Tema 3

Comunicação entre aplicações

Objetivos:

- Comunicação entre aplicações
- *Sockets* UDP
- *Sockets* TCP
- Acesso Assíncrono

3.1 Introdução

As aplicações informáticas realizam trabalho sobre dados que lhes são fornecidos, o que tem sido feito através de argumentos, introdução de informação pelo teclado, ou de ficheiros. No entanto também é possível e extremamente útil as aplicações trocarem informação diretamente entre si. Um exemplo muito comum é a consulta de páginas na Internet. Segundo a indicação do utilizador, um navegador *Web* irá consultar os diversos servidores de forma a obter páginas. Tanto o navegador, como o(s) servidores *Web* são aplicações que possuem a capacidade de trocarem informação, o que é feito através de mensagens que são transmitidas através de uma rede. A construção e transmissão das mensagens foi já abordado anteriormente. Este guião aborda a geração, envio e recepção das mensagens pelas aplicações.

3.2 Conceitos de comunicação

3.2.1 Modelo Cliente-Servidor

Nas comunicações entre aplicações é usual distinguir-se dois tipos de intervenientes: o cliente e o servidor. O cliente é aquele que inicia a comunicação e faz um pedido, enquanto o servidor é aquele que aceita um pedido e realiza uma ação, podendo emitir uma resposta. Por exemplo, o navegador *Web* é um cliente que emite pedidos enquanto os servidores *Web* são aqueles que aceitam pedidos dos clientes, fornecendo depois as páginas como resposta.

A Figura 3.1 representa uma simples interacção entre vários clientes e um servidor.

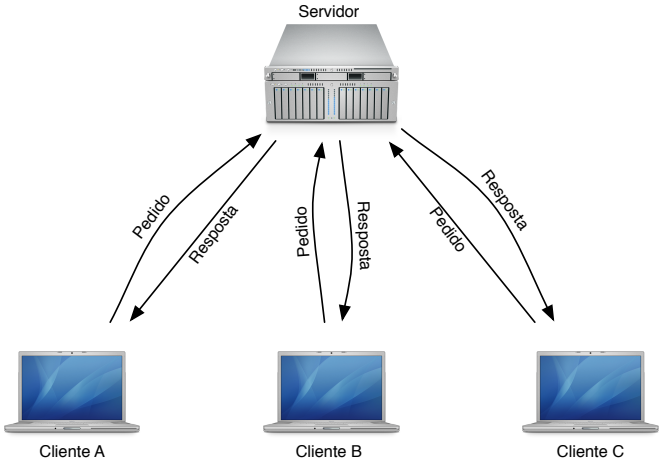

Figura 3.1: Modelo Cliente-Servidor

Esta separação de funções é importante porque implica que as aplicações do cliente e do servidor são distintas: o fluxo lógico (algoritmo) do cliente é diferente do fluxo lógico do servidor. Considerando o mesmo exemplo, os servidores *Web* têm de fornecer páginas e outros ficheiros, enquanto os clientes têm de processar HyperText Markup Language (HTML)[1], JavaScript (JS)[2], Cascading Style Sheets (CSS)[3], construir a página e interagir com o cliente. Outra característica deste modelo é que vários clientes se podem ligar a um único servidor.

Existem aplicações, como as utilizadas nas redes Peer to Peer (P2P) que são construídas de forma a funcionarem ao mesmo tempo como cliente e servidor. Isto não vai contra o dito anteriormente relativamente a existirem dois tipos de aplicações, pois para suportar esta modalidade, estas aplicações têm de possuir código para ambas as funções. Quando um aplicação P2P comunica com outra, durante esta transação essa age como servidor e a outra como cliente.

3.2.2 Sockets

As aplicações na consola podem interagir com o utilizador através de três dispositivos básicos: **`stdin`**, **`stdout`** e **`stderr`**. Estes dispositivos têm sido extensivamente utilizados quando se lê texto do teclado ou se escreve para a consola. As aplicações também podem criar representações internas de ficheiros e diretórios, que depois utilizam para ler, escrever, criar ou apagar estes elementos. Para permitir a comunicação entre processos, ou Inter Process Communication (IPC), existem outros mecanismos como o *Socket*.

