

**Universidade Federal de Minas Gerais**  
**Departamento de Ciência da Computação**  
**Redes de Computadores**

**O Protocolo Olímpico UDP**  
**(Trabalho Prático 2)**  
**Documentação**

Aluno: Marlon Junior Barbosa Marques

Professor: Luiz Filipe M. Vieira

## 1 SUMÁRIO

## 2 DESCRIÇÃO DA IMPLEMENTAÇÃO

### 2.1 Suposições em torno da especificação

### 2.2 Principais estruturas de dados

#### 2.2.1 Do cliente

#### 2.2.2 Do servidor

#### 2.2.2 Doxygen

### 2.3 Resolução de endereços (IPv4 vs IPv6 vs domain name)

### 2.4 Implementação de confiabilidade via UDP

## 3 TESTES

### 3.1 Arquivos de entrada e arquivos de saída esperados

### 3.2 Testando conexão a IPv4, IPv6 e nome de domínio

## 4 CONCLUSÃO

## 5 REFERÊNCIAS

# 1 SUMÁRIO

Este Trabalho Prático trata da criação de duas aplicações: um cliente e um servidor, os quais comunicam-se através de mensagens UDP. O fluxo se dá da seguinte forma:

- cliente: <marca de tempo>
- Servidor: <posição>
- cliente: <outra marca de tempo>
- servidor: <outra posição>

Onde marca de tempo é um tempo da forma %h %m %s %ms, e a a posição é um número inteiro. A lógica por trás disso é simples: cada tempo representa o desempenho de um atleta, e cada posição indica a posição daquele atleta dado o seu tempo. Tal laço se repete até o cliente encerrar a conexão, enviando um valor negativo. Assim, um exemplo mais completo de interação cliente-servidor seria:

- cliente: 1h
- servidor: 1
- cliente: 50m
- servidor: 1
- cliente: 1h 3m 5s 945ms
- servidor: 3
- cliente: -1 <encerra conexão>

Este programa possui uma interface de linha de comando (CLI) e foi implementada na linguagem C++ para plataformas GNU/Linux. Um makefile acompanha a

implementação, e instruções de como compilar e executar o programa podem ser encontradas no arquivo README.txt em anexo.

## 2 DESCRIÇÃO DA IMPLEMENTAÇÃO

### 2.1 Suposições em torno da especificação

Certos pormenores não ficaram completamente claros na especificação deste Trabalho Prático, portanto listam-se as suposições tomadas para a fase de implementação:

- Linguagem de implementação é C++ padrão C++11 (acordado com o professor em sala de aula);
- Tanto o servidor como o cliente irão “imprimir” na tela a posição em que um determinado cliente ficou colocado. Por exemplo: se o cliente envia “5h” ao servidor, e se esta é a primeira entrada deste cliente, tanto na tela do servidor quanto na do cliente será informado a posição “1”;
- Entradas que não sigam a especificação irão ocasionar em comportamento indefinido ao programa. Por exemplo: números que não são seguidos de h, nem de m, nem de s, nem de ms;
- Ambos cliente e servidor terminarão a execução caso haja alguma problema na alocação de recursos de socket do cliente ou binding do servidor;
- O servidor não possui “botão de desliga”: o único modo de desligá-lo é matando processo. A memória alocada pelo programa é desalocada de maneira *best-effort*;
- O servidor não “limpa a tela” quando novos clientes se conectam. A entrada de clientes antigos permanece na tela, mas o ranking é renovado a cada conexão (conforme especificação).
- Um cliente se comunica apenas com um servidor e vice-versa. Não há garantias de funcionamento quando se tratam múltiplos clientes e/ou múltiplos servidores.

- O cliente espera dois segundos por uma mensagem do servidor, repetindo este procedimento por 10 vezes antes de desistir de enviar e abandonar o programa.

