Servidor de Arquivos

Marco Cezar Moreira de Mattos[1], **Rômulo Manciola Meloca**[1]

[1]DACOM – Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87301-899 – Campo Mourão – PR – Brazil

{marco.cmm,rmeloca}@gmail.com

Abstract. *This report shows the development process of a file server software, since the conception, showing the designed solution until its implementation phase. This report shows project decisions and outlined obstacles.*

Resumo. *Este relatório apresenta o processo de desenvolvimento de um software servidor de arquivos, desde a sua concepção, apresentando a solução projetada até a fase de implementação do mesmo. Lê-se neste decisões de projeto e obstáculos contornados.*

1. O Problema

Disponibilizado no *moodle*, serviço de apoio educacional, chegou-se aos alunos o problema de transformar uma aplicação do modelo cliente-servidor *single-thread* em um programa que permitisse o acesso de vários clientes, bem como que cada um deles pudesse requisitar uma listagem do diretório e/ou o *download* de algum arquivo presente nele, tornando-o um servidor de arquivos. Além de permitir uma *thread* para cada cliente, deveria-se utilizar o paradigma produtor-consumidor *n:m*, onde várias *threads* produziriam requisições em um buffer compartilhado e várias *threads* consumidoras tratariam devidamente cada requisição.

Juntamente com o exemplo, foi disponibilizado também uma bliblioteca de funcionalidades que encapsulam as especificidades do protocolo TCP/IP com a linguagem de programação C, da qual o próprio exemplo dado a utilizava.

2. Organização da Solução

De início, fazia-se necessário a diagramação do problema para dar início ao desenvolvimento da solução, pois havia a necessidade de, em primeira instância, compreender o problema e projetar a solução, além de estabelecer uma linguagem comum para os desenvolvedores. Nesse sentido, utilizou-se o diagrama de classes e alguns outros esquemas de autoria própria.

Uma vez fornecida a biblioteca de encapsulamento do TCP/IP, já preparada para o desenvolvimento da solução, optou-se por implementar a solução na linguagem de programação C.