O termo inglês *Socket* designa uma tomada onde se pode ligar uma ficha, tal como uma tomada de corrente eléctrica ou uma tomada de rede ethernet. No âmbito do software, um *Socket* é um mecanismo pelo qual as aplicações podem criar tomadas virtuais onde outras aplicações se podem "ligar". Uma aplicação pode criar quantos *Sockets* quiser, tal como uma casa pode ter quantas tomadas forem necessárias. A Figura 3.2 demonstra uma aplicação com vários pontos de comunicação, sendo que alguns são *Sockets*.

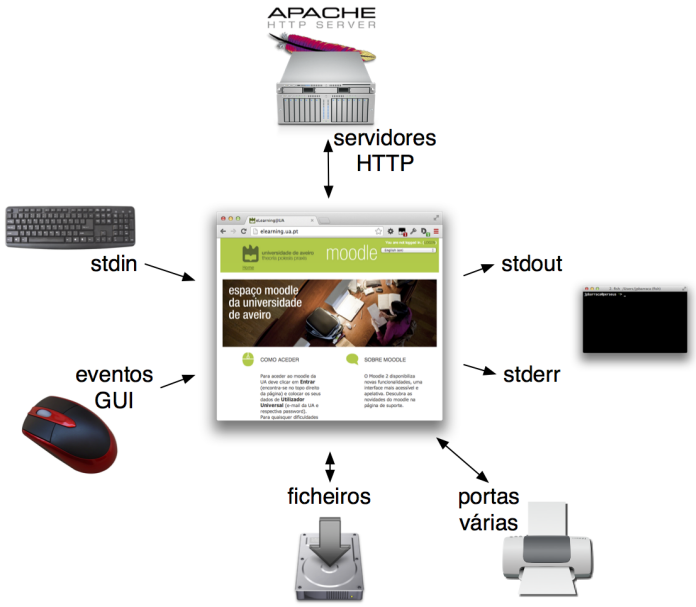

Figura 3.2: Vários pontos de comunicação com um navegador *Web*.

Tal como uma tomada possui um dada especificação, quanto à sua forma e contactos, também um *Socket* possui algumas características distintivas, das quais se destacam a *família* e o *tipo*. Um *Socket* de uma dada família e tipo só aceita ligações com essas características e não com outras.

Existem várias famílias de *Socket*. As principais são:

AF_UNIX: Indica um *Socket* para ser utilizado em comunicações entre aplicações locais (na mesma máquina).

AF_INET: Indica um *Socket* para ser utilizado sobre endereços Internet Protocol v4 (IPv4)[4]. Pode ser utilizado para comunicações entre aplicações locais ou remotas.

AF_INET6: Indica um *Socket* para ser utilizado sobre endereços Internet Protocol v6 (IPv6)[5]. Pode ser utilizado para comunicações entre aplicações locais ou remotas.

Neste caso, um *Socket* da família **AF_INET** pode ser utilizado para trocar mensagens com aplicações que estejam em sistemas com um IPv4 disponível, mas não permite trocar mensagens com aplicações que estejam em sistemas que só possuam IPv6.

Além da família, o tipo do *Socket* irá indicar como as mensagens devem ser encapsuladas antes de serem enviadas. Há dois tipos mais relevantes:

SOCK_DGRAM: As comunicações deverão utilizar o protocolo User Datagram Protocol (protocolo de transporte da Internet sem ligação, orientado ao datagrama) (UDP)[6]. Com este tipo de *Socket*, é possível que as mensagens se percam ou cheguem fora de ordem.

SOCK_STREAM: As comunicações deverão utilizar o protocolo Transmission Control Protocol (protocolo de transporte da Internet orientado à ligação) (TCP)[7]. Com este tipo de *Socket*, em caso de perda, o protocolo TCP irá retransmitir as mensagens. Este também garante que a ordem de envio é mantida à chegada.

