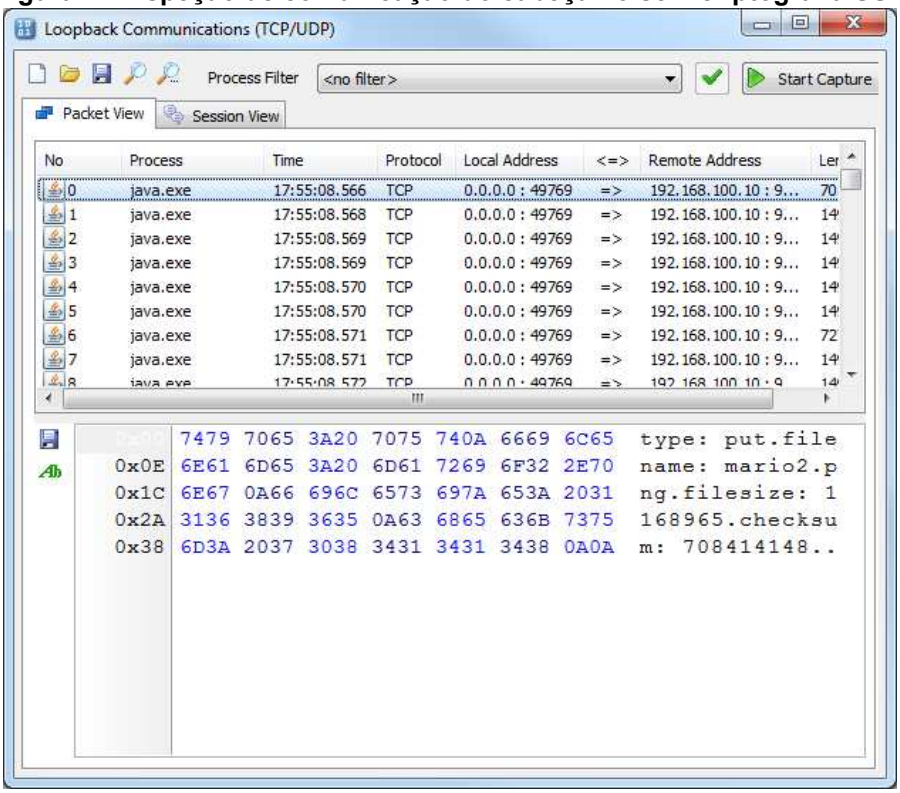
Fonte: Autoria própria.

É possível salvar essas comandos em *scripts* para utilizar chamadas mais simples no terminal. No caso dos programas cliente onde é necessário passar o nome do arquivo como parâmetro é preciso redirecionar parâmetros do *script* para os comandos, utilizando '*%n*' do terminal Windows e '*$1'* no terminal do Linux.

Após essas configurações os programas passarão a utilizar SSL na comunicação de *Sockets*.

A Figura 4, exibe a inspeção da comunicação sem criptografia SSL. É possível ver claramente o texto que compõe o cabeçalho de uma requisição do tipo *put.*

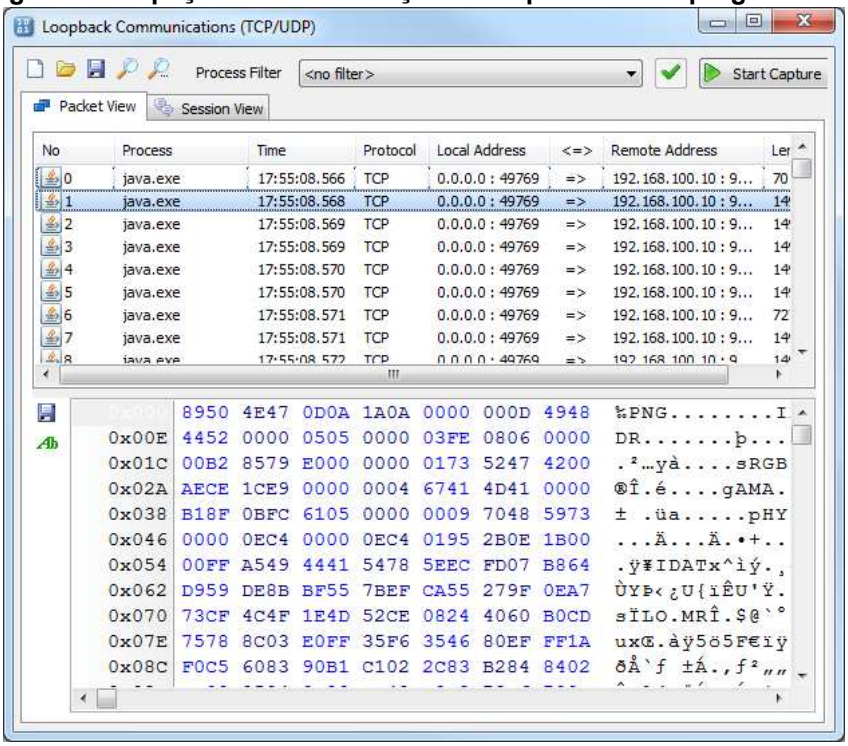
Figura 4 - Inspeção de comunicação do cabeçalho sem criptografia SSL



Fonte: Autoria própria.

A Figura 5, exibe a mesma inspeção da Figura 4, mostrando os bytes inicias do arquivo transferido. É possível var claramente que se trata de uma imagem do tipo *Portable Network Graphics* (PNG).

