Redes de Computadores
Trabalho Prático 1
Cliente e servidor em Sockets

André Taiar Marinho Oliveira

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

taiar@dcc.ufmg.br

Outubro de 2016

Resumo. *O problema tratado nesse trabalho é o de comunicação entre softwares cliente e servidor através de sockets em redes de computadores. Para isso, foi definido um protocolo de comunicação para um problema bem específico: coletar tempos de atletas das olimpíadas e retornar em ordem de chegada ou individualmente (quando passado o parâmetro da colocação do atleta). Foi feita a implementação dos dois programas (cliente e servidor) de forma modularizada e um Makefile para orientar a compilação do projeto.*

1. Implementação

O trabalho foi implementado em C++ para que eu pudesse me aproveitar melhor dos módulos de forma orientada a objetos. Foram implementados dois programas, cliente e servidor, de forma modularizada (cada módulo é uma classe basicamente).

Não foi necessária a implementação de nenhuma estrutura de dados mais elaborada. Nas versões iniciais do trabalho eu havia implementado uma lista para colecionar os tempos informados mas, quando eu precisei ordená-los isso se tornou inviável e eu decidi usar o $vector$ do C++.

Os módulos implementados bem como suas funções são descritos abaixo.

1.1. Token

A classe Token é responsável pela leitura da entrada fornecida pelo cliente. Basicamente ela separa a string fornecida por cada linha da entrada em sub-strings de forma que possamos fazer a leitura dos inputs fornecidos. O cabeçalho da classe é fornecido a seguir:

```
#ifndef __TOKEN_H
#define __TOKEN_H TOKEN

#define DELIMITER " "

#include <string>
#include <vector>

class Token {
public:
```

```
  Token(std::string);
  std::string getToken();
  std::string getNextToken();
  unsigned int total();
  int getCurrent();

private:
  std::string input;
  std::vector<std::string> tokens;
  int current;

  void tokenize();
};

#endif
```

1.2. Tempo

A classe Tempo é responsável por encapsular todo o processamento que precisamos fazer sobre o input de tempo no sistema. Suas principais funções são:

1. Instanciar o tempo à partir de uma string fornecida pela entrada;
2. Comparar se um tempo fornecido é maior ou menor que outro tempo.

O cabeçalho da classe é fornecido a seguir:

```
#ifndef __TEMPO_H
#define __TEMPO_H TEMPO

#include <iostream>
#include <vector>
#include <sstream>
#include <string>

#include <cctype>
#include <cstdlib>

#include "token.h"

using namespace std;

class Tempo {
public:
  int hours;
  int minutes;
  int seconds;
  int milisseconds;

  Tempo();
```

```
    Tempo(int, int, int, int);
    int compare(Tempo);
    Tempo* setFromString(char[]);
    string toString();
    void print();
    bool biggerThan(Tempo*);
    bool smallerOrEqualThan(Tempo*);

  private:
    void parseUnit(string);
    bool isNumber(char);
    void setTimeUnit(string, string);
    int toPseudoMs();
  };

  #endif
```

1.3. Servidor

A classe Servidor é responsável por encapsular toda a implementação do software servidor e a utilização de sockets para isso. Também contém a implementação do método $main$ da aplicação servidor. A intenção dela é abstrair as arestas trazidas pelas bibliotecas de sockets em C e implementar métodos que possam ser mais fáceis de usar.

Um bom exemplo de encapsulamento de um comportamento mais complicado seria o método a função $Servidor :: sendToClient$ que envia para o cliente já conectado uma string qualquer com uma quebra de linha ao final. Toda a comunicação que é feita dos servidor para o cliente neste trabalho utiliza esse método.

Uma outra vantagem em trabalhar com C++ neste projeto foi a maior facilidade de lidar com as strings ao invés de cadeias de caracteres.

