

UNIVERSIDADE FEDERAL DE MINAS GERAIS

Redes de Computadores

TP2: Protocolo Olímpico UDP
Luiz Henrique Silva Lelis

08/06/2016

1 Visão Geral

No trabalho proposto foi implementado um sistema atendendo a todos os requisitos básicos dispostos na documentação. Foi requisitado que se implementasse um sistema cliente/servidor no qual o cliente entrasse com o tempo do atleta e o servidor retornasse a sua respectiva posição. O sistema seria utilizado como um protocolo olímpico durante as Olimpíadas de 2016. O código foi escrito em linguagem C e a biblioteca dos sockets foi usada para tornar a comunicação possível. Foi requisitado uma comunicação envolvendo a camada de Transporte especificamente o protocolo UDP (User Datagram Protocol).

Todo programa num Sistema Operacional que tem contato externo, por meio de rede, usa socket. Por esse motivo, o uso da biblioteca de sockets é indispensável. Por fim, os algoritmos de busca e ordenação implementados foram baseados no livro: *"Projeto de Algoritmos com Implementações em Pascal e c"*

2 Implementação

O código possui 5 arquivos principais:

- cliente.c - Recebe os parametros da linha de comando, e utiliza sockets para iniciar uma comunicação com o servidor
- servidor.c - Recebe os parametros da linha de comando, aceita a conexão com um cliente e chama as funções para fazer a ordenação e a busca
- funcoes.c - Todas as funções do programa utilizadas pelo cliente ou pelo servidor encontram-se nesse arquivo
- funcoes.h - Estruturas como o registro (tempo de cada atleta) e a tabela de tempos estão nesse arquivo. Além deles, os cabeçalhos das funções também se encontram nesse arquivo
- makefile - Arquivo responsável por compilar o código e gerar os executáveis do cliente e servidor.

2.1 Configuração do tempo dos atletas

As duas estruturas principais do programa - Registro(Record) e Tabela(Table) - foram utilizadas para armazenarem os tempos dos atletas e serem manipuladas durante a ordenação e busca dos tempos. O Registro(Record) possui somente a Chave(Key) que corresponde ao tempo do atleta em milissegundos. Já a Tabela(Table), possui um array de Registros (onde todos os tempos ficam armazenados) e um inteiro n que representa o tamanho da tabela.

Para resolver o problema da ordenação de tempos, foi usado o método da inserção (InsertionSort). Foi escolhido esse método pois é o melhor quando temos uma estrutura quase totalmente ordenada de entrada. No caso atual, a tabela está quase totalmente ordenada pois a cada inserção de um novo tempo, terá uma nova ordenação.

No servidor, o programa faz primeiramente a inserção do novo tempo, depois ordena a tabela e por último faz a busca na tabela para saber em qual índice o

respectivo tempo se encontra. A busca está sendo realizada através do algoritmo de busca binária (BinarySearch).

Para resolver o problema do IPv4 e IPv6 no servidor, toda a conexão foi tratada como IPv6. Esse modelo foi escolhido pois o socket IPv6 tem compatibilidade com os nós IPv4. Foi atribuído `in6addr_any` ao endereço do servidor pois ele permite (por padrão) estabelecer conexões de qualquer cliente IPv4 e IPv6 que especificar a porta igual ao segundo argumento(`argv[1]`).

```
char *MsgRead = malloc (MAXBUF*sizeof(char));
char *MsgWrite = malloc (MAXBUF*sizeof(char));
char Ack[2];
struct sockaddr_in6 ServerAddr, ClientAddr;
int UdpSocket = -1, Port;
socklen_t MsgCheck;
int ClientAddrLen = sizeof(ClientAddr);

Port = atoi(argv[1]);
int Position, i, TimeMiliseconds;
Record AthleteTime;

UdpSocket = socket(AF_INET6, SOCK_DGRAM, 0);

memset(&ServerAddr, 0, sizeof(ServerAddr));
ServerAddr.sin6_family = AF_INET6;
ServerAddr.sin6_port = htons(Port);
ServerAddr.sin6_addr = in6addr_any;

bind(UdpSocket, (struct sockaddr *)&ServerAddr, sizeof(
    ServerAddr));
```

