# Relatório de Projeto de Infraestrutura de Comunicação

Tiago Figueiredo Gonçalves - tfg@cin.ufpe.br
Lucas Vinicius da Costa Santana - lvcs@cin.ufpe.br
Diogo Oliveira Rodrigues - dor@cin.ufpe.br
Gabriela Araujo Britto - gab@cin.ufpe.br

## 1. Projeto 1

Essa etapa do projeto foi dividida em 3 partes, uma para cada protocolo usado.

### 1.1 Comunicação via TCP

A ideia é que o servidor ecoe tudo que ele receber do cliente. Nessa parte, foi aberto um socket TCP no servidor na porta 27015, o qual fica sendo escutado pelo servidor por alguma tentativa de conexão de um cliente. O cliente, se conecta com o servidor e o servidor aceita a conexão. Tudo que o cliente envia, o servidor ecoa de volta pro cliente. Ao final da troca de mensagem, a conexão é encerrada.

### 1.2 Comunicação via UDP

A ideia é que o servidor ecoe tudo que ele receber do cliente. Nessa parte, foi aberto um socket UDP no servidor na porta 27015 e um socket UDP no cliente na porta 27018. Tudo que o cliente envia, o servidor ecoa de volta pro cliente. Os métodos referentes a essa troca de mensagem são via UDP, podendo haver perda do datagrama. Ao final do envio e recebimento da resposta do servidor, o cliente fecha seu socket.

### 1.3 Troca de mensagens HTTP

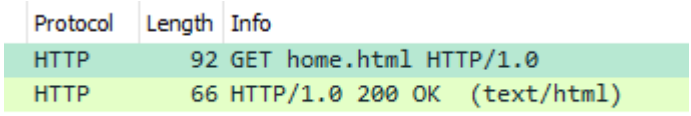
A ideia é que o servidor possua um arquivo "home.html" e que ele retorne tal arquivo caso seja requisitado por algum cliente. Nessa parte, foi aberto um socket TCP no servidor na porta 80, o qual fica sendo escutado pelo servidor por alguma tentativa de conexão de um cliente. O cliente, se conecta com o servidor e o servidor aceita a conexão. O cliente envia uma mensagem no formato HTTP. O servidor, como é bem simples, apenas pode retornar resposta para um tipo de requisição, isto é, um GET do arquivo home.html. Caso a requisição seja isso, ele retorna o arquivo. O cliente imprime o arquivo recebido e fecha seu socket. Ao inicializar o servidor, ele deve escolher que tipo de comunicação ele irá aceitar. Ao inicializar o cliente, ele deve escolher que tipo de comunicação ele irá fazer. Caso o servidor esteja aceitando esse tipo de comunicação, ele irá conseguir se comunicar com o servidor.