3.3 Sockets não orientados à ligação (UDP)

Um tipo de *Sockets* permite o envio de mensagens entre aplicações sem que estas estabeleçam uma ligação explícita persistente. Entre muitos outros cenários, são indispensáveis para envio de mensagens de *broadcast*, para sistemas com baixas capacidades computacionais, ou para comunicações com restrições de tempo real.

Este tipo de *Sockets* são normalmente utilizados nas redes IP TeleVision (IPTV) que temos em casa, nas aplicações Voice Over IP (VoIP) como o *Skype*, ou no processo de atribuição de endereços dinâmicos (Dynamic Host Configuration Protocol (DHCP)[8]). Este tipo de *Socket* utiliza o protocolo UDP para o transporte das mensagens.

Um *Socket*, seja ele orientado à ligação ou não, não realiza comunicações de forma imediata só porque existe. Comunicar com um *Socket* requer uma sequência de acções, nomeadamente:

Criação: Um *Socket* é criado, definindo-se a sua família e tipo. Utiliza-se para isso a instrução `socket`.

Nomeação: É necessário dar um nome ao *Socket*, usando-se a primitiva `bind`. O nome permite identificar **unicamente** um dado *Socket* num dado sistema que pode ter inúmeros *Sockets* de várias aplicações. As aplicações trocam informação a partir de e para um *Socket* em particular. O nome é composto por um caminho, normalmente um endereço IPv4 e um porto (ou porta).

Enviar Informação: A aplicação emissora usa a acção `send` para enviar informação para o *Socket*. O sistema de destino armazena essa informação numa memória associada ao *Socket*, temporariamente.

Receber Informação: A aplicação recetora usa a acção **`receive`** para recolher a informação recebida e armazenada pelo *Socket*.

Fechar: Tal como um ficheiro, o *Socket* deve ser fechado quando a comunicação termina.

A Figura 3.3 representa uma utilização destas primitivas numa comunicação entre uma aplicação cliente e uma aplicação servidora. Note que não há nenhuma diferença formal entre um *Socket* no servidor ou no cliente, mas a lógica das aplicações é diferenciada. Uma espera por pedidos, a outra efetua pedidos.

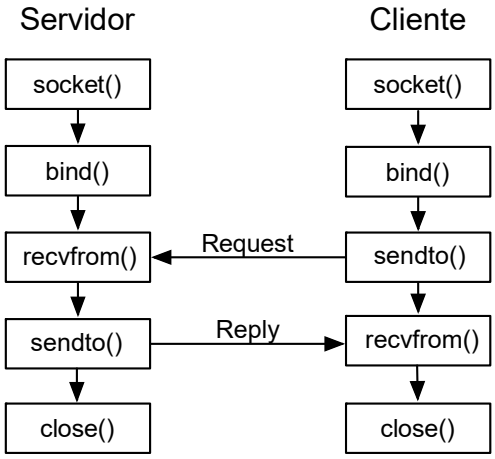

Figura 3.3: Sequência de primitivas utilizadas num *Socket* UDP.

Em *Python* é possível utilizar *Sockets* através da inclusão do módulo **`socket`**. Assim, um *Socket* pode ser criado e nomeado através das seguintes instruções:

```
import socket

def main():
    udp_s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_s.bind(("127.0.0.1", 1234))

main()
```

Neste exemplo o *Socket* é criado de forma a comunicar na porta **1234** para aplicações no próprio computador (`127.0.0.1`). Este endereço determina em que interface de rede o *Socket* vai comunicar, sendo que o valor especial `0.0.0.0` indica que irá comunicar através de todos os interfaces. Se o valor da porta for igual a 0, o sistema operativo irá escolher uma porta aleatória.

Atenção: Num dado sistema, não é possível existirem dois *Sockets* com o mesmo nome (endereço e porta)!

Para a troca de informação, é agora necessário receber e enviar informação. A seguinte implementação da função **`main`** do servidor espera por uma mensagem e responde enviando a mesma mensagem em maiúsculas. De notar que se utiliza o método **`recvfrom`** que possui como parâmetro o número máximo de octetos a receber e devolve 2 valores: uma string com os dados recebidos e o endereço do socket que os enviou.

```
def main():
    udp_s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_s.bind(("127.0.0.1", 1234))

    while True:
        b_data, addr = udp_s.recvfrom(4096)
        udp_s.sendto(b_data.upper(), addr)

    udp_s.close()
```

Exercício 3.1

Utilizando os exemplos anteriores, implemente um servidor de mensagens UDP designado por **udp_server.py**.