No caso do cliente, foi preciso inicialmente verificar se o endereço seria IPv4 ou IPv6. Para isso, foi usada a função `inet_pton` que converte o endereço de texto para a forma binária. Além disso, a função retorna um inteiro. Se for igual a 1, o endereço a ser convertido é um endereço válido. Dessa forma, se passar o `AF_INET` como parâmetro da função `inet_pton`, e o valor retornado for 1, significa que se trata de um endereço IPv4. Por outro lado, quando se passa `AF_INET6`, e o inteiro retornado for igual a 1, se trata de um endereço IPv6. Feito isso, usa-se a função `socket()` para retornar um descritor de sockets, e depois `sendto()` e `recvfrom()` para estabelecer conexões com o servidor.

```
memset(&Hints, 0x00, sizeof(Hints));
Hints.ai_flags = AI_V4MAPPED;
Hints.ai_family = AF_UNSPEC;
Hints.ai_socktype = SOCK_DGRAM;

Rc = inet_pton(AF_INET, argv[1], &ServerAddr);
if (Rc == 1){
    Hints.ai_family = AF_INET;
    Hints.ai_flags = AI_NUMERICHOST;
}
else{
    Rc = inet_pton(AF_INET6, argv[1], &ServerAddr);
    if (Rc == 1){
        Hints.ai_family = AF_INET6;
        Hints.ai_flags = AI_NUMERICHOST;
    }
}

getaddrinfo(argv[1], argv[2], &Hints, &Res);
```

```

    UdpSocket = socket(Res->ai_family, Res->ai_socktype, Res->
        ai_protocol);

```

Outro problema resolvido pelo sistema foi o tratamento do tempo de entrada. O tempo começa a ser tratado no cliente e termina seu tratamento no servidor. No cliente, a leitura da entrada é feita de caractere por caractere ignorando o espaço, tab e \r. A leitura só termina quando aparece um \n. Com isso, a string enviada para o servidor não tem espaços. A função que realiza essa leitura e edição é a ReadAndEditStr() da biblioteca funcoes.h. Com isso, no servidor, a mensagem é convertida de string para inteiro pela função ChangeToMilisec(). A função percorre a string atribuindo o valor numérico a uma string auxiliar. Quando chega o caractere descrevendo o tempo (h/m/s/ms), o valor numérico é multiplicado pela quantidade correspondente em milissegundos. Por exemplo, se a entrada do programa é 10h 3s, será passado para o servidor como 10h3s. No servidor, dentro da função ChangeToMilisec(), a string auxiliar receberá primeiro 10, esse valor será multiplicado por 3600000 ao receber o h (1h = 3600000ms). Feito isso, a auxiliar receberá agora 3, e ao percorrer o s, esse valor será multiplicado por 1000 (1s = 1000ms). Por fim, retorna-se a soma dos valores em milissegundos, no caso retornaria 3600000+1000.

O tempo em milissegundos é inserido na tabela, ocorre uma ordenação por InsertionSort e por fim, é feita uma busca binária. O índice do tempo buscado na tabela corresponde à colocação do atleta. Esse índice é convertido novamente para uma string (sprintf) e retornado para o cliente para ser impresso na tela posteriormente.

2.2 Configuração da comunicação UDP

Para tornar a comunicação possível no cliente, foi necessário primeiro mudar o tipo de socket para o formato datagrama (Hints.ai_socktype = SOCK_DGRAM). Além disso, foram utilizadas as funções getaddrinfo, socket, setsockopt, sendto e recvfrom(). A função getaddrinfo foi usada para pegar a informação do endereço para o servidor, a socket() retorna um descritor de sockets que representa um endpoint. Outra função essencial foi a setsockopt(), que permitiu definir um tempo limite (100ms) para o socket retornar a confirmação da mensagem. Sem ela, o código ficaria travado em recvfrom() caso o ACK não chegasse. Por fim, sendto e recvfrom foram usadas para enviar e receber mensagens do servidor.

