

# **Trabalho Prático 1 – Redes de Computadores**

**Vitor Rodarte Ricoy - 2019007112**

Universidade Federal de Minas Gerais (UFMG)  
Belo Horizonte – MG – Brasil

## **1. Introdução**

O trabalho proposto tem como objetivo treinar e fixar os conceitos aprendidos em aula relativos à comunicação de cliente e servidor por meio de sockets. Esse trabalho consiste na implementação de um programa servidor e de um programa cliente. O servidor possui uma pokédex com nomes de pokémons e aceita conexões de clientes que podem enviar quatro tipos de operações diferentes a serem realizadas na pokédex pelo servidor: adicionar um pokémon, remover um pokémon, trocar dois pokémons e listar os pokémons cadastrados na pokédex. O cliente se conecta com o servidor e envia mensagens lidas da entrada padrão.

## **2. Dificuldades**

### **2.1. Utilização da Biblioteca de Sockets**

Essa foi uma das principais dificuldades que tive ao implementar o trabalho. Acredito que a dificuldade foi natural, já que esse foi meu primeiro contato com a biblioteca. Apesar disso, a aula sobre código apresentada na disciplina e a documentação da biblioteca foram essenciais para compreender melhor o seu uso e implementar o trabalho. Tirando essa questão do contato inicial com a biblioteca não tive mais problemas relacionados a ela, já que ela funcionou como eu esperava e não obtive nenhum erro ao usar as suas funções.

### **2.2. Escrita de código em C**

Outra dificuldade que tive foi a escrita de código em C. O meu último contato mais profundo com a linguagem ocorreu há um tempo considerável, portanto eu não lembrava muito bem de algumas especificidades da linguagem, principalmente os seus contrastes com C++. Também, eu não conhecia muito bem os seus recursos, ou a falta deles, principalmente para a manipulação de strings.

Apesar da grande quantidade de erros de compilação e a reescrita de diversas partes do código devido a essa dificuldade, no final a linguagem não foi um grande impeditivo para a implementação do trabalho, ainda mais após consultar a documentação de diversas funções da biblioteca "string.h".

### **2.3. Desenvolvimento da Estrutura da Pokédex**

Outra dificuldade que tive, mas que teve um impacto menor, foi elaborar a estrutura de dados que define a Pokédex, principalmente ao considerar os recursos disponíveis em C. No fim, consegui implementá-la de uma forma simples que, apesar de ineficiente para listas muito grandes, atende os requisitos do trabalho. Essa dificuldade com a estrutura da Pokédex se relaciona também com a dificuldade que citarei a seguir, a respeito da dificuldade em organizar o código e deixá-lo mais legível.

### **2.4. Legibilidade e Organização do Código**

A última dificuldade que identifiquei foi escrever o código de uma forma mais organizada e legível. Essa dificuldade aconteceu, em grande parte, porque ao decidir organizar o código em funções em C, geralmente precisamos utilizar ponteiros para algumas variáveis o que, por sua vez, torna o código menos compreensível. Além disso, também existe a dificuldade de compreender o código em C no geral, já que ele costuma ser mais verboso e menos compreensível do que o código de linguagens de mais alto nível. Apesar dessa dificuldade, no fim acredito que o código ficou com uma boa legibilidade e organização.

### 3. Imprevistos

O único imprevisto que encontrei foi durante o teste do meu código, já que desenvolvi o código sem me preocupar muito com a organização e legibilidade inicialmente e, por isso, ele estava bem extenso e pouco legível. Como um exemplo, a função main do servidor possuía sozinha mais de 200 linhas. Em razão disso, tive que reescrever o código do cliente e servidor me preocupando mais com a organização, modularização e legibilidade. Devido a isso acredito que demorei mais tempo para implementar o trabalho, porém o processo de testar a minha implementação ficou muito mais eficiente.

### 4. Soluções Adotadas

Algumas observações gerais são que considere que nenhuma mensagem ultrapassa 512 bytes, portanto a declaração de arrays de caracteres seguem esse tamanho no geral, com algumas pequenas exceções que ampliei esse limite para 532 bytes, mas na prática não são enviados mais de 512 bytes de toda forma. Também foi criada uma biblioteca de funções compartilhadas entre o servidor e o cliente, que possui apenas uma função para encerrar o programa com erro e uma mensagem, e uma função de debug que converte um endereço de socket para uma string.

#### 4.1. Cliente

Para o cliente, a solução que adotei, usando a biblioteca sockets, foi um fluxo de conectar com o servidor, e se comunicar com ele em seguida. A comunicação com o servidor consiste no fluxo de ler uma mensagem do teclado, por meio do comando “fgets”, enviar essa mensagem ao servidor e, em seguida, receber a mensagem de resposta do servidor. Esse fluxo se repete até que o cliente envie um comando “kill” ou se desconecte do servidor.