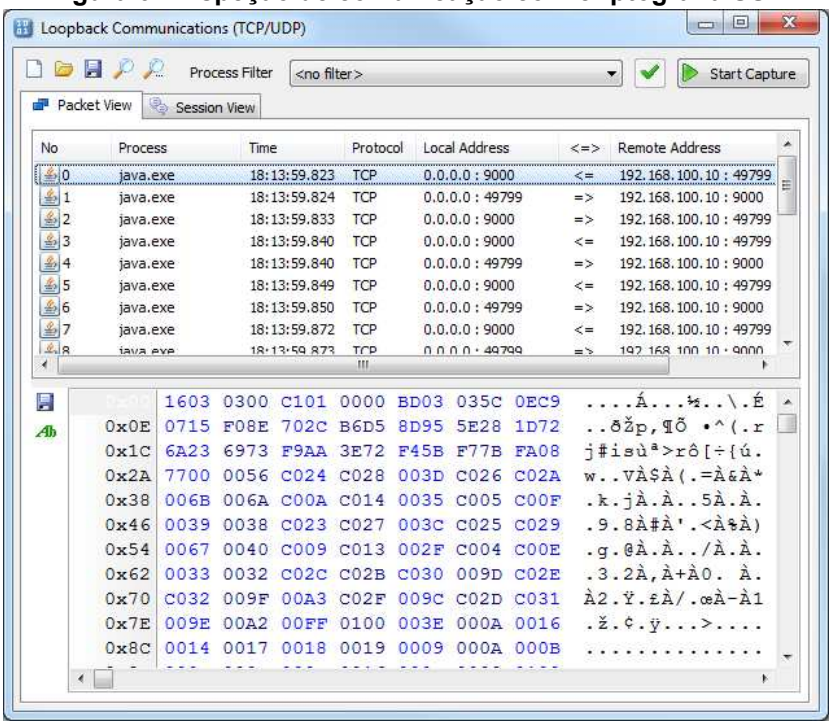
Figura 5 - Inspeção de comunicação do arquivo sem criptografia SSL



Fonte: Autoria própria.

A Figura 6, exibe a mesma inspeção da Figura 4, depois das alterações para uso de SSL. Neste caso, não há como identificar nem mesmo o cabeçalho da requisição.

Figura 6 - Inspeção de comunicação com criptografia SSL



Fonte: Autoria própria.

8 RESULTADO E DISCUSSÕES

Os programas desenvolvidos neste trabalho são bastante simples em suas funcionalidades, mas exemplificam as possibilidades disponibilizadas pelas bibliotecas de comunicação via *socket* da Linguagem Java.

Muitas outras funcionalidades podem ser implementadas tornar os programas mais eficazes. Esses programas transferem penas arquivos, mas transferir pastas de arquivos é uma ficcionalidade bastante prática.

Para transferir diretórios novos tipos de requisições devem ser implementadas, bem como o tratamento adequado para estas. É preciso desenvolver um recurso de varredura ou busca de diretórios como ‘profundidade primeiro’.

Listar os arquivos disponíveis no servidor também é um recurso bastante prático. Sem uma listagem dos arquivos não há como um usuário utilizar o nome dos arquivos disponíveis para baixar.

O programa *TextClient* pode ser modificado para ser utilizado como um navegador de diretórios utilizando comandos parecidos com comandos do Linux ou Windows como ‘*cd’*, ‘*ls*’ e ‘*dir*’ possibilitando a navegação em pastas disponíveis no servidor. E o comando ‘*get*’ poderia ser um comando do programa cliente para baixar o arquivo escolhido.

Quando utilizam-se *threads* e são realizados muitos acessos concorrentes em arquivos e diretórios há uma grande possibilidade de ocorrência de erros por que arquivos podem estar sendo utilizados por outras threads. Implementar recursos de *Lock* de extrema importância para garantir que os arquivos não serão corrompidos enquanto estão sendo acessados.

Há pouca validação nos protocolos nos programas apresentados. A validação de protocolos é importante para o caso de ser realizada alguma conexão de forma incorreta por programas que utilizam protocolos diferentes. Sem a validação adequada, o comportamento do programa pode se mostras instável quando acessado por outros protocolos.

Nenhum dos programas apresentados tenta reenviar a requisição em caso de falha. Quando o arquivo está sendo baixado por um programa operado por um usuário isso não chega a ser um problema, mas no caso do programa *FileReceiver* nenhuma

tentativa de retransmissão é efetuada, o que forçaria o usuário do programa a baixar o arquivo manualmente.

Um recurso interessante é poder transferir arquivos divididos em partes. Quando arquivos são transferidos inteiros sem interrupção e ocorre algum erro durante a transferência, o arquivo precisa ser retransmitido. Se ocorrer erro na transferência de um arquivo muito grande, o precisa arquivo ser transmitido inteiro novamente. As partes dos arquivos podem ser acompanhadas de CRCs que podem, inclusive serem armazenados como um *cache* de CRCs, para uso posterior, sem que seja necessário recalcula-los.

Outra vantagem de transferir partes de um arquivo é a possibilidade de se exibir uma barra de progresso da transferência a cada parte transferida que, inclusive pode exibir a quantidade de bytes ainda pendentes na transferência.

A transmissão de partes do arquivos leva ainda a outra possibilidade a verificação da diferença entre os arquivos do servidor e do cliente. A requisição de um arquivo pode já existente no cliente pose ser enviada com o seu CRC e comparado com o CRC do arquivo no servidor e caso os valores coincidam o arquivo nem mesmo precisa ser baixado para o cliente. Caso os valores não sejam coincidentes pode se comparar partes do arquivo com seus respectivos CRCs e somente as partes divergentes precisariam ser transferidas.