Para testar o servidor execute o programa **`python3 udp_server.py`** num terminal e noutro terminal utilize o comando **`nc -u localhost 1234`**. Neste terminal escreva mensagens e verifique as respostas do servidor.

O cliente para comunicar com esta aplicação seria programado de maneira muito semelhante. Apenas teria as instruções de envio e recepção por ordem inversa: primeiro envia e depois recebe a resposta. A implementação seguinte demonstra um cliente que lê uma frase do teclado, envia-a para o servidor, espera uma resposta e imprime-a.

```
import socket

def main():
    udp_s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_s.bind(("127.0.0.1", 0))
    server_addr = ("127.0.0.1", 1234)

    while True:
        str_data = input("<-: ")
        b_data = str_data.encode("utf-8")
        udp_s.sendto(b_data, server_addr)
        # ---
        b_data, addr = udp_s.recvfrom(4096)
        str_data = b_data.decode("utf-8")
        print("->: %s \n" % str_data)

    udp_s.close()
...
```

Exercício 3.2

Utilizando os exemplos fornecidos implemente um cliente (designado por **udp_client.py**) que permita a troca de mensagens. Tenha em consideração que, por residirem no mesmo sistema, o cliente não pode criar um *Socket* na mesma porta que o servidor.

Para testar o cliente coloque em execução o servidor num terminal e noutro terminal execute o programa `python3 udp_client.py`. No terminal do cliente escreva mensagens e verifique as respostas do servidor.

3.4 Sockets orientados à ligação (TCP)

Além das primitivas já vistas, um *Socket* do tipo `SOCK_STREAM` necessita de mais três. Isto deve-se ao facto dos *Sockets* deste tipo serem orientados à ligação, existindo a necessidade de se estabelecer uma ligação (ou sessão) entre as duas aplicações antes da transmissão de informação.

Aceitar Ligações: Um *Socket* do servidor irá ser definido como aceitando ligações de novos clientes, realizando-se esta ação através da instrução `listen`.

Estabelecer ligação: O cliente necessita de se ligar ao servidor antes de se poder trocar informação, usando-se para isso a instrução `connect`.

Aceitar Clientes: O servidor, após receber o pedido de ligação, pode aceitá-lo através da instrução `accept`.

Estas primitivas são utilizadas da forma descrita na Figura 3.4. Como pode ser visto, neste tipo de *Sockets*, **existe** uma declaração formal de que *Socket* é servidor ou cliente e a lógica das aplicações é também diferenciada. Uma aplicação espera por pedidos, outra efetua pedidos. As primitivas adicionais, necessárias ao estabelecimento da sessão, estão realçadas na Figura 3.4.

Utilizando *Python* é possível utilizar *Sockets* orientados à ligação de uma forma semelhante aos anteriores, mas agora considerando a necessidade de estabelecimento da ligação antes da troca de informação. Assim, um *Socket* TCP pode ser criado e nomeado através das seguintes instruções:

```
import socket

def main():
    tcp_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcp_s.bind(("127.0.0.1", 1234))

main()
```

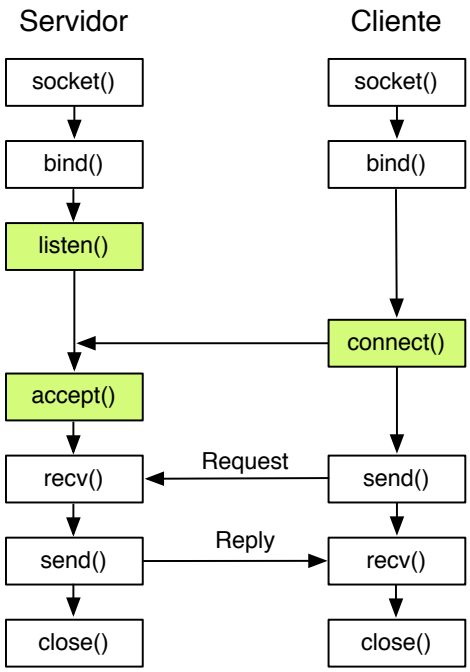

Figura 3.4: Sequência de primitivas utilizadas num *Socket* TCP.