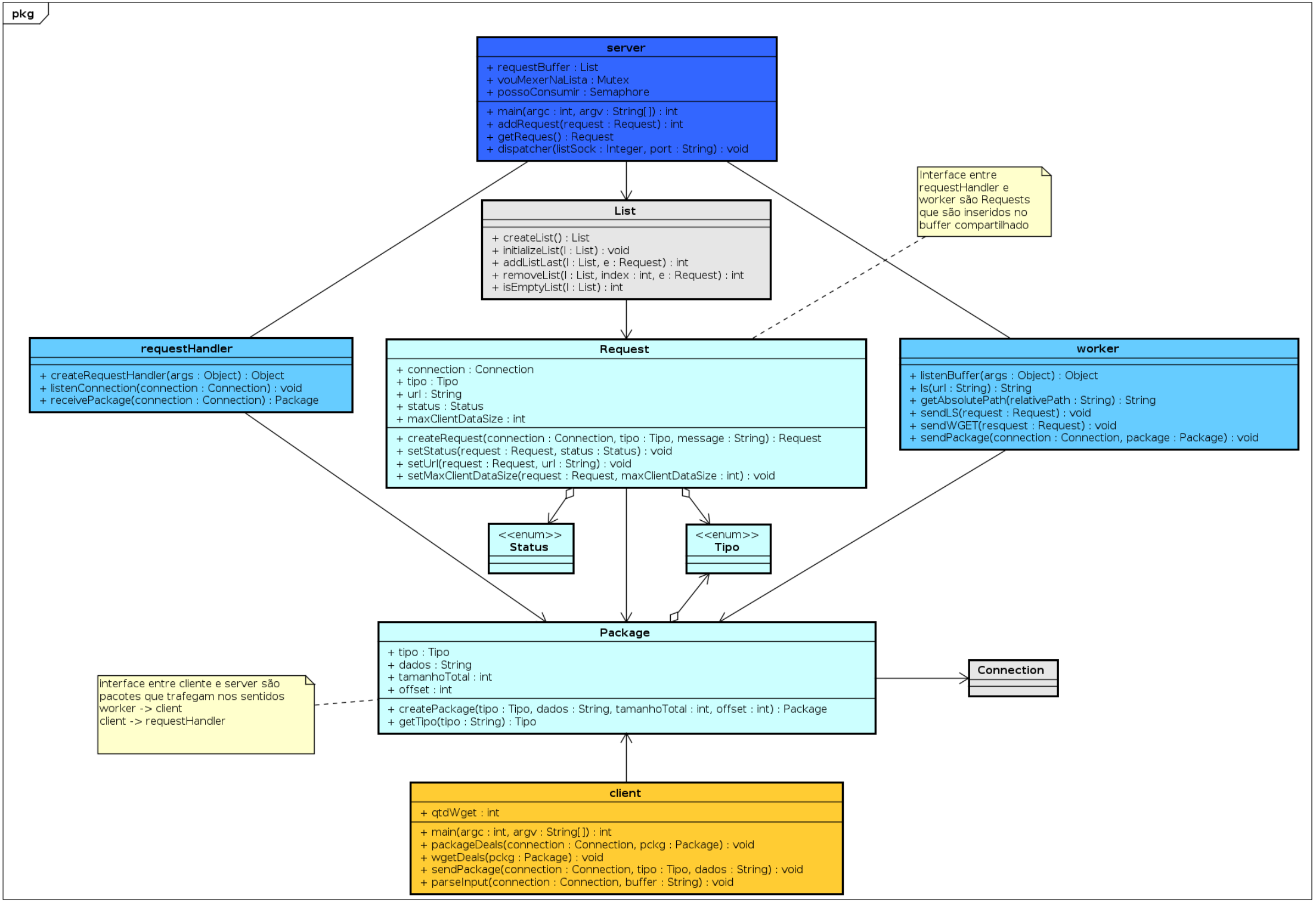
2.1. Diagramação

Como o diagrama de classes é uma ferramenta de modelagem de aplicações orientadas a objetos, foi preciso fazer algumas adaptações para o desenvolvimento procedural: As classes com inicial maiúscula são estruturas, e seus elementos são os atributos; As classes com inicial minúscula são arquivos; Os atributos oriundos de classes-arquivos são

variáveis globais; Os modificadores de acesso são todos públicos, uma vez que não há nenhuma proteção nas variáveis (embora tenha-se optado por fazer o acesso apenas através de “getters”e “setters”); Os métodos de nome create* simulam o método construtor; As associações representam os includes entre os arquivos; Tipos primitivos são mantidos e ponteiros tratados como objetos (salvo tipos enumeráveis, que também são tratados como objetos), como por exemplo String ao invés de char*; Associações bi-direcionais, mantêm a semântica de includes bi-direcionais.

Utilizou-se no diagrama cor cinza para os elementos previamente implementados e reutilizados, tons azuis para objetos inerentes ao processamento do servidor e tom amarelo para o cliente. Azul ciano para as interfaces de abstração dos dados a serem trafegados. Azul turquesa para as threads. Azul marinho para o servidor.

Segue a imagem do diagrama construído com o auxílio do *software* gratuito de modelagem Astah.



Acima visualizou-se o diagrama de classes mostrando os arquivos e estruturas criadas, bem como os métodos e atributos que detectou-se em cada objeto. Na subseção seguinte são detalhadas as interfaces criadas.

2.2. Interfaces

Lançou-se mão das bibliotecas já implementadas *lista.h*, que abstraiu os detalhes de gerenciamento de uma lista encadeada e do módulo *connection.h*, que abstraiu as especificidades do TCP/IP. Com o uso das bibliotecas mencionadas permitiu-se ainda que esses módulos sejam facilmente substituídos por outros, como por exemplo a troca de uma lista encadeada para uma lista de prioridades.

Trabalhando acima das camadas de *software* mencionadas, ainda eram necessárias mais abstrações com a finalidade de permitir as *threads* produtoras e consumidoras pudessem estabelecer um diálogo. Nesse sentido, criou-se a estrutura *Request*, que congrega a conexão com o cliente (para permitir a *worker-thread* responder adequadamente ao cliente), o tipo da requisição (Boas-vindas, encerramento de conexão, listagem de arquivos, *download*, tamanho máximo suportado, ou outro), o caminho (para listagem de arquivos e pedido de *download*), um *status* para monitorar o andamento da requisição, e ainda, o valor do tamanho máximo de dados que o cliente suporta.

