# Trabalho Computacional I
# Protocolo Olmpico
# DCC023 - Redes de Computadores

Paulo Cirino Ribeiro Neto

May 15, 2016

## 1 O Protocolo Olímpico

As Olimpíadas de 2016 serão no Rio de Janeiro - Brasil e segundo o professor Luiz Filipe M. Vieira o Comitê Olímpico Internacional (COI) precisa da nossa ajuda. O COI precisa coletar os dados das avaliações dos atletas e classifica-los remotamente. Baseado nessa ideia, nos foi incumbida a tarde de criar um programa cliente que envia o desempenhos dos atletas para um servidor e um programa servidor que retorna ao cliente a classificação geral do atleta dados os resultados parciais já recebidos de seus adversários naquela conexão.

## 2 Problemas tratados

Assim modelamos no domínio do problema, não da conexão, em 2 questões a serem resolvidas :

1. Leitura dos tempos;
2. Rankeamentos dos atletas.

Dessa forma a primeira solução é feita de forma a ler um texto com o tempo, e converter esse texto para um valor inteiro em milissegundos no próprio cliente onde essa informação é obtida, para que esse valor possa ser passado ao servidor;

Por outro lado o rankeamento é feito à partir do servidor e se baseia na ideia de "rankear" o valor em milissegundos, para que seja possível retornar a classificação do atleta.

Outro problema tratados é o paradigma Cliente/Servidor, que foi resolvido utilizando a ideia que o cliente lê e faz um pré-processamento na informação e ela é enviada ao servidor para armazenamento e pós-processamento, que nesse caso é a manutenção de um ranking atualizado.

# 3 Algorítimos e TADs criados

Para a solução dos problemas enumerados na seção anterior, criamos 4 conjuntos de arquivos:

1. **Server.c**
2. **Client.c**
3. **ranking.c/ ranking.h**
4. **readinTime.c / readingTime.h**

Esses conjuntos funcionam em pares, o *Client.c* funciona em conjunto com os arquivos *readingTime*, e o *Server* funciona em conjunto com os arquivos *ranking*.

## 3.1 readingTime

Esse conjunto de arquivos foi criado para a definição de duas funções :

- **int** $getTimeInMiliSeconds(\textbf{char}*measurement, \textbf{int}lengthMeasurement, \textbf{int}timeInUnit)$
- **int** $readLine()$

A primeira função, *getTimeInMiliSeconds*, recebe 3 parâmetros que são uma *string* com a unidade tempo da medida, o tamanho dessa *string*, e a quantidade de tempo nesse unidade em específica. Essa função foi criada utilizando a propriedade do **void switch ()** para dado um par $(tempo, unidade)$, converter de forma genérica para a unidade de milisegundos.

Essa função é utilizada com a outra função definida nesses arquivos que é *readLine*, que lê da entrada padrão pares de $(tempo, unidade)$ de forma sucessiva e converte todos os pares lidos retornando o seu total em milissegundos.

## 3.2 ranking

O grupo de arquivos *ranking* defina a estrutura utilizada para a criação e manutenção das posições dos tempos dos atletas.

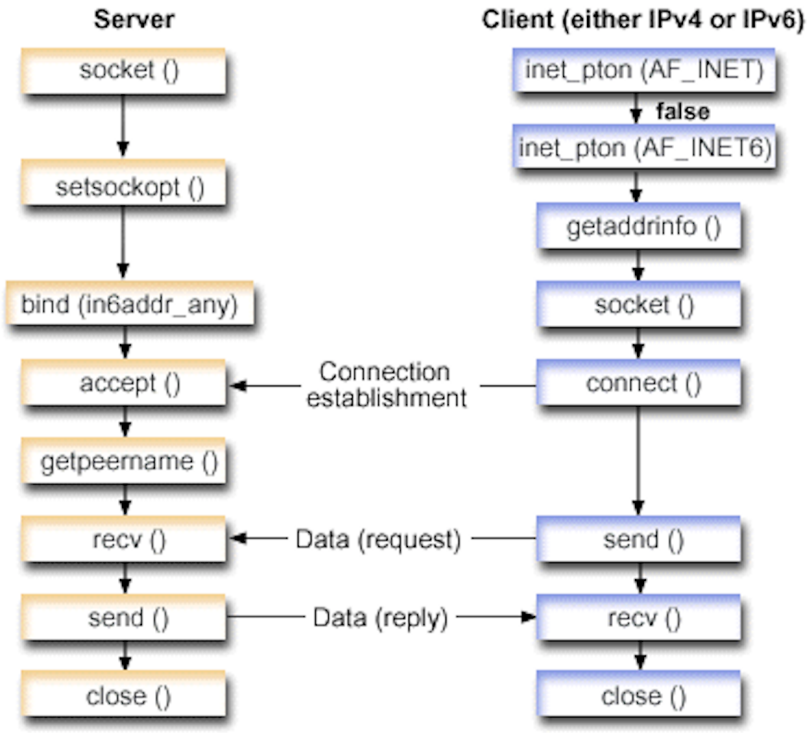
A definição da estrutura utilizada para isso foi criada com a ideia de um *priority queue* à partir da implementação de uma lista duplamente encadeada encabeçada.

