Cliente-servidor com Sockets TCP

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Comunicação orientada à conexão via TCP/IP

- Servidor fica à espera de ligações.
- Cliente liga-se ao servidor, sendo estabelecida conexão.
- Conexão é um canal fiável para comunicação bidireccional.
- Um socket representa um extremo de uma conexão.
- Uma conexão é caracterizada por um socket-pair.

Características de uma conexão TCP

- Fiabilidade:
  - quando são enviados dados é esperado um acknowledgment;
  - se não vier, os dados são retransmitidos.
- Ordem:
  - os dados são partidos em segmentos numerados;
  - se chegarem fora de ordem são reordenados;
  - se chegar em duplicado é descartado.
- Controlo de fluxo:
  - cada lado diz quantos bytes está disposto a receber: janela;
  - janela representa tamanho de buffer disponível no receptor;
  - pode aumentar ou diminuir conforme dados são lidos pela aplicação ou chegam;
  - se receptor estiver lento, emissor tem que esperar.
- Full-duplex:
  - uma conexão é bidireccional, para enviar e receber dados.

Endereçamento

- Cada máquina (host) tem (pelo menos) um endereço ip.
  Exemplo: 193.136.19.96
- Podem existir vários serviços em cada máquina.
- Serviços de uma máquina são distinguidos por portos.
- Um servidor especifica em que porto escuta.
- Um cliente especifica onde se ligar com um par (ip,porto).

Gamas de portos

- Um porto é um número de 16 bits.
- Serviços internet com números de 0–1023, standardizados pela IANA (Internet Assigned Numbers Authority); e.g.:
  - echo: 7
  - ftp: 21
  - ssh: 22
  - telnet: 23
  - www: 80
- Portos registadas, de 1024–49151; registadas na IANA como conveniência à comunidade.
- Portos dinâmicos, de 49152-65535:
  - são atribuídos a sockets de clientes para distinguir conexões;
  - podem ser usados por servidores que usam um serviço de localização.
- Ver lista em http://www.iana.org/assignments/port-numbers.

Cliente-servidor com TCP – esqueleto

Cliente:

```
socket()
connect()
while ()
  write()
  read()
close()
```

Servidor

```
socket()
bind()
listen()
while ()
  accept()
  while ()
    read()
    write()
  close()
```

Cliente

Etapas de um cliente:

- Criar socket.
- Connect:
  - é especificado o ip e porto do servidor;
  - inicia o three-way handshake para estabelecer conexão com o servidor.
- Troca de dados escrevendo e lendo do connected socket.
- Close: fecha conexão. (Pode ser cliente ou servidor.)

Servidor

Etapas de um servidor

- Criar socket.
- Bind: atribui endereço local ao socket:
  - tipicamente apenas o porto;
  - pode ser especificado qual dos ip locais no caso de uma máquina com vários ip.
- Listen:
  - coloca o socket no modo *listening*, para poder aceitar pedidos de connect a ele dirigidos;
  - permite especificar o backlog: quantas conexões são guardadas pelo kernel antes de serem retiradas pelo accept.
- Accept:
  - devolve *connected socket* da lista de conexões estabelecidas;
  - bloqueia caso ainda não exista conexão estabelecida.
- Troca de dados lendo e escrevendo do connected socket.
- Close: fecha conexão. (Pode ser cliente ou servidor.)

Estabelecimento de conexão

- O servidor tem que estar preparado para aceitar conexões, fazendo socket, bind, listen (um chamado *passive open*).
- Um accept que seja feito bloqueia.
- É efectuado um three-way handshake:
  - O cliente faz um *active open* com connect; é enviado um SYN.
  - O servidor toma nota de conexão a ser estabelecida; faz ACK ao SYN e envia o seu SYN; quando segmento chega, o connect retorna no cliente.
  - O cliente faz ACK ao SYN do servidor; quando segmento chega, a conexão está estabelecida e um accept pode retornar.

Estabelecimento de conexão

- Entre o momento em que uma conexão começa e acaba de ser estabelecida passa um *round-trip time* (RTT);
- Durante este tempo, a conexão está numa fila de conexões a ser estabelecidas.
- Se forem chegando muitos pedidos de conexão, esta fila pode crescer.
- O backlog especifica a soma dos tamanhos das filas de conexões a ser estabelecidas ou já estabelecidas e ainda não obtidas pelo accept.
- É importante que o backlog seja alto para servidores com muita carga em redes com elevada latência.

Terminação de conexão

- São efectuados 4 passos:
  1. Um lado (cliente ou servidor) faz close (*active close*), sendo enviado um FIN.
  2. O outro lado que recebe o FIN faz um *passive close*, sendo enviado um ACK do FIN; é provocado um EOF na leitura.
  3. Mais tarde a aplicação que recebeu o EOF faz close do socket; é enviado um FIN.
  4. O lado que fez o active close faz ACK a este FIN.
- Entre os passos 2 e 3, é possível serem enviados dados para quem fez o active close; neste caso temos um *half-close*.
- Um half-close pode ser obtido com *shutdown*, para permitir a parte activa receber dados.

