UNIVERSIDADE FEDERAL DE MINAS GERAIS

Redes de Computadores

TP3: Protocolo Olímpico - Múltiplas Conexões

Luiz Henrique Silva Lelis

02/07/2016

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 TCP (Transmission Control Protocol). Além disso, o servidor deve tratar várias conexões de uma vez, informando a classificação geral do atleta dados os valores já recebidos durante todas as conexões.

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.

Além do tratamento das múltiplas conexões no servidor e cliente (main), o programa possui 5 funções auxiliares e duas estruturas de dados:

- Record - Estrutura que possui um único item (a chave). A chave corresponde ao tempo do atleta em ms.
- Table - Estrutura responsável por armazenar os tempos. Possui dois itens, um vetor(de Records) onde os tempos ficam armazenados, e o tamanho da tabela.
- InsertTable - Função utilizada para inserir um tempo na tabela.
- BinarySearch - Função responsável por realizar a pesquisa binária na tabela.

- InsertionSort - Função responsável por fazer a ordenação pelo método da inserção na tabela.
- ChangeToMilisec - Função responsável por converter o tempo de entrada (string) para milissegundos (int).
- ReadAndEditStr - Função responsável por fazer a leitura do dispositivo de entrada (teclado) ignorando o espaço, tab e \r.

2.1 Configuração do tempo dos atletas

As duas estruturas principais do programa - Registro(Record) e Tabela(Table) - foram utilizadas para armazenar o tempo dos atletas e para 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 atribuido 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]).

```c
#define MAXBUF 100

int main(int argc, char **argv){

        char *MsgRead = malloc (MAXBUF*sizeof(char));
        char *MsgWrite = malloc (MAXBUF*sizeof(char));
        int  *ClientSocket = malloc (30*sizeof(int));
        struct sockaddr_in6 ServerAddr, ClientAddr;
        int Addrlen = sizeof(ClientAddr);
        int Sd =-1, Port, NewSocket, MaxClients = 30, Activity, ValRead, MaxSd, SdClient;

        // Conjunto de descritores de um socket
        fd_set ReadFds;

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

        Sd = socket(AF_INET6, SOCK_STREAM, 0);

        /* Foi atribuido in6addr_any ao endereço pois ele permite (por padrao) estabelecer conexoes
        de qualquer cliente IPv4 e IPv6 que especificar a porta igual ao segundo argumento(argv[2])*/

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

        bind(Sd,(struct sockaddr *)&ServerAddr,sizeof(ServerAddr));

        listen(Sd, 5); // Permite que o servidor aceite conexoes de entrada do cliente

        Table *AthletesPos = malloc(sizeof(Table));
```

Figure 1: Aceitando conexões de clientes IPv6 e IPv4

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 funcao socket() para retornar um descritor de sockets, e depois connect() para estabelecer uma conexão com o servidor.

```
15      int    Sd=-1, Rc;
16      struct in6_addr ServerAddr;
17      struct addrinfo Hints, *Res=NULL;
18      char *MsgWrite = malloc (MAXBUF*sizeof(char));
19      char *MsgRead = malloc (MAXBUF*sizeof(char));
20
21      memset(&Hints, 0x00, sizeof(Hints));
22      Hints.ai_flags = AI_NUMERICSERV;
23      Hints.ai_family = AF_UNSPEC;
24      Hints.ai_socktype = SOCK_STREAM;
25
26      Rc = inet_pton(AF_INET, argv[1], &ServerAddr);
27      if (Rc == 1){ // endereco valido de IPv4?
28         Hints.ai_family = AF_INET;
29         Hints.ai_flags = AI_NUMERICHOST;
30      }
31      else{
32         Rc = inet_pton(AF_INET6, argv[1], &ServerAddr);
33         if (Rc == 1){ // endereco valido de IPv6?
34            Hints.ai_family = AF_INET6;
35            Hints.ai_flags = AI_NUMERICHOST;
36         }
37      }
38
39      getaddrinfo(argv[1], argv[2], &Hints, &Res); //Pega a informacao do endereco para o servidor
40
41      // A funcao socket() retorna um descritor de sockets que representa um endpoint.
42      Sd = socket(Res->ai_family, Res->ai_socktype, Res->ai_protocol);
43
44      // Usa a função connect () para estabelecer uma conexão com o servidor.
45      connect(Sd, Res->ai_addr, Res->ai_addrlen);
46
47      do{
48          ReadAndEditStr(MsgWrite, MAXBUF);
49          write(Sd, MsgWrite, strlen(MsgWrite)+1);
50          read(Sd, MsgRead, MAXBUF);
51          if(atoi(MsgRead)>=0)
52              printf("%s\n", MsgRead);
53      } while(atoi(MsgRead)>=0);
```

Figure 2: Configuração para clientes IPv6 ou IPv4

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 so 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 funcao percorre a string atribuindo o valor numerico 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 feito 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 para múltiplas conexões

Para tornar possível a múltipla conexão, foi preciso fazer algumas mudanças no servidor. Foi escolhido select() para se tratar as múltiplas conexões. Esse método foi escolhido devido à implementação ser mais fácil, pois o servidor precisa de apenas um único processo para lidar com todas as solicitações.

