Engenharia de Telecomunicações
STD29006 – Sistemas Distribuídos

Prof. Emerson Ribeiro de Mello
mello@ifsc.edu.br

Laboratório 2: Sockets TCP e UDP: Java e C

13/09/2023

Esse laboratório tem por objetivo apresentar como desenvolver aplicações cliente e servidor em Java e em C usando a API *sockets*.

1 Programação com *sockets* na linguagem C

- API C criada em 1983 no 4.2 BSD UNIX, é padrão em todos S.O. No windows é a `winsock`
- Socket consiste de um ponto final para comunicação bidirecional entre duas partes, cliente e servidor. O canal de comunicação trata-se na verdade de dois sockets interconectados
- Na comunicação cliente e servidor:
  - Servidor aguarda por conexão de um cliente
  - Servidor pode atender vários clientes (ex: fork, threads)
  - Baixo acoplamento: A única exigência é que cliente e servidor falem a mesma língua, ou seja, que exista um protocolo bem definido e conhecido por ambos. Desta forma, cliente e servidor podem ser até desenvolvidos em diferentes linguagens de programação ou serem executados em diferentes sistema operacionais ou arquiteturas de máquina.

Na linguagem C, um socket é acessado através de um descritor, de forma análoga a um descritor de arquivos. Pode-se então executar operações de escrita (`write` or `send`) e leitura (`read` ou `recv`). Funções necessárias para trabalhar com sockets estão definidas em `<sys/sockets.h>`, sendo estas:

Função	Descrição
`socket`	para criar um socket
`bind`	para associar um nome ao socket
`listen`	para ouvir em uma determinada porta
`accept`	para receber conexões
`connect`	para conectar
`read` ou `recv`	para ler dados
`write` ou `send`	para escrever dados
`close`	para fechar

Diferentes famílias de processadores (arquiteturas) usam diferentes formas para representar internamente a ordem dos *bytes* (*big-endian* e *little-endian*). Assim é necessário que converta para o formato da rede antes de enviar números como sequência de *bytes* e lembre de converter de volta para o formato nativo após receber algo pela rede. As funções abaixo visam garantir que a ordem dos *bytes* nativa de cada arquitetura seja convertida para uma ordem de *byte* para a rede e vice-versa, garantindo assim harmonia na comunicação.

- `htons` – host to network short;

- `htonl` – host to network long;
- `ntohs` – network to host short;
- `ntohl` – network to host long.

Outro ponto que deve considerar ao enviar *bytes* via *sockets* é com o tamanho da palavra do CPU. Por exemplo, o cliente pode fazer uso de palavras de 32 bits e o servidor com 64 bits. Na linguagem C para transmitir números inteiros ao invés de usar o tipo `int`, é recomendado usar os tipos definidos na biblioteca `stdint.h`[1], por exemplo, o tipo `int32_t` que garante que todo inteiro será representado com 32 bits.

Para enviar cadeias de caracteres pela rede é importante saber que existem diferentes formas de codificação, por exemplo, ISO-8859-1, Windows-1252 e UTF-8. UTF-8 é o formato de codificação padrão da maioria dos sistemas operacionais e também de todos os protocolos da Internet padronizados pela IETF. O UTF-8[2] é de codificação binária de tamanho variável (1 a 4 bytes) que permite representar qualquer caractere universal.

- `1 byte` – para representar os 128 caracteres ASCII
- `2 bytes` – para representar caracteres latinos com diacríticos (i.e. acentos)
- `3 bytes` – para alfabetos Grego, Cirílico, Armênio, Hebraico, Sírio e Thaana
- `4 bytes` – para representar outros caracteres

Na linguagem C o tipo primitivo `char` ocupa 1 *byte* e permite representar caracteres ASCII[3]. O padrão C90 introduziu a biblioteca `wchar.h`, juntamente com o tipo `wchar_t` e algumas funções para lidar com caracteres UTF-8. É possível guardar uma cadeia de caracteres em UTF-8 em uma variável do tipo `char`, porém garanta que o tamanho do vetor seja suficientemente grande (lembre-se, são necessários 2 bytes para caracteres acentuados em português).

