Relatório da atividade 2.2

Isadora Sophia Garcia Rodopoulos *
Matheus Mortatti Diamantinos †
Luiz Fernando Bittencourt‡

Abstract

O objetivo do trabalho se baseou em modificar a implementação da aplicação cliente-servidor anterior, substituindo `fork()` por `select()`.

Neste projeto, foi implementado uma estrutura de comunicação entre cliente e servidor baseado em uma conexão TCP utilizando sockets na linguagem C. Nela, vários clientes podem se conectar a um mesmo IP, cada um por uma porta diferente. Foi utilizada a estrutura de `select()` para permitir que vários clientes se conectem simultaneamente.

Além disso, um *ack* é enviado para cada um dos clientes, em formato de *echo*, para certificar que a mensagem chegou com sucesso.

1. client.c

O código para o cliente não sofreu nenhuma modificação em relação aos projetos anteriores, uma vez que o seu comportamento continua o mesmo, basta conectar-se ao IP do servidor e a uma porta disponível.

O fluxo de execução de `client.c` pode ser resumido, portanto, em:

1. Criar o socket de conexão;
2. Estabelecer conexão com o servidor;
3. Receber mensagem do usuário, mandar ao servidor e esperar resposta;
4. Se algum erro ocorrer ou o cliente fechar a aplicação, fechar a conexão.

*RA 158018, Instituto de Computação, Universidade de Campinas, Unicamp. **Contact**: `isadorasophiagr@gmail.com`

†RA 156740, Instituto de Computação, Universidade de Campinas, Unicamp. **Contact**: `matdiamantino@gmail.com`

‡MC833, Instituto de Computação, Universidade de Campinas, Unicamp. **Contact**: `bit@ic.unicamp.br`

2. server.c

Semelhante ao comportamento do programa do cliente, o servidor foi divido em:

1. Criar o socket ativo e associá-lo a um descritor;
2. Inicializar `select()`;
3. Enquanto está ativo:
   (a) Checar novas conexões de clientes. Caso exceda o limite de conexões, exibir um erro e sair da aplicação.
   (b) Checar mensagens dos clientes ativos. Caso receba uma mensagem, imprimi-la e retornar o *echo*, especificando o IP e o port do cliente.
   (c) Caso um cliente se desconecte, fechar a conexão e desalocar o espaço do cliente.

Foi estabelecida uma interface bem simples, que permite que o usuário veja as mensagens que serão ecoadas na tela - as quais correspondem às mensagens que o cliente envia através da conexão com o servidor.

2.1. Criar o socket ativo e associá-lo a um descritor;

Do mesmo modo que foi feito no cliente, foi criado um socket com a família da qual o endereço pertence (no nosso caso, IPv4), o seu respectivo port de *listening* e o IP, *localhost*. A associação do socket ao descritor se dá a partir da função *bind()*.

Caso ocorra algum erro ou o socket não consiga aceitar nenhuma conexão, um erro é emitido e o programa é finalizado.

2.2. Inicializar `select()`;

A inicialização se dá desligando todos os bits do file descriptor, e associando o bit ao socket adequado, como visto em sala de aula.

Code 1. Inicialização de conjunto de descritores

```
FD_ZERO(&all_fds);
FD_SET(s, &all_fds);
```