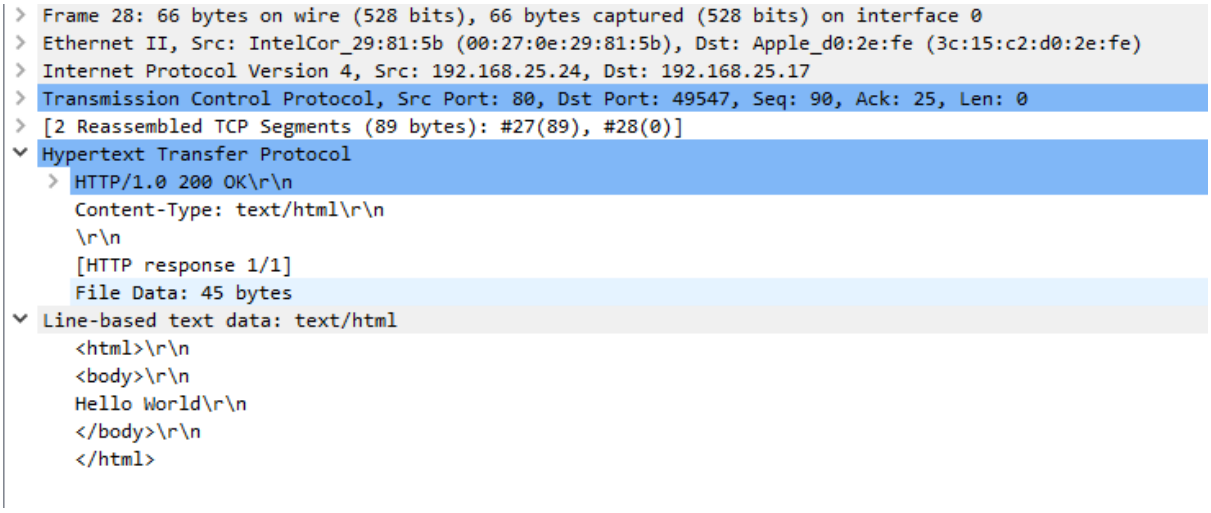
### 1.4 Como rodar

Execute o server.py em algum host e selecione que tipo de protocolo de comunicação ele fará. Depois, execute o client.py em algum host, escolha o protocolo e forneça o IP do servidor. Finalmente, caso tenha sido escolhido o protocolo UDP ou TCP, forneça uma mensagem para ser enviada ao servidor, que será ecoada pelo servidor. No caso de ter sido escolhido o protocolo HTTP, a requisição do arquivo home.html será feita automaticamente.

### 1.5 Wireshark

Nas figuras abaixo, pode-se observar a captura de uma troca de mensagens HTTP desta aplicação, feita pelo Wireshark.




## 2. Projeto 2

Para a realização da parte II foi necessária uma infraestrutura grande por parte do servidor, de maneira que todo o resto funcionasse de maneira simples. Neste relatório será explicado cada módulo criado e a importância de cada um. Como existem muitos métodos em cada módulo, seria inviável descrever cada um, então será dada apenas uma ideia do que cada módulo deve fazer.

## 2.1 Módulos

Existem duas pastas para esta parte, uma para o cliente e outra para o servidor. E cada pasta temos os módulos necessários para este.

### 2.1.1 Pasta server

- **DataBase.py**

  Para salvar os dados do servidor precisamos de um banco de dados para persistir as informações dos clientes e as pastas. Para um banco de dados este módulo cria um pasta onde cada entidade fica em um arquivo no disco. Ele possui métodos set para dar um valor para uma chave e get para recuperar o valor de uma chave.

- **FileSystem.py**

  Este módulo cuida do sistema de arquivos em geral, tem-se uma "árvore" onde cada pasta e arquivo possui um identificador único, e as pastas possuem uma lista com o identificadores dos itens (pastas ou arquivos) que estão dentro dela (filhos). Cada nó também salva o nome do item, o tipo (pasta ou arquivo), quais são as pessoas que podem abrir tal item, o identificador da pasta que o item está salvo (pai), por fim, salva um identificador para outro banco de dados, que salva os dados de fato de cada item (para uma pasta quem são os filhos, para um arquivo é um conteúdo do arquivo).

- **DataManager.py**

  Como visto anteriormente, separou-se as informações extras dos arquivos (ou pastas) dos dados. Assim, quando é preciso pegar informações sobre um não precisamos carregar todo o arquivo, que pode ser grande. Este módulo trata de gerenciar isto criando um banco de dados e dando identificadores para os dados de cada arquivo.

- **ClientManager.py**

  Um cliente possui um login e uma senha e duas pastas-padrão, uma que é onde ele vai criar arquivos e pastas, e outra onde ele vai encontrar todos os itens que foram compartilhados com ele.
  Este módulo gerencia os clientes, para isso cria-se um banco de dados onde é salvo login, senha e identificadores das duas pastas padrão de um usuário.

- **SystemManager.py**

  Este módulo faz basicamente o link entre o ClientManager e o FileSystem Ele serve com interface para os dois, fazendo os links necessários. Assim, de maneira geral, este módulo é a "inteligência" do sistema, onde, por exemplo, se verifica se um cliente pode abrir um arquivo antes de enviá-lo.

- **RequestManager.py**

  Este módulo possui um único método que é dado um request, retornar a resposta ou uma mensagem de erro. Isto ainda não tem nada a ver com conexão. O request tem que ser um dicionário de python, e deve conter os seguintes campos: 'login', 'password', 'type'.

  Dependendo do 'type', que pode ser: REGISTER, AUTHENTICATE, GET_ITEM_NAME, GET_ITEM_DATA, GET_ITEM_TYPE, GET_ITEM_PARENT, CREATE_FOLDER, CREATE_FILE, GET_ROOT_FOLDER, GET_SHARED_FOLDER, SHARE_ITEM, RENAME_ITEM, EDIT_FILE_DATA, MOVE_ITEM. Alguns outros campos são necessários, que são: where, name, itemId, data, fileId, newData, newParent. Cada um é obrigatório dependendo do request, e eles são auto-explicativos.