Uma outra decisão de projeto tomada pela equipe, foi a de criar ainda uma abstração a mais para o encapsulamento da mensagem que transitaria entre o cliente e o servidor. Assim, criou-se a estrutura *Package* que conta com aquele mesmo tipo enumerável da requisição, os dados contidos no pacote, um tamanho total da mensagem a ser enviada bem como seu *offset*, a fim de calcular, para cada pacote, qual é a sua porção na mensagem total, além de permitir o cálculo de quantos são os pacotes no total, para cada requisição do cliente.

Abaixo, segue um dos artefatos produzidos para representar o fluxo de dados durante o atendimento de uma requisição.

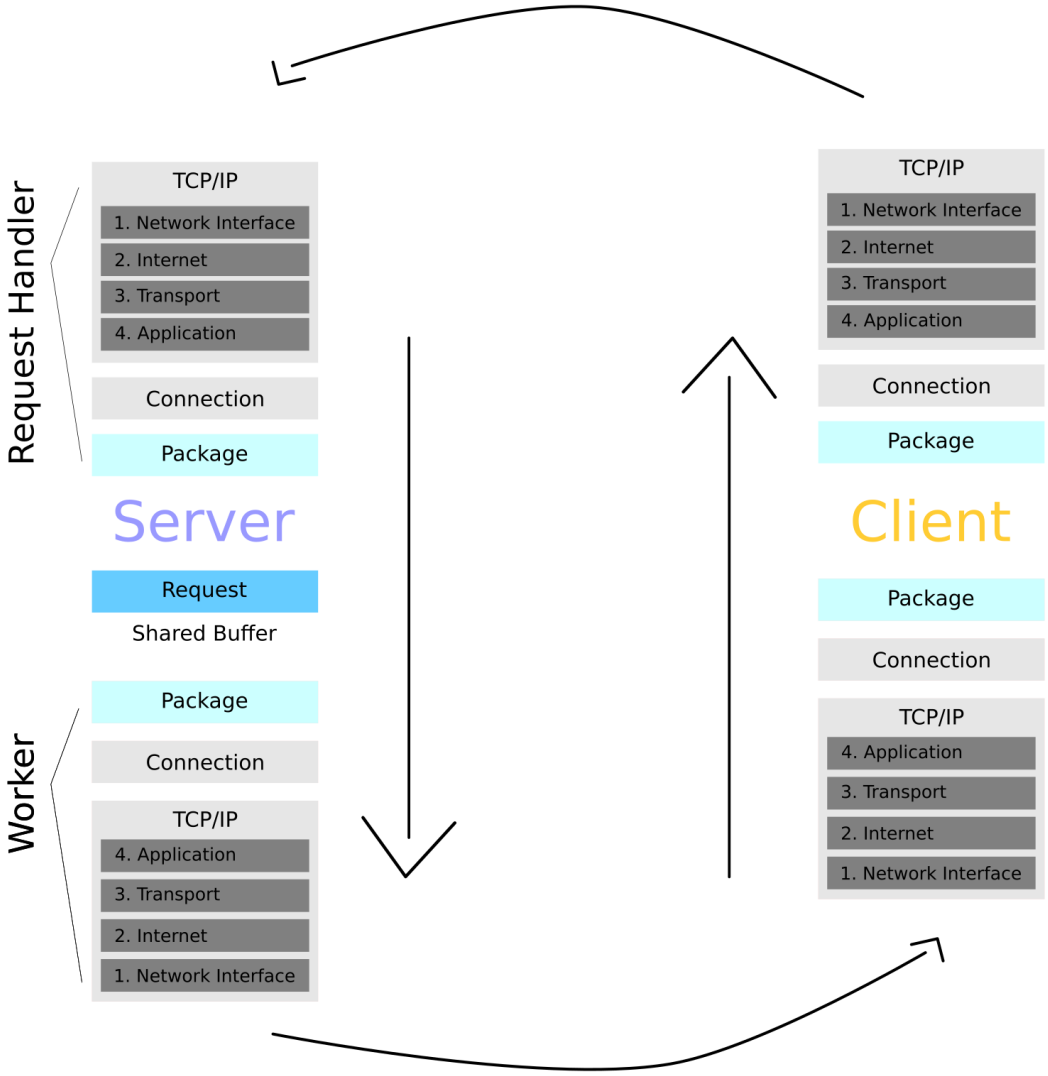


Figura 2. Fluxo de dados

Observa-se no fluxo de dados que a mensagem é encapsulada em pacotes (para além dos segmentos, pacotes e quadros do TCP/IP), de modo que ambos cliente e servidor precisam compreender essa linguagem. Uma vez que o pacote é recebido pelo servidor na sua *thread Request Handler*, responsável pelo cliente em questão, o pacote é interpretado e transformado em uma requisição, que é posta no *buffer* compartilhado entre todas as *threads*. Assim que disponível, alguma das *Worker-threads* por sua vez, toma a requisição, a executa e responde para o cliente em forma de pacotes.

Avistava-se ainda um último problema de projeto, pois algumas requisições do cliente poderiam demandar várias respostas seguidas do servidor. A subseção seguinte discute esse ponto.