Para isso forma implementas as estruturas da célula do *queue* que é um apontador que aponta para um local de memória onde estão armazenados o valor de um determinado atleta em milissegundos na forma de um inteiro e 2 apontadores para os tempos dos atletas nas posições anterior e posterior.

A estrutura do *queue* em sí é formada por 2 apontadores para as células de inicio e fim dele, e 3 funções para alocação da memória, inserção, e liberação de memória. Como só precisamos saber a posição na qual um atleta foi inserido não é necessária funções para localização nem pesquisa no *queue*.

# 4 Tratamento IPv4 e IPv6

Para o tratamento da criação de um programa *Cliente Servidor* que seja agnóstico quanto a *IPv4* e *IPv6*, foi utilizado como base na implementação exemplo da IBM presente nas referências, que se baseia no diagrama abaixo :




A ideia básica do funcionamento desse programa é a criação de um servidor **IPv6** que funciona exatamente como o exemplo apresentado em sala, so que nesse caso setando as variáveis para o protocolo IPv6, mesmo quando o servidor está em um rede que apenas suporta IPv4 isso funciona porque a API IPv6 faz o mapeamento entre os protocolos e o endereço de *loopback* que é **in6addr_any** que é dado no servidor já considera essa possibilidade.

Já do lado do cliente fazemos de forma diferente, utilizamos além da tradicional estrutura **sockaddr_in6** utilizamos também uma estrutura **addrinfo** e mais um outra variável que é um ponteiro para uma estrutura desse tipo.

A ideia é ler o endereço informado e testar à qual protocolo ele pertence para que as variáveis da estrutura **sockaddr_in6** possam ser setadas de acordo com o devido protocolo, lembrando que isso só é possível porque o API IPv6 faz o mapeamento de forma automática.

Utilizamos então a função **getaddrinfo** para pegar as informações do servidor e à salvamos no ponteiro do tipo **addrinfo**. Essas informações são então utilizadas para inicializar o socket da conexão que por fim é utilizado para abertura da conexão.

Toda a troca de mensagens são feitas de forma idênticas ao mostrado em sala.

# 5 Makefile

O *Makefile* foi feito de forma a compilar os arquivos por meio do **GCC** sem a utilização de arquivos da extensão *object file* porque cada *header* é utilizado em apenas 1 programa.

Para compilação do servidor e do cliente apenas com o **make** foi-se utilizado o argumento **"all : Client Server"**.

# 6 Testes

Foram feitos diversos testes para a validação do código apresentado, a única dificuldade apresentada foi em automatizar os testes com pipes, uma vez que por algum motivo eles geravam *segmentation faults* .

Com essa exceção, todo o resto ocorreu como o esperado . Seguem abaixo algumas fotos de alguns testes.

Teste de cliente utilizando **IPv4** como IP do servidor, e demonstração do caso onde todos os valores são repetidos :

```
[paulocirino Tp_01]$ ./Client 127.0.0.1 2345
Cliente :> 1s
Servidor :> 1

Cliente :> 1s
Servidor :> 2

Cliente :> 1s
Servidor :> 3

Cliente :> 1s
Servidor :> 4

Cliente :> -1

[paulocirino Tp_01]$
```

```
[paulocirino Tp_01]$ ./Server 2345
Waiting for Connection
Connection Accepted
Just Started a new connection with :
Ip Adress : ::ffff:127.0.0.1
Port : 50809

Echoing back Rank - 1
Echoing back Rank - 2
Echoing back Rank - 3
Echoing back Rank - 4
Echoing back Rank - -1
Waiting for Connection
```