Um recursos muito importante em sistemas de compartilhamento de arquivos é manter um *log* de acessos aos arquivos, para saber quem fez atualizações nos arquivos do servidor. Para isso preciso um elaborar sistema de contas de usuário e permissões e permitir que usuários e cadastrem no sistema.

Os programas apresentados utilizam a mesma pasta para manter arquivos de todos os usuários conectados. Fazendo uso de um sistema de contas de usuários que usuários tenham pastas exclusivas, que possam compartilhas seus arquivos com outro usuários e grupos e ainda, que mantenham uma lista de seus arquivos compartilhados e usuários com quem compartilha.

Com um sistema simples de contas de usuários é possível implementar uma forma simples de manter o usuário atualizado. Identificando o usuário por meio de nome e senha, pode-se manter um registro com a data de seu último acesso ou último arquivo recebido. Então pode-se enviar ao usuário todos os arquivos com data

posterior. Assim o usuário poderia receber atualizações que ocorreram enquanto não estava conectado.

Manter as versões anteriores dos arquivos pode ser bastante útil em muitos casos. Em caso de perda ou alteração acidental seria possível reverter para uma versão anterior. Para não ocupar muito espaço com várias cópias de um arquivo poderia cópias diferenciais (*diffs*).

A possibilidade de acessar os arquivos através de uma interface pode ser a única opção quando se tem disponível o computador com o programa correto instalado e configurado.

Através de uma interface Web o usuário poderia acessar o sistema de compartilhamento de qualquer computador ou smartphone, sem precisar instalar programas ou aplicativos. E o gerenciamento de arquivos e usuários seria muito mais fácil e intuitiva.

9 CONCLUSÃO

O uso de *Sockets* em Java não é trivial mas, com um pouco de prática, até mesmo estudantes iniciantes da linguagem podem criar programas que se comuniquem com servidores ou até mesmo programas servidores que usem texto como base para comunicação.

O desenvolvimento de servidores é desafiador mas a linguagem Java disponibiliza classes que facilitam o trabalho.

Ao longo deste trabalho encontrou-se mais dificuldade em elaborar um protocolo de comunicação do que estabelecer a comunicação entre cliente e servidor.

Comparando a funcionalidade com os códigos dos programas. É possível perceber que a complexidade aumenta ainda mais quando surge a necessidade de transferir dados binários sem interromper a comunicação entre cliente e servidor.

Para desenvolver sistemas mais complexos se faz necessária muito planejamento, disciplina e boas práticas de programação.

REFERÊNCIAS

KUROSE, James F.; KEITH, W. Ross. **Redes de computadores e a internet**. 6. ed., São Paulo: Pearson Education do Brasil, 2013.

SANTOS, Rafael. **Introdução à programação orientada a objetos usando Java**. 2. ed., São Paulo: Campus, 2013.

SILBERCHATZ, Abraham; et al. **Fundamentos de sistemas operacionais**. 9. ed., Rio de Janeiro: LTC, 2015.

TANENBAUM, Andrew S.; WETHERALL, David J. **Redes de computadores**. 5. ed., São Paulo: Pearson, 2011.

TANENBAUM, Andrew S.; BOS, Herbert. **Sistemas operacionais modernos**. 4. ed., São Paulo: Pearson, 2016.

APÊNDICES

APÊNDICE A: CLASSE “SIMPLESERVER”

```
import java.io.*;
import java.net.*;

public class SimpleServer{

  public static void main(String[] args){
    int port = 9000;
    if (args.length > 0) port = Integer.parseInt(args[0]);

    try{
      ServerSocket serverSocket = new ServerSocket(port);
      System.out.println("Servidor iniciado na porta " + port);

      System.out.println("Aguardando conexoes");
      Socket socket = serverSocket.accept();

      String remoteAddress = socket.getRemoteSocketAddress().toString();
      System.out.println("Cliente conectado: " + remoteAddress);

      InputStream input = socket.getInputStream();
      InputStreamReader streamReader = new InputStreamReader(input);
      BufferedReader reader = new BufferedReader(streamReader);

      OutputStream output = socket.getOutputStream();
      PrintWriter writer = new PrintWriter(output, true);

      String received = reader.readLine();

      System.out.println("Recebido: " + received);
      writer.println("("+ received + ")");

      writer.close();
    }catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

APÊNDICE B: CLASSE “SIMPLECLIENT”

```
import java.io.*;
import java.net.*;

public class SimpleClient{

  public static void main(String[] args){
    int port = 9000;

    if (args.length < 1){
      System.out.println("java SimpleClient hostname [port]");
      System.exit(0);
    }

    if (args.length == 2){
      port = Integer.parseInt(args[1]);
    }

    String hostname = args[0];

    try{
      Socket socket = new Socket(hostname, port);

      InputStream input = socket.getInputStream();
      InputStreamReader streamReader = new InputStreamReader(input);
      BufferedReader reader = new BufferedReader(streamReader);

      OutputStream output = socket.getOutputStream();
      PrintWriter writer = new PrintWriter(output, true);

      InputStreamReader terminalReader = new InputStreamReader(System.in);
      BufferedReader terminalIn = new BufferedReader(terminalReader);

      System.out.print("Enviar: ");
      String textToSend = terminalIn.readLine();
      writer.println(textToSend);