## 2.2 Principais estruturas de dados

### 2.2.1 Do cliente

- **ClientMain**: encapsula a função main da aplicação;
- **ClientApplication**: encapsula a captura de entrada e saída, além de interação com o usuário. Usa uma instância de ServerMediator. Seus métodos mais notórios são:
  - **runApplication**: interage com o usuário, loop principal da aplicação;
  - **void insertSeqNum (int seqNum, std::string &message)**: insere número de sequencia na mensagem, o qual pode ser 0 ou 1 (*stop and wait*);
- **UDPServerMediator**: encapsula toda a comunicação com o servidor através de sockets. Seus métodos mais notórios são:
  - **int setUpSocket(string addr, unsigned port)**: prepara um socket para se comunicar com o servidor especificado por addr:port;
  - **void cleanUp()**: limpa estruturas internas do mediador;
  - **void sendRequest(string message)**: envia a mensagem para o servidor;
  - **string getResponse(unsigned timeout)**: espera uma resposta do servidor por no máximo *timeout*, retornando a mensagem em caso positivo, ou retornando "TIMEOUT" caso nenhuma mensagem chegue a tempo;

### 2.2.2 Do servidor

- **ServerMain**: encapsula a função main da aplicação;
- **ServerApplication**: encapsula a captura de entrada e saída e conhece a regra de jogo da aplicação (o ranking). Utiliza instâncias de UDPServer e Ranking. Seus métodos mais notórios são:
  - **runApplication**: interage com o usuário, loop principal da aplicação;
  - **bool extractSeqNum (std::string &message)**: extrai o número de sequência da mensagem, o qual pode ser 0 ou 1 (*stop and wait*);
- **UDPServer**: classe singleton (apenas uma instância) que encapsula a comunicação via sockets UDP. Aceita apenas um cliente por vez. Seus métodos mais notórios são:
  - **void setUp(std::string port)**: realiza o processo de binding no socket local e na porta *port*;
  - **void sendMessageToClient(string message)** : manda a mensagem para o cliente guardado em *\_clientAddressStorage*;
  - **string getMessageFromClient()**: fica escutando na rede por um pacote UDP e retorna a mensagem escutada. Armazena endereço do remetente em *\_clientAddressStorage*;
  - **void closeConnection()**: fecha a conexão com cliente especificado em *\_clientAddressStorage*, enviando um sinal de fechamento.
  - **sockaddr\_storage \_clientAddressStorage**: estrutura responsável por guardar o endereço do cliente;
- **Ranking**: encapsula toda a lógica relacionada a persistência da aplicação. Contém a estrutura de dados que armazena os tempos dos clientes e suas posições em um *multiset de unsigned integers[1]*. Seus métodos mais notórios são:
  - **string insert(string s)**: insere o tempo indicado pela string *s* no multiset, e retorna a posição do novo elemento inserido; Converte a entrada do usuário para um valor em milisegundos;
  - **void clear()**: limpa a estrutura interna de armazenamento de tempos (elimina todos os elementos);

### 2.2.2 Doxygen

Uma documentação mais extensiva das estruturas de dados do projeto pode ser facilmente gerada. Para tal, basta instalar a aplicação *doxygen*[3] e digitar o comando *make doc* na linha de comando.

## 2.3 Resolução de endereços (IPv4 vs IPv6 vs domain name)

Esta seção descreve como a resolução de endereços (IPv4 vs IPv6 vs domain name[2]) foi realizada.

Do lado do servidor, adotou-se a seguinte estrutura da biblioteca *GNU* de sockets:

- `sockaddr_in6;`

Tal estrutura funciona com o modelo novo de endereçamento, mas também possui retrocompatibilidade, isto é, permitirá que o servidor se comunique com sockets tanto IPv4 quanto IPv6.

O funcionamento da estrutura acima mencionada se encontra encapsulado no método *setUp(string port)* da classe *Server*. Arquivo *Server.cc:14*

Já do lado do cliente, utiliza-se a mesma estrutura do lado do servidor. Entretanto, há um trabalho extra para se determinar o tipo de endereço baseado na entrada do usuário. Tal determinação ocorre no método *connectToServer(std::string,unsigned)* da classe *ServerMediator*, cuja definição se encontra no arquivo *ServerMediator.cc:16*.