O cliente precisa enviar o tempo do atleta para o servidor (primeiro sendto), e receber a confirmação dele (primeiro recvfrom). A variável MsgLen é utilizada para verificar o tamanho da mensagem que chega do servidor. Caso ela seja menor que zero, houve um erro na comunicação. Quando o erro ocorre, é necessário reenviar a mensagem (segundo sendto). Apesar disso, esse reenvio deve ocorrer depois de um tempo. Esse tempo de aguardo é necessário pois deve-se esperar a mensagem chegar ao servidor e retornar a confirmação para o cliente.

```

    sendto(UdpSocket, MsgWrite, MAXBUF, 0, Res->ai_addr, Res->
        ai_addrlen);

    setsockopt(UdpSocket, SOL_SOCKET, SO_RCVTIMEO, &TimeToAnswer,
        sizeof(TimeToAnswer));

```

```

RecMsgLen = recvfrom(UdpSocket, &Ack, 2, 0, Res->ai_addr, &
Res->ai_addrlen);

if(RecMsgLen < 0){
    Delay(0.1);
    sendto(UdpSocket, MsgWrite, MAXBUF, 0, Res->ai_addr, Res
->ai_addrlen);
}

```

Esse tempo de espera foi calculado experimentalmente. O código foi rodado várias vezes medindo-se o tempo entre o sendto (do tempo do atleta) e o recvfrom (confirmação). Pegando o pior caso, foi acrescido 33,3% do seu valor resultando em 0,1ms.

Para cada mensagem de tempo do atleta, existe um cabeçalho(identificador). O cabeçalho foi utilizado para verificar se a confirmação que chega ao cliente corresponde à mensagem que ele havia enviado anteriormente. A variável que armazena o cabeçalho foi chamada de NewHead, e ela corresponde a uma letra minúscula do alfabeto (97 ao 123 da tabela Asii). A cada nova mensagem, é somado um ao cabeçalho, e ao chegar no 123 (letra z), a próxima mensagem receberá 97 (letra a) reiniciando o ciclo. O cabeçalho é adicionado na função ReadAndEditStr().

A próxima verificação que ocorre no cliente é sobre a confirmação da mensagem. Verificando o cabeçalho e o ACK, é possível observar se a confirmação é de uma mensagem antiga ou da mensagem atual. Caso seja uma mensagem antiga, o cliente só recebe a respectiva posição. Por outro lado, se for uma confirmação do tempo do atual atleta, o cliente recebe e imprime a respectiva posição.

```

if ((Ack[0] != NewHead)&&(Ack[1] == 6))
    WrongMsg = 1;

else if ((Ack[0] == NewHead)&&(Ack[1] == 6))
    WrongMsg = 0;

if(WrongMsg == 0){
    recvfrom(UdpSocket, MsgRead, MAXBUF, 0, Res->ai_addr, &
Res->ai_addrlen);
    if(atoi(MsgRead)>=0)
        printf("Chegou do serv -> %s\n", MsgRead);
}else{
    recvfrom(UdpSocket, MsgRead, MAXBUF, 0, Res->ai_addr, &
Res->ai_addrlen);
}

```

Para a configuração do servidor, também foi necessário mudar o tipo de socket para o formato datagrama (socket(AF_INET6, SOCK_DGRAM, 0)). Além disso, foram utilizadas as funções bind, listen, sendto e recvfrom. A função bind() foi usada para atribuir um nome ao socket. Esse procedimento é necessário pois é ele que atribui um endereço local para que um socket possa receber conexões. Por outro lado, o listen() escuta a rede permitindo que o servidor aceite as proximas conexoes dos clientes. A fila de solicitacao de conexao tem 5 como tamanho maximo. Por fim, sendto e recvfrom foram usadas para enviar e receber mensagens do cliente. O código contém um único recvfrom (para receber o tempo do atleta) e dois sendto (um para enviar a confirmação e o outro para enviar a posição do atleta).

3 Compilação & Execução

Para compilar basta usar o comando:

```
|| make
```

Para executar deve-se seguir as instruções do enunciado:

```
|| ./servidor [porta]
```

```
|| ./cliente [ip/nome] [porta]
```

Inicia-se a comunicação cliente/servidor.