      String received = reader.readLine();
      System.out.println("Recebido: " + received);

      writer.close();
    }catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

APÊNDICE C: CLASSE “TEXTSERVER”

```
import java.io.*;
import java.net.*;

public class TextServer{

  public static void main(String[] args){
    int port = 9000;
    if (args.length > 0) port = Integer.parseInt(args[0]);

    try{
      ServerSocket serverSocket = new ServerSocket(port);
      System.out.println("Servidor iniciado na porta " + port);

      while(true){
        System.out.println("Aguardando conexoes");
        Socket socket = serverSocket.accept();

        String remoteAddress = socket.getRemoteSocketAddress().toString();
        System.out.println("Cliente conectado: " + remoteAddress);

        InputStream input = socket.getInputStream();
        InputStreamReader streamReader = new InputStreamReader(input);
        BufferedReader reader = new BufferedReader(streamReader);

        OutputStream output = socket.getOutputStream();
        PrintWriter writer = new PrintWriter(output, true);

        try{
          while(true){
            String received = reader.readLine();

            if(received.equals("sair")){
              System.out.println(remoteAddress + " desconectado");
              socket.close();
              break;
            }

            System.out.println("Recebido: " + received);
            writer.println("("+ received + ")");
          }
        }catch(Exception e){
          System.out.println("Conexão com " + remoteAddress + " perdida");
        }
      }
    }catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

APÊNDICE D: CLASSE “TEXTCLIENT”

```
import java.io.*;
import java.net.*;

public class TextClient{

  public static void main(String[] args){
    int port = 9000;

    if (args.length < 1){
      System.out.println("java TextClient hostname [port]");
      System.exit(0);
    }

      if (args.length == 2){
        port = Integer.parseInt(args[1]);
      }

      String hostname = args[0];

      try{
        Socket socket = new Socket(hostname, port);

        InputStream input = socket.getInputStream();
        InputStreamReader streamReader = new InputStreamReader(input);
        BufferedReader reader = new BufferedReader(streamReader);

        OutputStream output = socket.getOutputStream();
        PrintWriter writer = new PrintWriter(output, true);

        InputStreamReader terminalReader = new
InputStreamReader(System.in);
        BufferedReader terminalIn = new BufferedReader(terminalReader);

        while(true){
          System.out.print("Enviar: ");
          String textToSend = terminalIn.readLine();
          if(textToSend == null) break;
          if(textToSend.equals("")) continue;

          writer.println(textToSend);

          if(textToSend.equals("sair")){
            socket.close();
            break;
          }

          String received = reader.readLine();
          System.out.println("Recebido: " + received);
        }
    }catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

APÊNDICE E: CLASSE “SIMPLEFILESERVER”

```
import java.io.*;
import java.net.*;

public class SimpleFileServer{

  public static void main(String[] args){
    int port = 9000;
    if (args.length > 0) port = Integer.parseInt(args[0]);
    try{
      ServerSocket serverSocket = new ServerSocket(port);
      System.out.println("Servidor iniciado na porta " + port);
      while(true){
        System.out.println("Aguardando conexoes");
        Socket socket = serverSocket.accept();

        String remoteAddress = socket.getRemoteSocketAddress().toString();
        System.out.println("Cliente conectado: " + remoteAddress);

        InputStream input = socket.getInputStream();
        InputStreamReader streamReader = new InputStreamReader(input);
        BufferedReader reader = new BufferedReader(streamReader);

        OutputStream output = socket.getOutputStream();
        BufferedOutputStream out = new BufferedOutputStream(output);
        try{
          String filename = reader.readLine();
          filename = filename.replace("GET /", "").replace("HTTP/1.1", "");
// FIREFOX
          File file = new File(filename);
          if (file.isFile()){
            System.out.println("Enviando arquivo: " + filename);
            FileInputStream fileInput = new FileInputStream(file);
            BufferedInputStream fin = new BufferedInputStream(fileInput);

            int bytesRead = 0;
            byte [] buffer = new byte[1024];
            while((bytesRead = fin.read(buffer)) > -1){
              out.write(buffer, 0, bytesRead);
            }
            fin.close();
          }else{
            System.out.println(filename.trim() + " nao e um arquivo");
            out.write(Nome de arquivo invalido ou arquivo nao
encontrado".getBytes());
          }
          out.close();
        }catch(Exception e){
          System.out.println("Conexão com " + remoteAddress + " perdida");
          e.printStackTrace();
        }
      }
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

APÊNDICE F: CLASSE “SIMPLEFILECLIENT”

```
import java.io.*;
import java.net.*;

public class SimpleFileClient{

  public static void main(String[] args){
    int port = 9000;
    if (args.length < 1){
      System.out.println("java SimpleFileClient hostname [port]");
      System.exit(0);
    }

    if (args.length == 2){
      port = Integer.parseInt(args[1]);
    }

    String hostname = args[0];

    try{
      Socket socket = new Socket(hostname, port);

      InputStream input = socket.getInputStream();
      BufferedInputStream in = new BufferedInputStream(input);

      OutputStream output = socket.getOutputStream();
      PrintWriter writer = new PrintWriter(output, true);

      InputStreamReader terminalReader = new InputStreamReader(System.in);
      BufferedReader terminalIn = new BufferedReader(terminalReader);

      System.out.print("Arquivo: ");
      String filename = terminalIn.readLine();
      writer.println(filename);

      File file = new File(filename);
      if(file.isFile())
      {
        FileOutputStream fileOutput = new FileOutputStream(filename);
        BufferedOutputStream fout = new BufferedOutputStream(fileOutput);

        int bytesRead = 0;
        byte [] buffer = new byte[1024];
        while((bytesRead = in.read(buffer)) > 0){
          fout.write(buffer, 0, bytesRead);
        }
        fout.close();
      }
      else
      {
        System.out.print("Arquivo invalido!");
      }
      writer.close();
    }catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

APÊNDICE G: CLASSE “FILEPROTO”