De maneira breve, o processo de escolha será explicado. As seguintes estruturas adicionais e métodos se fizeram necessárias do lado do cliente:

- `in6_addr serverAddr;`
- `addrinfo hints;`
- `addrinfo* result;`
- `inet_pton(int af, const char *src, void *dst);`
- `int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);`

Por fim, o processo de escolha de endereço se dá da seguinte forma:

1. Tenta-se converter o endereço para forma binária assumindo que é IPv4 (flag `AF_INET`). Caso tenha dado certo, a estrutura *hints* será atualizada como tal;
2. Caso 1. tenha dado errado, tenta-se converter o endereço para forma binária assumindo que é IPv6 (flag `AF_INET6`). Caso tenha dado certo, a estrutura *hints* será atualizada como tal;
3. Caso 1. e 2. tenham dado errado, saberemos que se trata de um *domain name*;
4. A função *getaddrinfo()* é chamada com a estrutura *hints* definida acima, nos dando efetivamente o endereço que precisamos para conectar com o servidor;

## 2.4 Implementação de confiabilidade via UDP

A fim de se atender ao requisito deste trabalho de implementar pelo menos uma retransmissão de pacote por parte do cliente, decidiu-se implementar um



simples mecanismo de parada-e-espera (*stop and wait*)[4]. Tal decisão implica que a implementação do protocolo olímpico **é confiável**, apesar de usar um protocolo de transporte não confiável (o UDP). Tal confiabilidade passou a ser garantida pela camada superior, a camada de aplicação.

De maneira sucinta, descreve-se o fluxo de comunicação entre cliente e servidor neste modelo:

- *Cliente*: Anexa número de sequência a mensagem (0 ou 1) e envia;
- *Servidor*: Recebe mensagem, verifica número de sequência. Se número for o esperado envia mensagem atual, senão, reenvia mensagem anterior;
- *Cliente*: Fica aguardando por no máximo dois segundos por resposta do servidor. Repete esse processo por no máximo dez vezes. Se nenhuma resposta for obtida, abandona a aplicação;

O comportamento da janela deslizante (*stop and wait*) pode ser configurado através dos seguintes parâmetros, presentes em `tp2_constants.h`:

- **TIMEOUT**: número em segundos pelo qual o cliente esperará uma resposta do servidor. Valor atual: 2 segundos;
- **MAX\_NUM\_TIMEOUTS**: número máximo de timeouts que o cliente tolerará antes de desistir da comunicação com o servidor; Valor atual: 10 tentativas.

## 3 TESTES

### 3.1 Arquivos de entrada e arquivos de saída esperados

A fim de se testar a solução implementada, foram-se usados dois arquivos (anexos ao código): in1.txt e in2.txt. A saída resultante desses dois arquivos está presente, respectivamente, nos arquivos out1.txt e out2.txt. Tais arquivos serão transcritos abaixo para fins de clareza.

in1.txt	out1.txt	in2.txt	out2.txt
1s	1	10h	1
1s	2	9h 59m 59s 999ms	1
1s	3	8h 7s	1
1s	4	15h 15s	4
1s	5	15h 14s 666ms	4
1s	6	09 h 11s 666ms	2
01s	7	09 h 11s 666ms	3
0001s	8	-9	
00001ms	1		
-1			

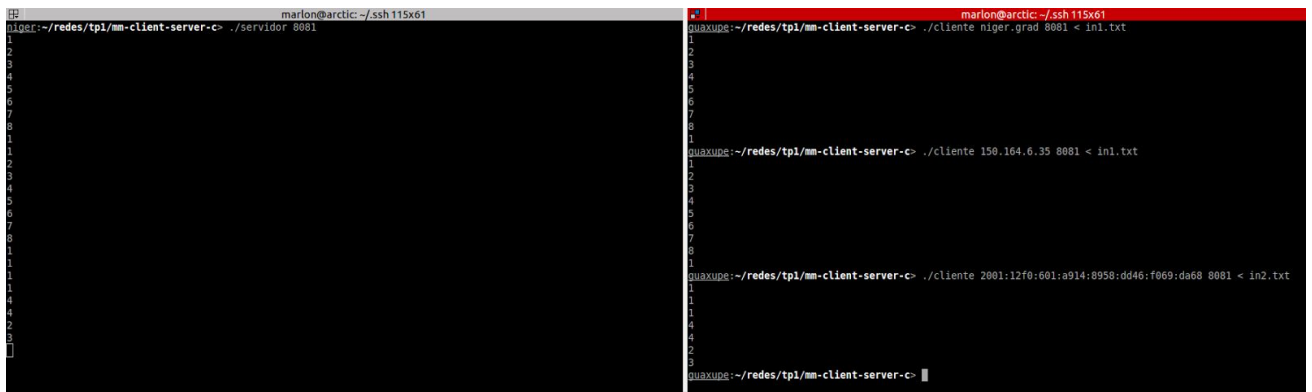
A saída do programa está seguindo as regras exposta na especificação do trabalho. A próxima seção descreve melhor a execução dos programas.