Primeiramente, a tabela foi criada fora do loop para que cada conexão pudesse inserir um novo tempo na mesma estrutura de dados. A inserção na tabela, a pesquisa e ordenação foram manipuladas dentro do loop while(1).

O processo é bloqueado até que algo (entrada de dados) aconteça em um descritor de arquivo (ou seja, em um socket). Foi criado um conjunto de descritores de socket (ReadFds) para monitorar esse tipo de atividade. O método select() recebe uma lista de sockets para monitorar. Quando tiver uma entrada de dados, a função select modifica ReadFds, que agora armazenará os sockets que estão prontos para serem lidos.

```
while(1){

    FD_ZERO(&ReadFds);

    FD_SET(Sd, &ReadFds);
    MaxSd = Sd;

    for (i=0; i < MaxClients; i++){

        SdClient = ClientSocket[i];

        if(SdClient > 0)
            FD_SET(SdClient, &ReadFds);

        if(SdClient > MaxSd)
            MaxSd = SdClient;
    }

    Activity = select( MaxSd+1, &ReadFds, NULL, NULL, NULL);
```

Se algo acontecer no socket principal, significa que uma conexão está chegando. Caso contrario, é alguma operação de entrada e saída de dados em outro socket, assim sendo, é necessário percorrer o vetor de sockets (ClientSocket). Dentro dessa segunda condição, é onde se faz a leitura dos dados provenientes do cliente (read(SdClient, MsgRead, MAXBUF)). O valor que chega é convertido para inteiro em milissegundos. Caso o valor lido na função read seja igual a zero, significa que a mensagem não tem nada(informa e fecha). Se o tempo for menor que zero, significa que um cliente está se desconectando, nesse caso é necessário retornar o valor negativo para o cliente, depois informar e fechar a conexão. Por fim, quando não acontecer nenhum dos casos acima, o procedimento é retornar para o cliente a posição do atleta que fez aquele tempo.

```
    if (FD_ISSET(Sd, &ReadFds)){
      NewSocket = accept(Sd,(struct sockaddr*)&ClientAddr, &
          Addrlen);

      for (i=0; i < MaxClients; i++){
        if ( ClientSocket[i] == 0 ){
          ClientSocket[i] = NewSocket;
          break;
```

```
        }
      }
    }

    for(i = 0; i < MaxClients; i++){
      SdClient = ClientSocket[i];

      if(FD_ISSET(SdClient, &ReadFds)){

        ValRead = read(SdClient, MsgRead, MAXBUF);
        TimeMiliseconds = ChangeToMilisec(MsgRead, strlen(
            MsgRead));

        if((ValRead == 0)){
          close(SdClient);
          ClientSocket[i] = 0;
        }
        else if(TimeMiliseconds<0){
          sprintf(MsgWrite,"%d", TimeMiliseconds);
          write(SdClient, MsgWrite, strlen(MsgWrite)+1);
          close(SdClient);
          ClientSocket[i] = 0;
          strcpy(MsgRead, "");
          strcpy(MsgWrite, "");
        }
        else{
          AthleteTime.Key = TimeMiliseconds;
          InsertTable(AthleteTime, AthletesPos);
          InsertionSort(AthletesPos->Item, AthletesPos->n);
          Position = BinarySearch(AthleteTime.Key, AthletesPos);
          sprintf(MsgWrite,"%d", Position);
          write(SdClient, MsgWrite, strlen(MsgWrite)+1);
        }

      }
    }
```

2.3 Limitações

- O protocolo foi desenvolvido para suportar no máximo 30 clientes.
- As mensagens trocadas entre cliente e servidor (tempo dos atletas e posição retornada) devem conter no máximo 100 caracteres.
- A tabela suporta no máximo 1000 tempos. Ou seja, o número de atletas para cada prova não deve ultrapassar 1000.