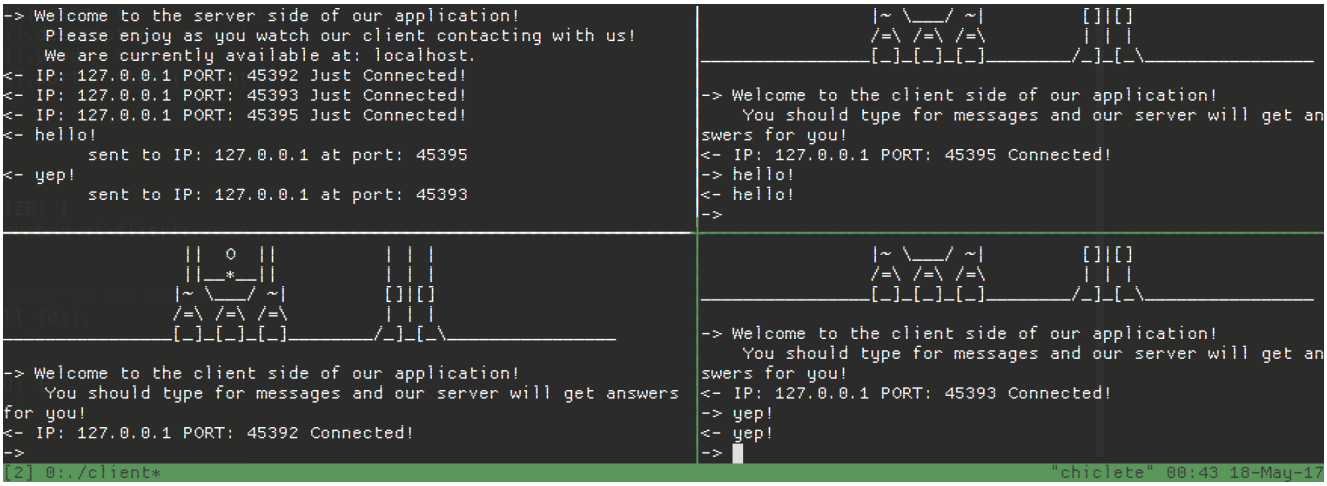

Figure 1. Exemplo de funcionamento do projeto

2.3. Enquanto está ativo...

2.3.1 Checar novas conexões de clientes. Caso exceda o limite de conexões, exibir um erro e sair da aplicação;

A função *select()* é chamada, de modo a permitir o monitoramento dos file descriptors. Assim, é verificado se existe alguma conexão nova, feita por algum cliente, com a função `FD_ISSET()`.

Caso positivo, é realizada a conexão (`accept()`) a partir do registro do novo cliente ao servidor. As informações do cliente são coletadas (`getpeername()`) para manter a conexão posteriormente, e um novo file descriptor é adicionado.

Senão, basta partir para o próximo passo.

Code 2. Estabelecimento de conexão a um novo cliente

```
valid((new_s = accept(s, (struct sockaddr *)
    &socket_addr, &len)), ..);
..
getpeername(new_s, (struct sockaddr *)&client,
    &client_size);
inet_ntop(AF_INET, &(client.sin_addr), ip,
    INET_ADDRSTRLEN);
..
/* check available spot */
for (i = 0; i < FD_SETSIZE; i++) {
    if (!clients[i].used) {
        clients[i].descriptor = new_s;
        clients[i].port = ntohs(client.sin_port);
        clients[i].used = true;
        break;
    }
}
if (i == FD_SETSIZE)
    error("Max. number of clients reached!");

/* add new descriptor to our set */
FD_SET(new_s, &all_fds);
```

Observe que, caso o limite de conexões ao servidor seja atingido ou qualquer chamada de função tenha retornado um valor de erro, o programa é finalizado.

2.3.2 Checar mensagens dos clientes ativos. Caso receba uma mensagem, imprimi-la e retornar o *echo*, especificando o IP e o port do cliente;

Assim, para cada um dos clientes registrados e ativos, é verificado se o socket foi ativado de acordo com seu respectivo file descriptor.

Caso positivo, a mensagem é capturada (`recv()`). Além disso, é verificado se a conexão foi encerrada - caso tenha ocorrido, o seu file descriptor é registrado como 'desligado' (`FD_CLR()`) e mais um spot é liberado para que um cliente se conecte ao servidor. Se a conexão não foi encerrada, basta imprimir a mensagem recebida e enviá-la de volta ao cliente (`send()`), através da respectiva porta e IP.

Senão, basta partir para o próximo cliente, até que não esteja esperando mais nenhuma mensagem de descritor.

Code 3. Recebimento de mensagens das conexões em aberto

```
len = recv(sockfd, buff, sizeof(buff), 0);
..
if (!len) {
    close(sockfd);
    FD_CLR(sockfd, &all_fds);

    clients[i].used = false;
} else {
    ..
    /* print IP and port and send back to client */
    if (send(new_s, buff, len, 0) == ERROR) {
        fprintf(stdout, "..");
        break;
    }
}
```

2.4. Comportamentos inesperados

Caso um cliente se desconecte, fechar a conexão e desalocar o espaço do cliente, ou qualquer erro ocorra durante os procedimentos descritos acima, é realizado o tratamento de erro e, caso seja necessário, a aplicação é encerrada imediatamante.

A mesma API utilizada nos projetos anteriores, `Api.h`, foi utilizada para encapsular as chamadas de erros.

3. Testes

Para testar o funcionamento da infraestrutura, foi necessário abrir o processo do servidor e diversos clientes para se conectar ao servidor por diferentes portas. Foram testados tanto os corner quanto edge cases (como nenhum cliente ou muitos clientes se conectando simultaneamente) e foi checado se os erros foram tratados como o apropriado.

Apesar de algumas oscilações entre as conexões dos clientes (por algum motivo, eventualmente alguns programas do cliente não imprimiriam os *ack* do servidor até que a conexão fosse encerrada, por conflitos do próprio OS), o comportamento se manteve dentro o esperado.