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.zip.CRC32;
import java.util.zip.CheckedInputStream;

public class FileProto{

  private String repos = "";
  private Socket socket;
  private BufferedInputStream in;
  private BufferedOutputStream out;

  public FileProto(Socket socket) throws Exception{
    configStreams(socket);
  }

  public FileProto(Socket socket, String repos) throws Exception{
    this.repos = repos + "/";
    configStreams(socket);
  }

  private void configStreams(Socket socket) throws Exception{
    this.socket = socket;
    InputStream input = socket.getInputStream();
    this.in = new BufferedInputStream(input);
    OutputStream output = socket.getOutputStream();
    this.out = new BufferedOutputStream(output);
  }

  public String getRemoteAddress() throws Exception{
    return socket.getRemoteSocketAddress().toString();
  }

  public void close() throws Exception{
    out.close();
  }

  public static HashMap <String, String> parseHeader(String temp){
    HashMap <String, String> header = new HashMap<String, String>();

    String [] lines = temp.split("\n");
    for(String line : lines)
    {
      String [] items = line.split(":");
      if (items.length < 2) continue;
      String key = items[0].trim();
      String value = items[1].trim();
      header.put(key, value);
    }
    return header;
  }
```

```
public static String readFromFile(String filename) throws Exception{
    String config = "";
    BufferedReader reader = new BufferedReader(new FileReader(filename));
    String line = "";
    while((line = reader.readLine()) != null){
      config += line + "\n";
    }
    reader.close();
    return config;
  }

  public static String makeGet(String filename) throws Exception{
    String header = "";
    header += "type: get\n";
    header += "filename: " + filename + "\n";
    header += "\n";
    return header;
  }


public static String makeGetResponse(String filename, long filesize) throws
Exception{
    String header = "";
    header += "type: get-response\n";
    header += "filename: " + filename + "\n";
    header += "filesize: " + String.valueOf(filesize) + "\n";
    header += "status: success\n";
    header += "\n";
    return header;
  }

  public static String makeGetResponse(String message) throws Exception{
    String header = "";
    header += "type: get-response\n";
    header += "status: failure\n";
    header += "status: " + message + "\n";
    header += "\n";
    return header;
  }

  public String receiveHeader() throws Exception{
    String header = "";
    int n = 0;
    while((n = in.read()) > -1){
      header += (char)n;
      if((char)n == '\n'){
        if((n = in.read()) > -1){
          header += (char)n;
          if((char)n == '\n'){
            break;
          }
        }
      }
    }
    return header.trim();
  }
```

```
public void sendHeader(String header) throws Exception{
    out.write(header.getBytes());
    out.flush();
  }

public void receiveFile(String filename, String filesize) throws Exception{
    File repository = new File(this.repos);
    repository.mkdir();

    FileOutputStream fileOutput = new FileOutputStream(this.repos +
filename);
    BufferedOutputStream fout = new BufferedOutputStream(fileOutput);

    int bytesRead = 0;
    long bytesReadTotal = 0;
    byte [] buffer = new byte[1024];
    while((bytesRead = in.read(buffer)) > -1){
      fout.write(buffer, 0, bytesRead);
      bytesReadTotal += bytesRead;
      if(bytesReadTotal == Long.parseLong(filesize)) break;
    }
    fout.close();
  }

  public void sendFile(String filename) throws Exception{
    File file = new File(filename);
    FileInputStream fileInput = new FileInputStream(file);
    BufferedInputStream fin = new BufferedInputStream(fileInput);
    int bytesRead = 0;
    byte [] buffer = new byte[1024];
    while((bytesRead = fin.read(buffer)) > -1){
      out.write(buffer, 0, bytesRead);
    }
    fin.close();
    out.flush();
  }
}
```

APÊNDICE H: CLASSE “PROTOFILESERVER”

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ProtoFileServer{

  public static void main(String[] args){
    int port = 9000;
    if (args.length > 0) port = Integer.parseInt(args[0]);

    try{
      ServerSocket serverSocket = new ServerSocket(port);
      System.out.println("Servidor iniciado na porta " + port);

      while(true){
        System.out.println("Aguardando");
        Socket socket = serverSocket.accept();
        FileProto proto = new FileProto(socket);

        String remoteAddress = proto.getRemoteAddress();
        System.out.println("Cliente conectado: " + remoteAddress);

        try{
          String header = proto.receiveHeader();
          if(header.equals("")) break;
          System.out.println(header);

          HashMap<String, String> headerMap = proto.parseHeader(header);

          String filename = headerMap.get("filename");
          File file = new File(filename);
          if(file.isFile()){
            String responseHeader  = FileProto.makeGetResponse(filename,
file.length());
            System.out.println("Enviando: '" + filename + "'");
            proto.sendHeader(responseHeader);
            proto.sendFile(filename);
            System.out.println("(Concluido!)");
          }else{
            String message = "Arquivo nao encontrado";
            String responseHeader  = FileProto.makeGetResponse(message);
            proto.sendHeader(responseHeader);
          }
        }catch(Exception e){
          System.out.println("Conexão com " + remoteAddress + " perdida");
        }
        proto.close();
      }
    }catch (Exception ex){
    ex.printStackTrace();
    }
  }
}
```

APÊNDICE I: CLASSE “PROTOFILECLIENT”