  As resposta também são dicionário de python e contém os campos: status, code, answer.

- **server.py**

  Aqui é onde socket é utilizado, por simplicidade, para cada nova conexão uma nova thread é criada para tratar desta conexão. Este módulo simplesmente recebe a mensagem da rede, usa o RequestManager para tratar o request e envia a resposta de volta para o usuário.

### 2.1.2 Pasta client

- **NetworkManager.py**

  Módulo que implementa as funções que enviam e recebem mensagens pela rede. É necessário um cuidado especial com mensagens grandes (como os arquivos). Desta maneira, utilizando o módulo struct, nos primeiro 4 bytes da mensagem temos a informação de seu tamanho, e em seguida a mensagem de fato. Também vale lembrar que como as mensagens trocadas entre servidor e cliente são dicionários, faz-se uso do módulo pickle para serializar e deserializar os dados.

- **ApplicationManager.py**

  Para melhor uso da aplicação, foi implementado um sistema baseado no terminal do linux. Os seguintes comando foram implementados:
  **help**: listar todos os comandos
  **register**: criar uma conta
  **login**: entrar em uma conta criada
  **logout**: sair da conta que logou
  **ls** *[caminho]*: mostrar o conteúdo de uma pasta, dada em caminho ou a pasta atual (se o argumento não foi dado).
  **cd** *[caminho]*: mover para a pasta dada pelo caminho, caso o caminho não seja expecificado, é movido para a pasta raiz
  **mkdir** *nome*: cria uma pasta vazia com o nome dado, na pasta atual.

**mkfile** *caminhoArquivo*: fazer o upload do arquivo identificado pelo **caminho** *caminhoArquivo* no sistema de arquivo local. Ou seja, o arquivo deve existir no computador.
**share** *caminho usuario*: compartilha o arquivo ou pasta especificado pelo caminho com o usuário dado com entrada
**download** *caminho*: baixa o arquivo dado pelo caminho e salva no sistema local.
**clear**: limpa a tela
**quit**: fecha o programa

Para comunicação com o servidor o módulo NetworkManager foi utilizado. Um dicionário salva todos os comandos, uma breve descrição sobre cada um e uma referência para a função correspondente.

- **client.py**

  Módulo basicamente responsável por inicializar o módulo NetworkManager, conectando com o servidor dado como entrada, e por ler os comandos digitados pelo cliente chamando o método(do ApplicationManager) para processá-lo.

### 2.2 Como rodar

Para rodar o servidor basta digitar, na pasta *Part2/server,*
*$ python3 server.py*
O servidor escuta os requests na porta 20041 usando o protocolo TCP da camada de transporte.
Já para o cliente, na pasta *Part2/client*
*$ python3 client.py hostname port*
Onde hostname é o endereço do servidor, e port é a sua porta (neste caso 20041).

## 3. Projeto 3

Esta parte do projeto foi desenvolvida em três arquivos, *client.py*, que executa o código do cliente, *TicTacToe.py*, que é a classe que representa um jogo da velha, e *server.py*, que executa o código do servidor.

### 3.1 Servidor

O servidor é bastante simples, pois na organização do projeto foi adotada a ideia de que a complexidade deveria estar concentrada no cliente, já que o protocolo da camada de transporte é UDP. Ou seja, o cliente vai lidar com problemas como perda de pacote, desse modo o servidor tem poucas responsabilidades.

No servidor há uma fila contendo endereços (no formato (IP, porta)) de clientes que desejam jogar. O servidor funciona num *loop* da seguinte forma: inicialmente, o servidor aguarda até receber uma mensagem UDP na porta 1337, cujo conteúdo é a mensagem ‘PLAY’, que identifica uma solicitação de um jogador. Essa solicitação é então processada, de forma que se o requisitante,

identificado por um endereço (IP, porta), já está aguardando na fila, ele não é acrescentado à fila novamente, mas caso contrário é adicionado ao final da fila de endereços. Após a requisição ser processada, enquanto for possível fazer um *match* entre dois solicitantes distintos, o servidor envia uma mensagem para cada endereço solicitante, contendo o endereço do outro jogador e o seu turno (se o jogador irá jogar como 'O' ou 'X'). Como já foi mencionado, o servidor não trata problemas de perda de pacotes ou perda de conexão de jogadores.