### 3.2 Testando conexão a IPv4, IPv6 e nome de domínio

A fim de se validar a conexão TCP em suas diferentes formas (IPv4, IPv6 e nome de domínio), logou-se em duas máquinas distintas do CRC (Centro de Recursos Computacionais) da UFMG. Tais máquinas foram, a saber:

- **niger.grad**: desempenhando o papel de servidor;
- **guaxupe.grad**: desempenhando o papel de cliente;

A Figura 1 deixa bem uma conexão foi estabelecida entre as duas máquinas e mensagens foram trocadas nos três casos: IPv4, IPv6 e nome de domínio.



The image shows two terminal windows side-by-side. The left window is titled 'marlon@arctic: ~/ssh115x61' and shows a terminal session on the 'niger' machine. The prompt is 'niger:~/redes/tp1/mm-client-server-c-> ./servidor 8081'. The right window is also titled 'marlon@arctic: ~/ssh115x61' and shows a terminal session on the 'guaxupe' machine. It displays three commands being executed: './cliente niger.grad 8081 < in1.txt', './cliente 150.164.6.35 8081 < in1.txt', and './cliente 2001:12f0:601:a914:8958:d446:f069:da68 8081 < in2.txt'. Each command is followed by a series of numbers (1 through 8) and a final prompt, indicating successful communication.

Figura 1 : Servidor na esquerda e cliente na direita: exemplos de execução com testes da seção anterior(imagem ampliada em anexo) - UDP.

O comando utilizado pelo **servidor** para iniciar na porta **8081** foi:

```
./servidor 8081
```

Os comandos utilizados pelo **cliente** para se conectar ao servidor foram:

```
./cliente niger.grad 8081  
./cliente 150.164.6.35 8081  
./cliente 2001:12f0:601:a914:48f0:206d:97d2:935b 8081
```

## 4 CONCLUSÃO

Neste trabalho foram implementadas duas aplicações: cliente e servidor, as quais se comunicam via sockets da biblioteca GNU. A linguagem de programação utilizada foi C++, e ambos programas possuem uma interface de linha de comando (CLI).

O trabalho permitiu adquirir conhecimento valioso da biblioteca de sockets em C, além de pincelar em termos práticos a comunicação entre camadas existente na arquitetura de redes. O enfoque deste trabalho no protocolo UDP permitiu adquirir conhecimentos valiosos no funcionamento deste protocolo, em especial na construção de confirmação em cima da camada de transporte, na camada de aplicação.

## 5 REFERÊNCIAS

[1]: <http://www.cplusplus.com/reference/set/multiset/> visualizado em maio de 2016;

[2]: [http://www.gnu.org/software/libc/manual/html\\_node/Sockets.html](http://www.gnu.org/software/libc/manual/html_node/Sockets.html) visualizado em maio de 2016;

[3] Installation, *Doxygen Manual*. Retirado de <https://www.stack.nl/~dimitri/doxygen/manual/install.html> em junho de 2016

[4]: “Stop and wait protocol”, ISI - University of Southern California retirado de [http://www.isi.edu/nsnam/DIRECTED\\_RESEARCH/DR\\_HYUNAH/D-Research/stop-n-wait.html](http://www.isi.edu/nsnam/DIRECTED_RESEARCH/DR_HYUNAH/D-Research/stop-n-wait.html) em junho de 2016;