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ProtoFileClient{
  public static void main(String[] args) throws Exception{
    String hostname = "localhost";
    int port = 9000;
    if(args.length < 1){
      System.out.println("java ProtoFileCLient filename");
      System.exit(0);
    }
    String filename = args[0];
    try{
      String config = FileProto.readFromFile("config");
      HashMap<String, String> configMap = FileProto.parseHeader(config);
      if(configMap.containsKey("host")) hostname = configMap.get("host");
      if(configMap.containsKey("port")){
        port = Integer.parseInt(configMap.get("port"));
      }
    }catch (Exception ex){
      System.out.print("Verifique o arquivo 'config'\nhost: {ip}\nport:
{porta}");
      System.exit(0);
    }
    try{
      Socket socket = new Socket(hostname, port);
      FileProto proto = new FileProto(socket);
      String header = FileProto.makeGet(filename);
      proto.sendHeader(header);
      String responseHeader = proto.receiveHeader();
      System.out.println(responseHeader);
      HashMap<String, String> headerMap =
proto.parseHeader(responseHeader);
      String status = headerMap.get("status");
      String filesize = headerMap.get("filesize");
      if(status.equals("success")){
        System.out.println("Recebendo: '" + filename + "'");
        proto.receiveFile(filename, filesize);
        File file = new File(filename);
        if(file.isFile()){
          if(file.length() == Long.parseLong(filesize)){
            System.out.println("Arquivo recebido");
          }else{
            System.out.println("Arquivo recebido CORROMPIDO");
          }
        }else{
          System.out.println("Não foi possível salvar o arquivo");
        }
      }
      proto.close();
    }catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

APÊNDICE J: ATUALIZAÇÃO DE MÉTODOS DA CLASSE “FILEPROTO”

```
  public static String makeGet(String filename) throws Exception{
    String header = "";
    header += "type: get\n";
    header += "filename: " + filename + "\n";
    header += "\n";
    return header;
  }

  public static String makeGetResponse(String filename, long filesize)
throws Exception{
    String header = "";
    header += "type: get-response\n";
    header += "filename: " + filename + "\n";
    header += "filesize: " + String.valueOf(filesize) + "\n";
    header += "status: success\n";
    header += "\n";
    return header;
  }

  public static String makeGetResponse(String message) throws Exception{
    String header = "";
    header += "type: get-response\n";
    header += "status: failure\n";
    header += "status: " + message + "\n";
    header += "\n";
    return header;
  }

  public void handleGet(HashMap<String, String> headerMap) throws
Exception{
    String filename = headerMap.get("filename");
    File file = new File(this.repos + filename);

    if(file.isFile()){
      String responseHeader  = this.makeGetResponse(filename,
file.length());

      System.out.println("Enviando: '" + filename + "'");
      this.sendHeader(responseHeader);
      this.sendFile(this.repos + filename);
      System.out.println(" (Concluido!)");

    }else{
      String message = "Arquivo nao encontrado";
      String responseHeader  = this.makeGetResponse(message);
      this.sendHeader(responseHeader);
    }
  }
```

```
public void handlePut(HashMap<String, String> headerMap) throws Exception{
    String filename = headerMap.get("filename");
    String filesize = headerMap.get("filesize");

    System.out.println("Recebendo: '" + filename + "'");
    this.receiveFile(filename, filesize);
    System.out.println(" (Concluido!)");

    String responseHeader = "";
    File file = new File(this.repos + filename);
    if(file.isFile()){
        long fileCrc = this.calcFileCRC(filename);
        if(file.length() == Long.parseLong(filesize)){
        responseHeader  = this.makePutResponse(true);
      }else{
        String message = "Nao foi possivel salvar o arquivo";
        responseHeader  = this.makePutResponse(false, message);
      }
    }else{
      String message = "Arquivo recebido CORROMPIDO";
      responseHeader  = this.makePutResponse(false, message);
    }
    this.sendHeader(responseHeader);
  }
```

APÊNDICE K: CLASSE “SHARESERVER”

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ShareServer{

  public static void main(String[] args){
    int port = 9000;
    if (args.length > 0) port = Integer.parseInt(args[0]);

    try{
      ServerSocket serverSocket = new ServerSocket(port);
      System.out.println("Servidor iniciado na porta " + port);

      while(true){
        System.out.println("Aguardando");
        Socket socket = serverSocket.accept();
        FileProto proto = new FileProto(socket, "repositorio");

        String remoteAddress = proto.getRemoteAddress();
        System.out.println("Cliente conectado: " + remoteAddress);

        try{
          String header = proto.receiveHeader();
          if(header.equals("")) break;

          HashMap<String, String> headerMap = proto.parseHeader(header);
          String type = headerMap.get("type");

          System.out.println(header);
          if(type.equals("get")) proto.handleGet(headerMap);
          if(type.equals("put")) proto.handlePut(headerMap);

        }catch(Exception e){
          e.printStackTrace();
          System.out.println("Conexão com " + remoteAddress + " perdida");
        }
      }
    }catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

APÊNDICE L: CLASSE “FILESENDER”

```
import java.io.*;
import java.net.*;
import java.util.*;

public class FileSender{

    public static void main(String[] args) throws Exception{
    String hostname = "localhost";
    int port = 9000;
    if(args.length < 1){
      System.out.println("java FileSender filename");
      System.exit(0);
    }
    String filename = args[0];
    try{
      String config = FileProto.readFromFile("config");
      HashMap<String, String> configMap = FileProto.parseHeader(config);
      if(configMap.containsKey("host")) hostname = configMap.get("host");
      if(configMap.containsKey("port")){
        port = Integer.parseInt(configMap.get("port"));
      }
        }catch (Exception ex){
      System.out.print("Verifique o arquivo 'config'\nhost: {ip}\nport:
{porta}");
      System.exit(0);
    }
    try{
      Socket socket = new Socket(hostname, port);
      FileProto proto = new FileProto(socket);
      File file = new File(filename);
      if(file.isFile()){
        String header = proto.makePut(filename, file.length());
        System.out.print("Enviando: '" + filename + "'");
        proto.sendHeader(header);
        proto.sendFile(filename);
        System.out.println(" (Concluido!)");
        String responseHeader = proto.receiveHeader();
        HashMap<String, String> headerMap =
proto.parseHeader(responseHeader);
        String status = headerMap.get("status");
        String message = headerMap.get("message")
        if(status.equals("success")){
          System.out.println("Arquivo recebido com sucesso pelo servidor");
        }
        else{
          System.out.println(message);
        }
      }else{
        System.out.print("Arquivo invalido!");
      }
      proto.close();
        }catch (Exception ex){
      ex.printStackTrace();
        }
    }
}
```