3 Compilação & Execução

Para compilar basta usar o comando:

```
make
```

Para executar deve-se seguir as intruções 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 endereco IPv4\n",argv[1]);
    } else if (Res->ai_family == AF_INET6) {
         printf("%s e um endereco IPv6\n",argv[1]);
    } else {
         printf("%s e um formato desconhecido %d\n",argv[1],Res
             ->ai_family);
    }
```

Os resultados obtidos seguem abaixo:

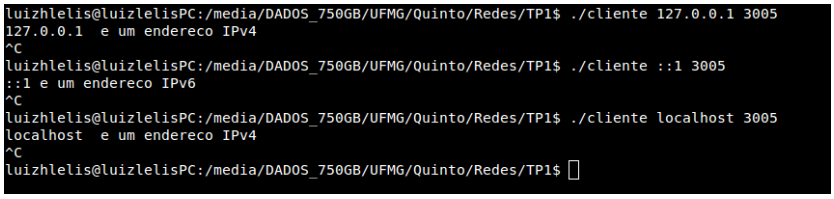

Figure 3: Teste com entrada de endereços IPv6, IPv4 ou nome

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

Figure 4: Teste comunicação cliente/servidor

O teste de múltiplas conexões em um único servidor segue abaixo. O servidor é o terminal do canto inferior direito. Os tempos foram sendo inseridos dos terminais superiores para posteriormente o inferior. Observe como ao terminar duas conexões, o servidor continua intacto, e um dos clientes permanece conectado.

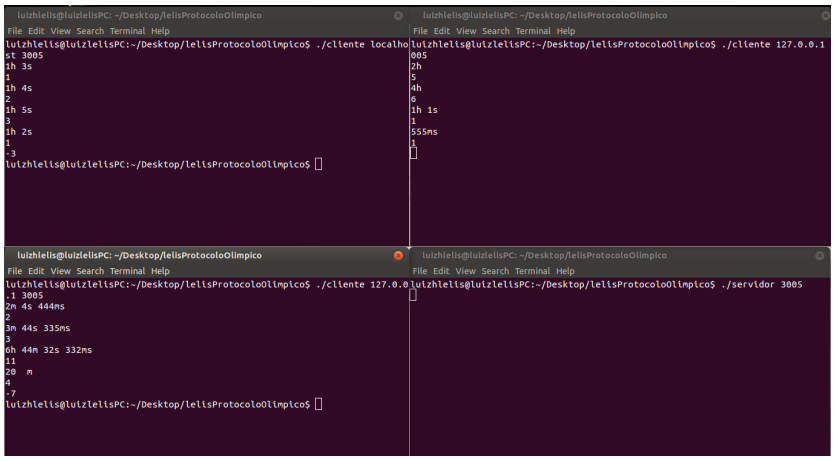

Figure 5: Teste - Múltiplas Conexões

Por fim, foi feito um último teste para verificar se as mensagens realmente estavam trafegando no formato do protocolo TCP. Para isso, foi utilizada uma ferramenta de captura de tráfego no rede (Wireshark). O resultado segue abaixo.

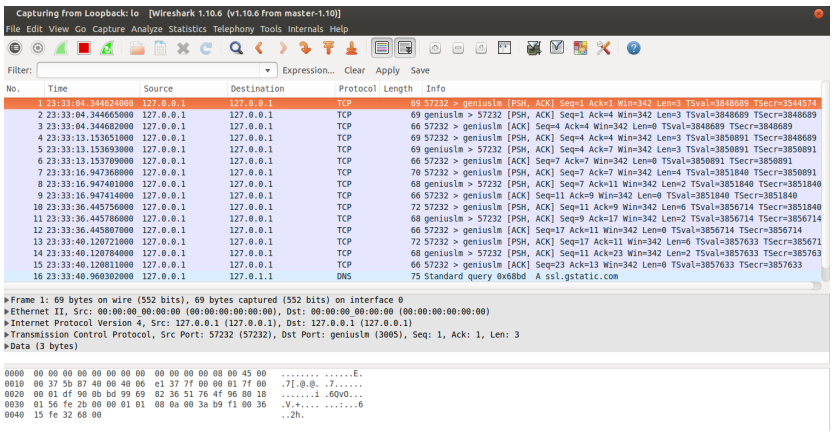

Figure 6: Teste - Verificando se a comunicação está implementada em TCP

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. Além disso, o programa está funcionando usando TCP como foi pedido. E também está resolvendo o problema das múltiplas conexões de clientes para um único servidor através do uso de select. 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 2 de julho de 2016.

IBM Knowledge Center – **select()– Wait for Events on Multiple Sockets**. Disponível em: `<https://www.ibm.com/support/knowledgecenter/ssw_i5_54/apis/sselect.htm>`. Acesso em 2 de julho de 2016.

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