2 Exemplo de um servidor escrito na linguagem C

Para desenvolver uma aplicação servidora com sockets TCP, deve-se:

1. criar um socket;
2. associar um nome ao socket (é aqui que deve-se indicar o IP e porta);
3. ouvir em uma determinada porta;
4. aceitar as conexões dos clientes;
5. realizar comunicação com o cliente (receber ou enviar *bytes*);
6. fechar a conexão.

[1] https://cplusplus.com/reference/cstdint
[2] https://pt.wikipedia.org/wiki/UTF-8
[3] https://www.asciitable.com

2.1 Criando um socket

Ao criar um socket deve-se informar 3 parâmetros: domínio, tipo e protocolo.

- **Domínio**
  - `AF_UNIX` – somente para processos (cliente e servidor) executados em uma mesma máquina
  - `AF_INET` – para processos em diferentes máquinas – IPv4
  - `AF_INET6` – IPv6
- **Tipo**
  - `SOCK_STREAM` – orientado a conexão com transmissão de fluxos de bytes, sequencial e bidirecional
  - `SOCK_DGRAM` – orientado a datagramas, com tamanho fixo e entrega não confiável
  - `SOCK_RAW` – interface de datagrama diretamente para ir sobre o IP (camada de rede)
- **Protocolo**
  - Geralmente se informa o valor 0 para que se escolha automaticamente o protocolo adequado para o domínio e tipo. Se um domínio e tipo puder fazer uso de diferentes protocolos, então deve-se indicar se deseja TCP ou UDP.

```
1  /* Criando um socket */
2  // AF_INET = ARPA INTERNET PROTOCOLS -- IPv4
3  // SOCK_STREAM = orientado a conexao
4  // 0 = protocolo padrao para o tipo escolhido -- TCP
5  int socket_desc = socket(AF_INET , SOCK_STREAM , 0);
6
7  if (socket_desc == -1){
8    printf("Nao foi possivel criar socket\n");
9    return -1;
10 }
```

2.2 Associando um nome e ouvindo

```
1  struct sockaddr_in servidor;
2  servidor.sin_family = AF_INET;
3  servidor.sin_port = htons( 1234 ); // porta
4  servidor.sin_addr.s_addr = INADDR_ANY; // Obtém IP do S.O.
5
6  //Associando o socket a porta e endereço
7  if( bind(socket_desc,(struct sockaddr *)&servidor , sizeof(servidor)) < 0){
8    puts("Erro ao fazer bind\n");
9  }
10 puts("Bind efetuado com sucesso\n");
11
12 // Ouvindo por conexões
13 listen(socket_desc, 3);
```

2.3 Aceitando conexões

```c
struct sockaddr_in cliente;
int c;
puts("Aguardando por conexoes...\n(pressione CTRL+C para encerrar o processo)\n\n");
c = sizeof(struct sockaddr_in);
conexao = accept(socket_desc, (struct sockaddr *)&cliente, (socklen_t*)&c);
if (conexao<0){
  perror("Erro ao receber conexao\n");
  return -1;
}
```

2.4 Realizando comunicação e fechando o socket

```c
char *cliente_ip;
int cliente_port;
cliente_ip = inet_ntoa(cliente.sin_addr); // obtendo o endereço IP do cliente
cliente_port = ntohs(cliente.sin_port); // obtendo a porta de origem do cliente
printf("cliente conectou: %s : [%d]\n",cliente_ip, cliente_port);

// lendo dados enviados pelo cliente
if((tamanho = read(conexao,resposta, 1024)) < 0){
  perror("Erro ao receber dados do cliente: ");
  return -1;
}

// Coloca terminador de string
resposta[tamanho] = '\0';
printf("O cliente falou: %s\n", resposta);

// Enviando resposta para o cliente
mensagem = "Ola cliente, tudo bem?";
write(conexao , mensagem , strlen(mensagem));

// fechando
shutdown(socket_desc, 2);
printf("Servidor finalizado...\n");
```