APÊNDICE M: CLASSE “SHARESERVERTHREAD”

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ShareServerThread extends Thread{

  private Socket socket;
  private FileProto proto;

  public ShareServerThread(Socket socket){
    this.socket = socket;
  }

  public void run(){
    try{
      proto = new FileProto(socket, "repositorio");
      String remoteAddress = proto.getRemoteAddress();
      try{
        while(true){
          String header = proto.receiveHeader();
          if(header.equals("")) break;

          HashMap<String, String> headerMap = proto.parseHeader(header);
          String type = headerMap.get("type");

          System.out.println(header);
          if(type.equals("get")) proto.handleGet(headerMap);
          if(type.equals("put")) proto.handlePut(headerMap);
        }
      }catch (Exception e){
        System.out.println("Conexão com " + remoteAddress + " perdida");
      }
    }catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

APÊNDICE N: CLASSE “MULTISHARESERVER”

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ShareServerThread extends Thread{

  private Socket socket;
  private FileProto proto;

  public ShareServerThread(Socket socket){
    this.socket = socket;
  }

  public void run(){
    try{
      proto = new FileProto(socket, "repositorio");
      String remoteAddress = proto.getRemoteAddress();
      try{
        while(true){
          String header = proto.receiveHeader();
          if(header.equals("")) break;

          HashMap<String, String> headerMap = proto.parseHeader(header);
          String type = headerMap.get("type");

          System.out.println(header);
          if(type.equals("get")) proto.handleGet(headerMap);
          if(type.equals("put")) proto.handlePut(headerMap);
        }
      }catch (Exception e){
        System.out.println("Conexão com " + remoteAddress + " perdida");
      }
    }catch (Exception ex){
      ex.printStackTrace();
        }
  }
}
```

APÊNDICE O: ATUALIZAÇÃO DE MÉTODOS DA CLASSE “FILEPROTO” PARA CÁLCULO DE CRC

```
  public static String makeGetResponse(String filename, long filesize, long
crc) throws Exception{
    String header = "";
    header += "type: get-response\n";
    header += "filename: " + filename + "\n";
    header += "filesize: " + String.valueOf(filesize) + "\n";
    header += "cheksum: " + String.valueOf(crc) + "\n";
    header += "status: success\n";
    header += "\n";
    return header;
  }

  public static String makePut(String filename, long filesize, long crc)
throws Exception{
    String header = "";
    header += "type: put\n";
    header += "filename: " + filename + "\n";
    header += "filesize: " + String.valueOf(filesize) + "\n";
    header += "checksum: " + String.valueOf(crc) + "\n";
    header += "\n";
    return header;
  }

  public static long calcFileCRC(String filename) throws Exception{
    CheckedInputStream cis = null;
    cis = new CheckedInputStream(new FileInputStream(filename), new
CRC32());
    byte[] buf = new byte[128];
    while(cis.read(buf) >= 0){}
    return cis.getChecksum().getValue();
  }

  public void handleGet(HashMap<String, String> headerMap) throws
Exception{
    String filename = headerMap.get("filename");
    File file = new File(this.repos + filename);
    long fileCrc = this.calcFileCRC(filename);
    if(file.isFile()){
      String responseHeader  = this.makeGetResponse(filename,
file.length(), fileCrc);
      System.out.println("Enviando: '" + filename + "'");
      this.sendHeader(responseHeader);
      this.sendFile(this.repos + filename);
      System.out.println(" (Concluido!)");
    }else{
      String message = "Arquivo nao encontrado";
      String responseHeader  = this.makeGetResponse(message);
      this.sendHeader(responseHeader);
    }
  }
```

```
public void handlePut(HashMap<String, String> headerMap) throws Exception{
    String filename = headerMap.get("filename");
    String filesize = headerMap.get("filesize");
    String crc = headerMap.get("checksum");
    System.out.println("Recebendo: '" + filename + "'");
    this.receiveFile(filename, filesize);
    System.out.println(" (Concluido!)");
    String responseHeader = "";
    File file = new File(this.repos + filename);
    if(file.isFile()){
        long fileCrc = this.calcFileCRC(filename);
        if(file.length() == Long.parseLong(filesize) && fileCrc ==
Long.parseLong(crc)){
        responseHeader  = this.makePutResponse(true);
      }else{
        String message = "Nao foi possivel salvar o arquivo";
        responseHeader  = this.makePutResponse(false, message);
      }
    }else{
      String message = "Arquivo recebido CORROMPIDO";
      responseHeader  = this.makePutResponse(false, message);
    }
    this.sendHeader(responseHeader);
  }
```

APÊNDICE P: CLASSE “BROADSHARESERVERTHREAD”

