Lauro Cruz e Souza - 156175

Pedro Emílio Machado de Brito - 137264

Atividade 2.1

server.c

O servidor foi implementado de forma a esperar em loop por conexões, e também em um loop, ler uma linha de texto, imprimi-la, mandar eco, e tentar ler mais texto, assim como no trabalho 1.

Nessa versão do servidor, no loop que aceita conexões, sempre que uma nova conexão for aceita, o programa realiza um fork e chama a função `handle_connection(int s)`, que vai realizar o trabalho descrito acima para a conexão feita no socket local `s`.

A função também é responsável por informar no começo o cliente que se conectou ao servidor (IP e porta remotos).

Todas as funções que tratam da conexão com a rede tem seus códigos de erro verificados e tratados, se necessário, assim como no trabalho 1.

Além das mesnagens de erro já existentes na versão anterior do trabalho 1, um novo erro que podemos obter é quando o servidor não consegue obter os dados do cliente (IP, porta). Não obtivemos esse erro, mas a saída caso ocorresse seria:

ERROR: Could not resolve remote port and ip values

client.c

Assim como no trabalho 1, o cliente foi implementado de forma a se conectar ao servidor, e então entrar em um loop de enviar uma linha de texto, e esperar pelo eco. O cliente para de ler texto e fecha a conexão ao receber o sinal CTRL-D (EOF), e então sai.

A única diferença no cliente é que ao se conectar ao servidor, ele informa seu IP e porta de conexão usando a função `getsockname`.

Além das mensagens de erro já presentes no trabalho 1, podemos agora também obter um erro na obtenção do IP e porta locais. Apesar de não termos obtido este erro, a mensagem a ser impressa seria:

ERROR: Could not resolve local port and ip values

Funções utilizadas nos programas:

Além das funções utilizadas no trabalho 1, que foram todas mantidas, utilizamos também:

`fork`: Cria novo processo a partir do ponto de chamada.

`getsockname`: Dado o número do socket, retorna a estrutura `sockaddr` com os dados daquele socket, incluindo endereço de IP e número da porta (utilizado para obter os dados da conexão).

`inet_ntoa`: Traduz o IP presente na estrutura `sockaddr` para uma string.

`ntohs`: Traduz o valor da porta na estrutura `sockaddr` para um inteiro.

Um exemplo de sessão:

Em um terminal na máquina local, rodamos:

`$ ./server`

Em outro na mesma máquina rodamos

`$ ./client localhost`

estabelecendo uma conexão. A porta usada é implicitamente 12345, pelas constantes definidas em cada arquivo fonte.

Temos então os primeiros outputs com as infos:

```
$ ./client localhost
Local IP: 127.0.0.1
    Port: 46704
```

```
$ ./server
CLIENT CONNECTED
  IP: 127.0.0.1
PORT: 46704
```

Várias linhas de texto são digitadas no console com o cliente aberto:

```
$ ./client localhost
oi
oi
teste
teste
batata
batata
the quick brown fox jumps over the lazy dog
```

```
the quick brown fox jumps over the lazy dog
```

```
$ ./server

CLIENT CONNECTED
  IP: 127.0.0.1
PORT: 34552

From 127.0.0.1:
oi

From 127.0.0.1:
teste

From 127.0.0.1:
batata

From 127.0.0.1:
the quick brown fox jumps over the lazy dog
```

Nesse ponto conectamos mais um cliente no servidor, rodando `./client <hostname>` na máquina `xaveco` do IC (por SSH). Obtemos assim:

```
$ client

Local IP: 143.106.16.163
    Port: 35344
oi2
oi2
teste2
teste2
batata2
batata2
the quick brown fox jumps over the lazy dog 2
the quick brown fox jumps over the lazy dog 2
```

```
$ server

CLIENT CONNECTED
  IP: 127.0.0.1
PORT: 34552

From 127.0.0.1:
oi

From 127.0.0.1:
teste

From 127.0.0.1:
batata

From 127.0.0.1:
```

```
the quick brown fox jumps over the lazy dog

CLIENT CONNECTED
  IP: 143.106.16.163
PORT: 35344

From 143.106.16.163:
oi2

From 143.106.16.163:
teste2

From 143.106.16.163:
batata2

From 143.106.16.163:
the quick brown fox jumps over the lazy dog 2
```

Mesmo um cliente tendo terminado, o servidor continua ativo, esperando, outras conexões.