Uma preocupação que tive ao desenvolver a solução foi garantir que o cliente sempre envie mensagens no formato válido, ou seja, terminadas em ‘\n’ e sem o caractere ‘\0’. Já o tratamento de caracteres inválidos é feito exclusivamente no servidor. Além disso, também defini que o cliente recebe mensagens do servidor, com o comando “recv”, até que ela termine em ‘\n’, ou seja, caso o servidor envie as mensagens particionadas o cliente é capaz de reconhecê-las.

Por fim, o cliente imprime apenas as mensagens recebidas por ele do servidor, exatamente como são recebidas.

#### 4.2. Servidor

Para o servidor, a solução que adotei foi um fluxo de receber um cliente e esperar mensagens dele até que ele se desconecte. Caso o cliente seja desconectado, o servidor passa a escutar por um próximo cliente. Também existe o caso em que um cliente envia o comando “kill”, que faz com que o servidor seja encerrado, junto com o cliente conectado.

Quanto ao recebimento de mensagens do servidor, assim como o cliente, defini que ele recebe as mensagens, usando o comando “recv”, até que ela termine em ‘\n’, para possibilitar o reconhecimento de mensagens enviadas em fragmentos. Vale notar que essa decisão também permite receber múltiplas mensagens em uma comunicação fragmentada, ou seja, caso o cliente envie uma mensagem com mais de um ‘\n’ de forma fragmentada, o servidor é capaz de lê-la corretamente.

Para a leitura de mensagens com diversos ‘\n’ utilizei a função “strtok”, e para a separação de componentes da mensagem por espaço utilizei a função “extrairStringAtéEspaco” implementada por mim e que possui um funcionamento similar à “strtok”, mas sem o efeito colateral da função devido ao seu ponteiro interno.

Também vale notar que a solução desenvolvida considera que a resposta para um comando “add” com múltiplos pokémons gera uma resposta independente para cada pokémon separada por espaço. Ou seja, caso a adição gere algum erro esse erro é impresso e o próximo pokémon do comando é tratado. Outra decisão tomada é que o comando “exchange” recebe exatamente dois pokémons e o “remove” recebe exatamente um, caso mais pokémons do que o esperado sejam

enviados, eles são ignorados.

Como foi esclarecido na reunião com a monitora, a minha solução para o servidor encerra a conexão do cliente quando ele envia uma operação digitada de forma incorreta. Já os erros de mensagem inválida, causados por uma mensagem com a operação correta, mas formato incorreto são retornados sem encerrar a conexão do cliente. O mesmo vale para as mensagens inválidas devido a caracteres não permitidos.

Finalmente, vale notar que, pela minha implementação, caso um cliente receba uma operação inválida em uma mensagem com caracteres não permitidos a conexão do cliente não é encerrada, pois o erro de mensagem inválida é gerado antes da análise das operações.

#### **4.2.1. Pokédex**

Já a minha solução para a pokédex foi implementada em uma biblioteca separada, por questão de organização do código. A estrutura de dados adotada foi um array de vetores de caracteres, com um tamanho fixo de 40 por 512 posições.

A solução implementada possui uma complexidade constante para adicionar ou trocar pokémons e uma complexidade linear para remover, listar e buscar pokémons. Vale notar que a solução escolhida não possui a melhor complexidade possível, mas funciona bem para a situação do trabalho, em que temos no máximo 40 pokémons cadastrados na pokédex.

### **5. Conclusão**

O trabalho foi dividido na implementação do cliente e servidor. Achei que a implementação de ambos e a execução dos testes ajudaram consideravelmente a entender como esse tipo de serviço é implementado, além de mostrar na prática como a comunicação por sockets é feita, pelo menos no contexto da linguagem C.

Os desafios que enfrentei para realizar o trabalho foram razoáveis e puderam ser superados sem maiores dificuldades. O mesmo aconteceu para o único imprevisto que tive, que no fim atrasou o desenvolvimento mas evitou atrasos nos testes da implementação. Já as soluções adotadas por mim foram o ponto central do trabalho, na minha opinião, pois representam o entendimento que tive do funcionamento esperado desse tipo de sistema e é a parte que consumiu o maior tempo de desenvolvimento.

Finalmente, achei o trabalho muito bom, apesar das dificuldades e desafios, para o aprendizado prático de sockets em C e para o entendimento do funcionamento da arquitetura cliente/servidor na prática. Também achei produtivo ter que pensar como fazer o servidor e cliente se comunicarem de acordo com a especificação.