3 Exemplos com sockets TCP e UDP na linguagem C

3.1 Servidor atende um único cliente e é encerrado após o atendimento

O servidor fica ativo esperando conexão de clientes, mas atende apenas um único cliente e espera receber uma única mensagem (cadeia de caracteres) desse cliente. O cliente, após conectar no servidor, envia uma cadeia de caracteres. O servidor imprime a mensagem em seu console, retorna uma cadeia de caracteres como resposta e por fim, finaliza o processo. O cliente imprime a resposta em seu console e depois finaliza o processo.

- Baixe o código do cliente clicando aqui.
- Baixe o código do servidor clicando aqui.

3.2 Envio de números inteiros

O servidor fica ativo esperando conexão de clientes, mas atende apenas um único cliente e espera receber dois números inteiros (duas mensagens). O cliente, após conectar no servidor, envia os dois inteiros em duas mensagens separadas. O servidor faz a soma dos números recebidos, imprime o resultado em seu console, retorna o resultado ao cliente e por fim, finaliza o processo. O cliente imprime a resposta em seu console e depois finaliza o processo.

- Baixe o código do cliente clicando aqui.
- Baixe o código do servidor clicando aqui.

3.3 Servidor atende um único cliente por vez, mas continua ativo após atendimento

O servidor atende um único cliente por vez e espera receber uma única mensagem desse cliente, mas continua ativo após atendimento. O cliente, após conectar no servidor, envia uma cadeia de caracteres. O servidor imprime a mensagem em seu console, retorna uma cadeia de caracteres como resposta.

- Baixe o código do servidor clicando aqui.
  - Para interagir com esse servidor, use o cliente apresentado na Subseção 3.1.

3.4 Servidor dispara *threads* para atender clientes de forma concorrente

Para cada cliente que conectar, o servidor irá disparar uma *thread* específica para atendê-lo. Esse servidor tem um comportamento diferente dos anteriores, pois aqui é ele quem manda a primeira mensagem no *sockets* logo após o cliente conectar. O processo continua em execução mesmo que não tenha mais clientes conectados.

- Baixe o código do cliente clicando aqui.
- Baixe o código do servidor clicando aqui.

3.5 Cliente e servidor usando protocolo UDP

- Baixe o código do cliente clicando aqui.
- Baixe o código do servidor clicando aqui.

4 Exemplos com sockets TCP e UDP em Java com a API JAVA I/O

Java possui diferentes APIs para lidar com operações de I/O:. Aqui serão apresentados exemplos com a API Java I/O, que segue modelo bloqueante e é mais simples de entender. A API Java NIO.2 permite desenvolver aplicações com comunicação assíncrona, porém seu entendimento demanda mais tempo. Veja mais detalhes em https://docs.oracle.com/javase/8/docs/technotes/guides/io/index.html.

- Exemplo com TCP
  - Baixe o código do cliente Java TCP clicando aqui.
  - Baixe o código do servidor Java TCP clicando aqui.

- Exemplo com UDP
  - Baixe o código do cliente Java UDP clicando aqui.
  - Baixe o código do servidor Java UDP clicando aqui.

5 Exercício

1. Desenvolva um aplicativo cliente C e um aplicativo servidor em Java que permitam a transferência de arquivos (de qualquer tamanho) pela rede. O servidor deverá ser capaz de atender múltiplos clientes (várias *threads*). Ao subir o servidor deve-se informar a porta e o caminho do diretório onde estarão os arquivos que serão servidos aos clientes. Ao iniciar o cliente deve-se informar, como argumentos de linha de comando, o endereço IP e porta do servidor, bem como o nome do arquivo que deseja transferir. Exemplos:

```
java Servidor 1234 /tmp/repositorio

./cliente 127.0.0.1 1234 arquivo.txt
```