Sockets em Java

- Classes Socket e ServerSocket, em java.net;
- No cliente, criação de socket e connect efectuados em conjunto pelo construtor:

```
String host;
int port;
Socket con = new Socket(host, port);
```

- No servidor, criação de socket, bind e listen efectuados pelo construtor de ServerSocket:

```
ServerSocket srv = new ServerSocket(port);
```

- O accept devolve um connected socket de uma conexão:

```
Socket con=srv.accept();
```

Streams

- Os InputStream e OutputStream associados ao socket são obtidos com:

```
InputStream is = con.getInputStream();
OutputStream os = con.getOutputStream()
```

- Normalmente são usados streams com mais funcionalidade:

```
BufferedReader in =
    new BufferedReader(new InputStreamReader(con.getInputStream()));
PrintWriter out = new PrintWriter(con.getOutputStream());

DataInputStream dis = new DataInputStream(con.getInputStream());
DataOutputStream dos = new DataOutputStream(con.getOutputStream());

ObjectInputStream ois = new ObjectInputStream(con.getInputStream());
ObjectOutputStream oos = new ObjectOutputStream(con.getOutputStream());
```

Exemplo: Hello world, cliente em Java

```
import java.io.*;
import java.net.*;

public class Client {
    public static void main(String[] args) {
        try{
        if(args.length<2)
            System.exit(1);
        String host = args[0];
        int port = Integer.parseInt(args[1]);
        Socket s = new Socket(host, port);
        BufferedReader in =
            new BufferedReader(new InputStreamReader( s.getInputStream()));
        PrintWriter out = new PrintWriter(s.getOutputStream());
        out.println("Hello world");
        out.flush();
        String res = in.readLine();
        System.out.println(res);
        }catch(Exception e){
            e.printStackTrace();
            System.exit(0);
        }
    }
}
```

Exemplo: Hello world, servidor em Java

```
import java.io.*;
import java.net.*;

public class Server {
    public static void main(String[] args) {
        try{
        int port = java.lang.Integer.parseInt(args[0]);
        ServerSocket srv = new ServerSocket(port);
        while (true) {
            Socket cli=srv.accept();
            System.out.println(cli.getInetAddress() + "␣"+ cli.getPort() );
            BufferedReader in =
                new BufferedReader(new InputStreamReader(cli.getInputStream()));
            PrintWriter out = new PrintWriter(cli.getOutputStream());
            String st = in.readLine();
            String res = st.toUpperCase();
            out.println(res);
            out.flush();
        }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Exemplo: servidor de chat em Erlang

```
-module(chat).
-export([server/1]).

server(Port) ->
  {ok, LSock} = gen_tcp:listen(Port, [binary, {packet, line}]),
  Room = spawn(fun()-> room([]) end),
  acceptor(LSock, Room).

acceptor(LSock, Room) ->
  {ok, Sock} = gen_tcp:accept(LSock),
  Room ! {new_user, Sock},
  gen_tcp:controlling_process(Sock, Room),
  acceptor(LSock, Room).
```

- Dois processos: room e acceptor;
- Linhas convertidas em mensagens enviadas ao room;

Exemplo: servidor de chat em Erlang (room)

```
room(Sockets) ->
  receive
    {new_user, Sock} ->
      io:format("new user~n", []),
      room([Sock | Sockets]);
    {tcp, _, Data} ->
      io:format("received ~p~n", [Data]),
      [gen_tcp:send(Socket, Data) || Socket <- Sockets],
      room(Sockets);
    {tcp_closed, Sock} ->
      io:format("user disconnected~n", []),
      room(Sockets -- [Sock]);
    {tcp_error, Sock, _} ->
      io:format("tcp error~n", []),
      room(Sockets -- [Sock])
  end.
```

Exemplo: servidor de chat em Erlang (V2)

```
-module(chatv2).
-export([start/1, stop/1]).

start(Port) -> spawn(fun() -> server(Port) end).
stop(Server) -> Server ! stop.

server(Port) ->
  {ok, LSock} = gen_tcp:listen(Port, [binary, {packet, line}, {reuseaddr, true}]),
  Room = spawn(fun()-> room([]) end),
  spawn(fun() -> acceptor(LSock, Room) end),
  receive stop -> ok end.

acceptor(LSock, Room) ->
  {ok, Sock} = gen_tcp:accept(LSock),
  spawn(fun() -> acceptor(LSock, Room) end),
  Room ! {enter, self()},
  user(Sock, Room).
```

- Processos: room, acceptor, e um processo por cliente (user);
- Depois do accept, é criado novo acceptor;
- Acceptor corrente passa a servir cliente (torna-se user);

Exemplo: servidor de chat em Erlang (V2, room)

```
room(Pids) ->
  receive
    {enter, Pid} ->
      io:format("user␣entered~n", []),
      room([Pid | Pids]);
    {line, Data} = Msg ->
      io:format("received␣~p~n", [Data]),
      [Pid ! Msg || Pid <- Pids],
      room(Pids);
    {leave, Pid} ->
      io:format("user␣left~n", []),
      room(Pids -- [Pid])
  end.
```

- Mantém lista de processos user que interagem com clientes;
- Não usa os sockets directamente;

Exemplo: servidor de chat em Erlang (V2, user)

```
user(Sock, Room) ->
  receive
    {line, Data} ->
      gen_tcp:send(Sock, Data),
      user(Sock, Room);
    {tcp, _, Data} ->
      Room ! {line, Data},
      user(Sock, Room);
    {tcp_closed, _} ->
      Room ! {leave, self()};
    {tcp_error, _, _} ->
      Room ! {leave, self()}
  end.
```

- User interage com clientes remotos via socket, e com room;
- Único código que manipula o formato do protocolo de rede;