4 Resultados

Primeiramente, foi feito o teste de comunicação, para verificar se o programa estava aceitando comunicações de endereços IPv4 e IPv6. Para isso, foi adicionado ao cliente o seguinte código:

```
|| if(Res->ai_family == AF_INET) {  
||     printf("%s e um endereço IPv4\n",argv[1]);  
|| } else if (Res->ai_family == AF_INET6) {  
||     printf("%s e um endereço IPv6\n",argv[1]);  
|| } else {  
||     printf("%s e um formato desconhecido %d\n",argv[1],Res  
||         ->ai_family);  
|| }
```

Os resultados obtidos seguem abaixo:



```
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFGM/Quinto/Redes/TP1$ ./cliente 127.0.0.1 3005  
127.0.0.1 e um endereço IPv4  
^C  
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFGM/Quinto/Redes/TP1$ ./cliente ::1 3005  
::1 e um endereço IPv6  
^C  
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFGM/Quinto/Redes/TP1$ ./cliente localhost 3005  
localhost e um endereço IPv4  
^C  
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFGM/Quinto/Redes/TP1$
```

Figure 1: Teste com entrada de endereços IPv6, IPv4 ou nome

Além desse teste, foi testado a conexão com o servidor. O teste da comunicação cliente/servidor e das particularidades do Protocolo Olímpico seguem abaixo:

Ao encerrar o cliente (entrada de um valor negativo), o servidor deve continuar seu funcionamento normalmente, aguardando pela conexão de um novo cliente. Esse novo cliente deve ter uma nova tabela de tempos e começar as configurações do início. O teste para verificar o funcionamento nessas condições segue abaixo(no teste, o serviço do servidor já está rodando na porta 3005):

```

CLIENTE
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFMG/Quinto/Redes/TP1$ make
gcc servidor.c funcoes.c -o servidor
gcc cliente.c funcoes.c -o cliente
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFMG/Quinto/Redes/TP1$ ./cliente 127.0.0.1 3005
1h 3s
1
1h 3m 4s 444ms
2
0h 33m 44s 333ms
1
289ms
1
2h
5
465m
6
334ms 44s 33m 0h
3
288ms
1
1h 2m 33m 346ms
7
22h
10
-8
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFMG/Quinto/Redes/TP1$ █

SERVIDOR
luizhlelis@luizlelisPC:~$ cd /media/DADOS_750GB/UFMG/Quinto/Redes/TP1/
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFMG/Quinto/Redes/TP1$ ./servidor 3005
█

```

Figure 2: Teste comunicação cliente/servidor

```

CLIENTE
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFMG/Quinto/Redes/TP1$ ./cliente 127.0.0.1 3005
2h 33m
1
1
4h 23s 456ms
2
4h 22s
2
4h 33m 22s 300ms
4
1h 20m 57s 500ms
1
1h
1
-2
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFMG/Quinto/Redes/TP1$ ./cliente 127.0.0.1 3005
88h
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFMG/Quinto/Redes/TP1$ ./cliente 127.0.0.1 3005
2h 33m
1
1
█

SERVIDOR
luizhlelis@luizlelisPC:~$ cd /media/DADOS_750GB/UFMG/Quinto/Redes/TP1/
luizhlelis@luizlelisPC:/media/DADOS_750GB/UFMG/Quinto/Redes/TP1$ ./servidor 3005
█

```

Figure 3: Teste - encerra a comunicação com um cliente e inicia nova conexão

5 Conclusão

Analisando o exposto acima, observa-se resultados bastante satisfatórios. O programa aceita comunicação de endereços IPv4, IPv6 ou de hostNames. Além disso, os tempos inseridos são tratados e armazenados e a ordenação e busca são realizados em um tempo satisfatório. Por fim, a posição do respectivo atleta é retornada para o cliente. Sendo assim, os objetivos propostos foram resolvidos possibilitando que o problema do Protocolo Olímpico fosse concluído com êxito.

6 Referências

IBM Knowledge Center – **Example: Accepting connections from both IPv6 and IPv4 clients**. Disponível em: <https://www.ibm.com/support/knowledgecenter/ssw_i5_54/rzab6/xacceptboth.htm?lang=pt-br>. Acesso em 8 de maio de 2016.

ZIVIANI, Nivio. **Projeto de algoritmos com implementações em Pascal e C. 3.** ed. Belo Horizonte: Cengage, 2004.