## 3.2 Cliente

O cliente tem então a responsabilidade de garantir que um oponente seja encontrado e também tratar a perda de pacotes, chegada fora de ordem e perda de conexão do outro oponente. Uma perda de conexão é definida como o oponente não se comunica de forma apropriada dentro de um certo intervalo de tempo definido por parâmetros do código.

Inicialmente o cliente deve encontrar um oponente. É enviada periodicamente uma solicitação ao servidor escolhido (cujo IP é solicitado na execução do programa), e entre uma solicitação e outra verifica-se periodicamente se houve resposta do servidor contendo o endereço do oponente. Esse processo é repetido até que se obtenha o endereço de um oponente. Durante o envio da solicitação ao servidor, se este não estiver recebendo mensagens no IP de entrada e na porta estabelecida, é lançada uma exceção e o usuário é informado de que não foi possível se comunicar com o servidor, e então é possível que o usuário insira um IP diferente.

Após a obtenção do oponente, o jogo é iniciado. O jogo em si consiste de um *loop* que se repete até o final do jogo. Existem dois casos: o turno é do cliente ou de seu oponente, e cada turno do jogo é numerado sequencialmente a partir do 0. Quando é a vez do cliente, uma jogada é solicitada. A jogada deve estar no formato *linha coluna*, em que o canto superior esquerdo é a posição *0 0* e o canto inferior direito é a posição *2 2*. A validez da jogada é verificada e depois que o usuário digita uma jogada válida, a jogada é enviada. A informação enviada consiste nos dois inteiros que indicam a jogada e um terceiro inteiro que indica o turno correspondente à jogada (identificado no código como *run*). Esse número possibilita identificar uma jogada atual e uma jogada antiga. Essa informação é enviada periodicamente até um limite máximo de envios até que se obtenha uma mensagem de confirmação contendo a string 'ACK' e o número correspondente àquele turno. Se a confirmação não for recebida, considera-se que o oponente foi desconectado.

Quando a vez é do oponente, verifica-se periodicamente, até um limite máximo de verificações, o recebimento de um pacote UDP. Se este contiver uma jogada válida do turno atual do oponente, é enviado um 'ACK' correspondente a esse turno e o jogo prossegue. Se a jogada não for recebida em tempo hábil, considera-se que o oponente foi desconectado.

Em ambos os casos acima, se durante a leitura de pacotes UDP for recebida uma mensagem de uma jogada de um turno já passado, uma mensagem de confirmação para esse turno é enviada em resposta, enquanto que se o cliente está esperando um *ACK* para a jogada atual e recebe já a

próxima jogada do oponente, esta é ignorada, o que simplifica o fluxo do código. Neste caso então o cliente irá esperar a mensagem de confirmação, e só depois irá esperar pela jogada do oponente, que será reenviada pois o cliente ainda não terá enviado o *ACK* correspondente. Note que esta situação ocorre se a respectiva mensagem 'ACK' do oponente for perdida.

### 3.3 Jogo da Velha (TicTacToe)

Essa classe contém toda a lógica do jogo. A classe TicTacToe tem um construtor para inicializar um jogo com uma matriz vazia, onde o primeiro jogador é 'O' e o segundo jogador é 'X'.

O método "checkWin" serve para saber o estado do jogo. Ele retorna 'O' se o primeiro jogador venceu, retorna 'X' se o segundo jogador venceu, retorna 'Draw' se foi empate e 'Active' se o jogo ainda está em andamento. Ela usa "__checkWin" para ver se existe um padrão de vitória para algum caractere na matrix, ou seja, linha ou diagonal ou coluna preenchida com o mesmo caractere.

O método "makePlay" recebe as coordenadas de uma jogada e checa se é uma jogada válida. Se for, efetua a jogada na matriz e retorna verdadeiro. Se não for, retorna falso para que o usuário saiba que a jogada não foi válida.

Além disso, existe um método para ver de qual jogador é o turno: getTurn. Apenas isso é o bastante para cobrir toda a lógica do jogo.

### 3.4 Como rodar

Abra o server.py em algum host. Abra o client.py em dois hosts e digite o IP do host do server. O server irá estabelecer conexão entre os dois clients. Se algum dos clients não estiver mais conectado, o conectado receberá uma mensagem de desconexão. Notar que o processo de pegar input do usuário é bloqueante, a mensagem de desconexão aparece apenas após pegar a entrada, caso seja necessário.