2.3. Protocolo

É verdade que utilizou-se o protocolo TCP/IP para mediar a transmissão de *bytes* entre cliente e servidor e é verdade que este divide os dados em partes para enviá-lo, conforme o

Maximum Transmission Unit (MTU) da rede e ainda remonta os pacotes na ordem correta. Contudo, algumas requisições eventualmente precisariam de várias respostas ou por ser um arquivo demasiado grande ou porque o cliente suporta apenas pacotes menores.

Em qualquer dos casos, precisava-se criar um protocolo de comunicação que suportasse o envio de informações *multi-part*, seja para *download* de arquivos, ou por requisições do tipo *ls* (listagem de arquivos) em diretórios muito extensos, que, sobretudo, fosse capaz de remontar os pacotes a exemplo do TCP/IP.

O problema residia no momento de perguntar ao cliente qual o tamanho máximo de dados que ele suportava, dado que a *worker-thread* não tinha condições de escutar o cliente, uma vez que a *thread Request Handler* mantinha-se em escuta ao cliente. Também não seria possível que a *thread Request Handler* enviasse a resposta da pergunta para a *worker thread* correta. Em todos os casos, fugia da responsabilidade da *worker* escutar a conexão e fugia da responsabilidade do *Request Handler* enviar pacotes.

Frente a essa dificuldade, pensou-se e criou-se o protocolo que pode ser visto na imagem a seguir.