O cabeçalho da classe Servidor é fornecido a seguir:

```
  #ifndef __SERVIDOR_H
  #define __SERVIDOR_H SERVIDOR

  #include <vector>
  #include <string>
  #include <cstdlib>
  #include <cstring>
  #include <cstdio>
  #include <unistd.h>

  #include <sys/socket.h>
  #include <arpa/inet.h>

  #include "tempo.h"

  struct sockaddr;
```

```
void logexit(const char*);
void fill(const struct sockaddr *addr, char *line);

class Servidor {
public:
  Servidor(int);
  void logexit(const char*);
  void fill(const struct sockaddr*, char*);
  void run();
  void pushTime(char[]);
  void getPosition(char[]);
  void dumpTimes();
  void shutdown();
  static int compare(const void*, const void*);
  void parse(char[]);
  void sendToClient(string);

private:
  int porta;
  int s;
  int r;
  std::vector<Tempo*> tempos;
  Tempo* returnThePosition(unsigned int);
};

#endif
```

1.4. Implementação do software cliente

Apesar de ter a extensão *.cpp* o código do cliente não é modularizado e está escrito em C mesmo. Eu basicamente utilizei o exemplo mostrado pelo monitor com algumas partes do exemplo publicado no Moodle e implementei o trecho de comunicação com o servidor para que tudo funcionasse bem. O maior foco do trabalho foi, com certeza, no software servidor.

1.5. Decisões de implementação

Uma decisão de implementação relevante fora o que já descrevi anteriormente é que ao colecionar os tempos no vetor eu faço a inserção deles já em ordem crescente. Dessa forma, quando recebo um novo tempo do cliente, eu vou comparando com os tempos já inseridos desde o início do vetor. Ao encontrar algum tempo que seja maior do que o que estou analisando, por exemplo, na posição N, eu insiro o novo tempo nessa posição e desloco todos os outos já existentes uma posição para frente no vetor.

Outra decisão, não tão baixo nível quanto a descrita anteriormente, é sobre as respostas do servidor retornadas ao cliente. Para toda mensagem recebida na servidor, o cliente espera uma resposta. Esta resposta pode ser para imprimir algo, continuar esperando o próximo comando ou terminar a execução do cliente. Esta implementação fica claro no software cliente neste trecho:

```
if(strcmp(line2, "Z\n") == 0)
  break;
else if(strcmp(line2, "O\n") == 0 || strcmp(line2, "D\n") == 0 || str
  continue;
else {
  printf("%s", line2);
  continue;
}
```

2. IPv4 e IPv6

Para fazer funcionar em ambos protocolos, eu me baseei no exemplo de código enviado no Moodle. Eu implementei no servidor o método $Servidor :: fill$ para que ele utilize corretamente o protocolo de acordo com o que o cliente tente se conectar.

3. Testes

Segue alguns testes efetuados mostrando várias conexões não-simultâneas de clientes ao servidor. A execução do servidor será omitida visto que ele não faz qualquer output durante o seu funcionamento.

```
~ src git:(master) ./cliente 127.0.0.1 1234
D 1h
D 61m
D 1m 1h 1ms
O
1h 0m 0s 0ms
0h 61m 0s 0ms
1h 1m 0s 1ms
D 2h
O
1h 0m 0s 0ms
0h 61m 0s 0ms
1h 1m 0s 1ms
2h 0m 0s 0ms
C 2
0h 61m 0s 0ms
C 7
C 0
C -1
C 1
1h 0m 0s 0ms
C 4
2h 0m 0s 0ms
O
1h 0m 0s 0ms
0h 61m 0s 0ms
1h 1m 0s 1ms
2h 0m 0s 0ms
```

```
Z
~  src git:(master) ./cliente 127.0.0.1 1234
O
1h 0m 0s 0ms
0h 61m 0s 0ms
1h 1m 0s 1ms
2h 0m 0s 0ms
D 22m
O
0h 22m 0s 0ms
1h 0m 0s 0ms
0h 61m 0s 0ms
1h 1m 0s 1ms
2h 0m 0s 0ms
Z
~  src git:(master)
```

4. Conclusão

Eu gostei de trabalhar na implementação deste trabalho. Já tinha programado em redes mas com linguagens mais alto nível (basicamente em Ruby) e ver como funciona a implementação de trocas de mensagem foi bem interessante.

Acredito que tenha acertado na modularização do meu trabalho visto que posso aproveitar a arquitetura produzida para os próximos trabalhos que seguirem a mesma linha de problema.