```
import java.io.*;
import java.net.*;
import java.util.*;

public class BroadShareServerThread extends Thread{

  private Socket socket;
  private FileProto proto;
  ArrayList<FileProto> clientList;

  public BroadShareServerThread(Socket socket, ArrayList<FileProto>
clientList){
    this.socket = socket;
    this.clientList = clientList;
  }

  public void run(){
    try{
      proto = new FileProto(socket, "repositorio");
      String remoteAddress = proto.getRemoteAddress();
      try{
        while(true){

          String header = "";
          synchronized(clientList){
            if(!clientList.contains(proto)) header = proto.receiveHeader();
          }
          if(header.equals("")) break;

          HashMap<String, String> headerMap = proto.parseHeader(header);
          String type = headerMap.get("type");

          System.out.println(header);
          if(type.equals("get")) proto.handleGet(headerMap);
          if(type.equals("put")) proto.handlePut(headerMap, clientList);
          if(type.equals("subscribe")){
            synchronized(clientList){
              clientList.add(proto);
            }
          }
        }
      }catch (Exception e){
        System.out.println("Conexão com " + remoteAddress + " perdida");
      }
    }catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

APÊNDICE Q: CLASSE “BROADMULTISHARESERVER”

```
import java.io.*;
import java.net.*;
import java.util.*;

public class BroadMultiShareServer{

  static ArrayList<FileProto> clientList = new ArrayList<FileProto>();

  public static void main(String[] args){
    int port = 9000;
    if (args.length > 0) port = Integer.parseInt(args[0]);

    try{
      ServerSocket serverSocket = new ServerSocket(port);
      System.out.println("Servidor iniciado na porta " + port);

      while(true){
        Socket socket = serverSocket.accept();

        String remoteAddress = socket.getRemoteSocketAddress().toString();
        System.out.println("Cliente conectado: " + remoteAddress);

        BroadShareServerThread shareServer = new
BroadShareServerThread(socket, clientList);
        shareServer.start();
      }
    }catch (IOException ex){
      System.out.println("Server exception: " + ex.getMessage());
      ex.printStackTrace();
    }
  }
}
```

APÊNDICE R: ATUALIZAÇÃO DOS MÉTODOS DA CLASSE “FILEPROTO” PARA DISTRIBUIÇÃO DE ARQUIVOS

```
  public static String makePutResponse(boolean status, String message)
throws Exception{
    String header = "";
    header += "type: put-response\n";
    header += "status: failure\n";
    header += "message: " + message + "\n";
    header += "\n";
    return header;
  }

  public static String makeSubscribe() throws Exception{
    String header = "";
    header += "type: subscribe\n";
    header += "\n";
    return header;
  }

  public static String makeSubscribeResponse(boolean status, String
message) throws Exception{
    String header = "";
    header += "type: subscribe-response\n";
    header += "status: failure\n";
    header += "message: " + message + "\n";
    return header;
  }
```

```
public void handlePut(HashMap<String, String> headerMap,
ArrayList<FileProto> clientList) throws Exception{
    String filename = headerMap.get("filename");
    String filesize = headerMap.get("filesize");
    String crc = headerMap.get("checksum");
    boolean distribute = false;
    System.out.println("Recebendo: '" + filename + "'");
    this.receiveFile(filename, filesize);
    System.out.println(" (Concluido!)");
    String responseHeader = "";
    File file = new File(this.repos + filename);
    long fileCrc = -1;
    if(file.isFile()){
        fileCrc = this.calcFileCRC(filename);
        if(file.length() == Long.parseLong(filesize) &&
          fileCrc == Long.parseLong(crc)){
          responseHeader  = this.makePutResponse(true);
          distribute = true;
      }else{
        String message = "Nao foi possivel salvar o arquivo";
        responseHeader  = this.makePutResponse(false, message);
      }
    }else{
      String message = "Arquivo recebido CORROMPIDO";
      responseHeader  = this.makePutResponse(false, message);
    }
    this.sendHeader(responseHeader);
    if(distribute){
      synchronized(clientList){
        for(FileProto client : clientList){
          if(client != null){
            try{
              String header = FileProto.makePut(filename, file.length(),
fileCrc);
              client.sendHeader(header);
              client.sendFile(this.repos + filename);
            }catch (Exception ex){
              clientList.set(clientList.indexOf(client), null);
              System.out.println("'" + filename + "' -> " +
client.getRemoteAddress() + ": ERRO!");
            }
          }
        }
      }
    }
  }
```

APÊNDICE S: CLASSE “FILERECEIVER”

```
import java.io.*;
import java.net.*;
import java.util.*;

public class FileReceiver{

    public static void main(String[] args){
    String hostname = "localhost";
    int port = 9000;

    try{
      String config = FileProto.readFromFile("config");
      HashMap<String, String> configMap = FileProto.parseHeader(config);
      if(configMap.containsKey("host")) hostname = configMap.get("host");
      if(configMap.containsKey("port")){
        port = Integer.parseInt(configMap.get("port"));
      }
    }catch (Exception ex){
      System.out.print("Verifique o arquivo 'config'\nhost: {ip}\nport:
{porta}");
      System.exit(0);
    }

    try{
      Socket socket = new Socket(hostname, port);
      FileProto proto = new FileProto(socket);

      String subscribe = FileProto.makeSubscribe();
      proto.sendHeader(subscribe);

      while(true){
        String header = proto.receiveHeader();
        System.out.println(header);

        HashMap<String, String> headerMap = proto.parseHeader(header);
        String filename = headerMap.get("filename");
        String filesize = headerMap.get("filesize");
        String crc = headerMap.get("checksum");

        proto.handlePut(headerMap);
      }
    }catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```