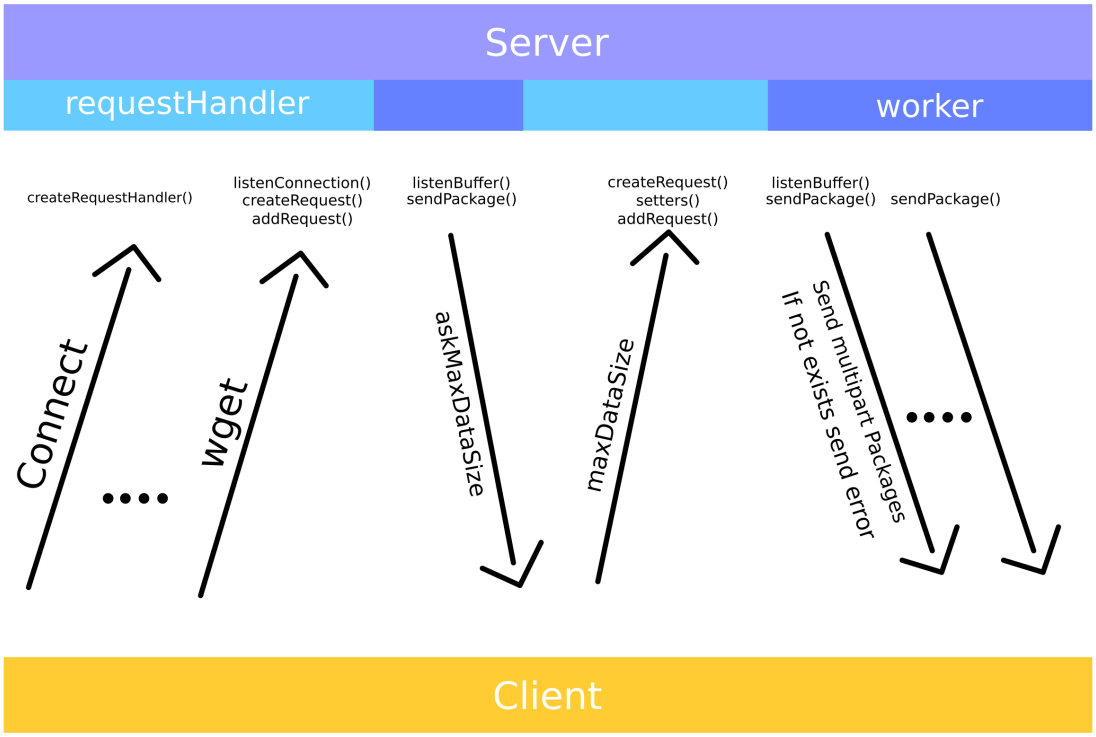


Figura 3. Protocolo de transferência *multi-parts*

A solução consiste no *Request Handler* "segurar os pacote de pedidos do tipo *ls* e *wget* e, ao invés de transformá-lo em uma requisição e colocá-lo no *buffer* compartilhado, enviar para a fila de trabalho das *worker-threads* uma requisição do tipo MAXPACKAGESIZE, que questiona ao cliente qual o tamanho máximo que ele suporta. Ao receber o pacote de resposta, a *thread Request Handler* acorda e pode prosseguir com o tratamento da requisição anterior, informando à *worker-thread* qual o tamanho máximo de cada pacote.

Tendo sido preenchidas as maiores lacunas do problema e fechadas todas as arestas no tocante à dificuldades com a estruturação da solução, já era possível implementar a solução. A seção seguinte expressa essa etapa.

3. Implementação

Dado as bibliotecas já implementadas, como mencionado na seção anterior, desenvolveu-se a solução com a linguagem de programação C.

Além das abstrações já citadas, para facilitar o desenvolvimento e torná-lo mais natural, tornando-o mais próximo ao conceito de objetos, utilizou-se apelidos característicos de objetos para estruturas já nomeadas (como por exemplo *Connection* ao invés de *connection_t*) e definições de métodos redundantes (como por exemplo *sendPackage()* em contraposição à *CONN_send()*). Tais fatores, permitiram o encapsulamento de algumas funcionalidades e possibilidade de fácil e rápida manutenção/refatoração do código.

Tendo sido mapeado os *includes* no diagrama de classes, facilmente pode-se visualizar os módulos que os programas *server* e *client* deveriam incluir, mantendo a coerência com a distância que as partes (naturalmente) deveriam ter, com excessão da interface que os conecta, os pacotes definidos na biblioteca *package.h* (que já conta com a inclusão da biblioteca *connection.h*).

Colaborou-se o código com o auxílio do controle de versões git, onde o integrante Rômulo responsabilizou-se pela implementação do arquivo *server.c* e das duas *threads requestHandler.c* e *worker.c*. O núcleo da aplicação foi desenvolvido de maneira conjunta e interativa em relação as partes. O integrante Marco responsabilizou-se por todo o arquivo *client.c* além do *upload* do arquivo em *multi-parts*.

4. Considerações Finais

Considera-se, por fim, que o desenvolvimento de um projeto que conta com várias *threads*, o conceito de produtor-consumidor, um *buffer* compartilhado onde apresentam-se condições de corrida bem específicas, a integração de vários módulos além da comunicação inter-processos via *socket*, permite no mínimo alargar os conhecimentos e fixar o aprendizado de todos esses conceitos vastamente utilizados nas mais diversas aplicações atuais. Nesse espectro, salienta-se a importância de tal desenvolvimento e, sobretudo, a fase de projeto, que tanto agrega para a visualização panorâmica deste ponto em específico da disciplina de Sistemas Operacionais.

Conclui-se que aplicações de determinada escala demandam a produção de vários artefatos, que somente são produzidos após o fiel debruçar-se nas ideias e adiantar-se a respeito de todos os problemas que são solucionados e gerados a partir delas. Toma-se como proveito o pensar em soluções que possam ser integradas a demais programas e o pensar em abstrações e interfaces que possam ser escaladas e reutilizadas.