Caso de teste simples que testa sequência de valores na mesma unidade, esse também é um teste de **IPv6**, onde o IP informado servidor foi :: 1 :

```
[paulocirino Tp_01]$ ./Client ::1 2121
Cliente :> 2s
Servidor :> 1

Cliente :> 3s
Servidor :> 2

Cliente :> 1s
Servidor :> 1

Cliente :> 5s
Servidor :> 4

Cliente :> 5s 10ms
Servidor :> 5

Cliente :> 5s 1ms
Servidor :> 5

Cliente :> 5s 2ms
Servidor :> 6

Cliente :> -1

[paulocirino Tp_01]$
```

```
[paulocirino Tp_01]$ ./Server 2121
Waiting for Connection
Connection Accepted
Just Started a new connection with :
Ip Adress : ::1
Port : 50905

Echoing back Rank - 1
Echoing back Rank - 2
Echoing back Rank - 1
Echoing back Rank - 4
Echoing back Rank - 5
Echoing back Rank - 5
Echoing back Rank - 6
Echoing back Rank - -1
Waiting for Connection
```

Caso de teste com o nome do computador na rede e mais de uma conexão simultânea. Esse também é um teste mais complicado e que mostra que valores inseridos em seções diferentes influência a outra:

```
[paulocirino Tp_01]$ ./Client localhost
Wrong Number of Arguments.
Correct usde would be
 ./Server [ip/nome] [Port]

[paulocirino Tp_01]$ ./Client localhost 2345
Cliente :> 10h 10m 10s 10ms
Servidor :> 1

Cliente :> 11h 10m 10s 10ms
Servidor :> 2

Cliente :> 11h 11m 10s 10ms
Servidor :> 3

Cliente :> 11h 11m 11s 10ms
Servidor :> 4

Cliente :> 11h 11m 11s 11ms
Servidor :> 5

Cliente :> 10h 10m 10s 9ms
Servidor :> 1

Cliente :> 10h 10m 9s 10ms
Servidor :> 1

Cliente :> 10h 9m 10s 10ms
Servidor :> 1

Cliente :> 9h 10m 10s 10ms
Servidor :> 1

Cliente :> 12h
Servidor :> 10

Cliente :> 9ms
Servidor :> 1

Cliente :> -1

[paulocirino Tp_01]$
```

```
Last login: Sun May 15 17:58:24 on ttys001

[paulocirino Tp_01]$ ./Client ::1 2345
Cliente :> 8h
Servidor :> 1

Cliente :> 12h
Servidor :> 2

Cliente :> -1

[paulocirino Tp_01]$
```

Tp_01 — Server 2345 — 42×28

```
Connection Accepted
Just Started a new connection with :
Ip Adress : ::1
Port : 52161

Echoing back Rank - 1
Echoing back Rank - 2
Echoing back Rank - 3
Echoing back Rank - 4
Echoing back Rank - 5
Echoing back Rank - 1
Echoing back Rank - 1
Echoing back Rank - 1
Echoing back Rank - 1
Echoing back Rank - 10
Echoing back Rank - 1
Echoing back Rank - -1
Waiting for Connection
Connection Accepted
Just Started a new connection with :
Ip Adress : ::1
Port : 52162

Echoing back Rank - 1
Echoing back Rank - 2
Echoing back Rank - -1
Waiting for Connection
```

# 7 Conclusões

Foi concluído que os testes validaram os requisitos apresentados na especificação do trabalho. Além disso, também foi possível concluir que a estrutura IPv6 é um tanto quanto mais genérica que a estrutura IPv4, isso porque ela foi construída para ser capaz de funcionar com ambas especificações de protocolo.

No que diz respeito à decisões de projeto que não estavam especificadas na definição do trabalho e nem no fórum da disciplinas podemos ressaltar a metodologia utilizada para tratar IPv4 e IPv6, onde foi implementada a forma já descrita na seção 4. Podemos também apontar a decisão de qual tipo de tratamento e processamento seria feito em servidor ou cliente. E por fim os testes que forma utilizados para validar o programa.

# 8 Referências

Foram utilizadas os seguintes links e sites :


- $https://www.ibm.com/support/knowledgecenter/ssw_i5_54/rzab6/xacceptboth.htm$
- $https://www.ibm.com/support/knowledgecenter/ssw_i5_54/rzab6/xip6client.htm$
- $http://linux.die.net/man/$
- $http://stackoverflow.com/$
- $http://pubs.opengroup.org/onlinepubs/000095399/basedefs/netinet/in.h.html$
- $http://man7.org/linux/man-pages/man3/$


Além desses materiais foi utilizado também as informações contidas em **man** e nas notas de aula da Bruna Peres apresentadas em sala.