Para o estabelecimento da sessão é agora necessário que o servidor defina o *Socket* como aceitando ligações e que espere por novos clientes, aceitando-os depois. Note que quando se aceita uma ligação de um novo cliente **é criado um novo *Socket***, que é utilizado para identificar a ligação entre o servidor e o cliente.

```
import socket

def main():
    tcp_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcp_s.bind(("127.0.0.1", 1234))
    tcp_s.listen(1) # máximo de 1 cliente à espera de aceitação

    # esperar por novo cliente
    client_s, client_addr = tcp_s.accept()

    while True:
        b_data = client_s.recv(4096)
        client_s.send(b_data.upper())

    client_s.close()
    tcp_s.close()
...
```

De notar que neste caso utiliza-se o *Socket* chamado **client_s** para todas as futuras comunicações. Também se utilizam os métodos **recv** em vez de **recvfrom** e **send** em vez de **sendto**. Isto porque não é necessário saber de onde chegou ou para onde vai a informação: toda a informação que seja lida ou escrita no **client_s** é trocada entre o servidor e o cliente específicos.

Exercício 3.3

Implemente um servidor TCP (**tcp_server.py**) utilizando o exemplo fornecido.

Para testar o servidor execute o programa **python3 tcp_server.py** num terminal e noutro terminal utilize o comando **nc localhost 1234**. Também pode utilizar o comando **telnet localhost 1234**. Neste terminal escreva mensagens e verifique as respostas do servidor.

O cliente será bastante semelhante ao caso dos *Sockets* não orientados à ligação, mas neste caso, é necessário estabelecer previamente a sessão.

```
import socket

def main():
    tcp_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcp_s.bind(("127.0.0.1", 0))
    tcp_s.connect(("127.0.0.1", 1234))  # Ligar ao servidor

    while True:
        str_data = input("<-: ")
        b_data = str_data.encode("utf-8")
        tcp_s.send(b_data)
        # ---
        b_data = tcp_s.recv(4096)
        str_data = b_data.decode("utf-8")
        print("->: %s \n" % str_data)

    tcp_s.close()
...
```

Tal como acontece com a implementação do servidor, a troca de informação é feita através dos métodos **recv** e **send**, não existindo necessidade de especificar sempre qual o destino das mensagens, ou de obter informação relativa à sua origem.

Exercício 3.4

Implemente um cliente que utilize *Sockets* orientados à ligação, designado por **tcp_client.py**. Mais uma vez, tenha em atenção que os clientes devem utilizar portas aleatórias de forma a não colidirem com serviços existentes.

Para testar o cliente proceda como foi explicado anteriormente para os *Sockets* UDP.

3.5 Servidor de Mensagens Instantâneas

De uma forma simples é possível implementar um servidor que actue como uma ferramenta de *chat*, permitindo a troca de mensagens entre utilizadores. Este exemplo é útil pois permite demonstrar com é possível interagir com múltiplos clientes e, no caso do cliente, ler de forma alternada de duas fontes de informação. De notar que o servidor e o cliente podem ser implementados utilizando qualquer um dos tipos de *Socket* abordados. De forma a simplificar a explicação, o exemplo irá focar-se no caso de comunicações através de UDP deixando-se a implementação com TCP para exercício.

A implementação do servidor é simples, bastando que possua uma lista de *Sockets* conhecidos e envie as mensagens recebidas de um *Socket* para todos.

```
import socket

def main():
    udp_s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_s.bind(("127.0.0.1", 1234))

    addr_list = []  # Lista de sockets conhecidos

    while True:
        b_data, addr = udp_s.recvfrom(4096)
        print(b_data.decode("utf-8"))

        # Adicionar o nome do socket à lista de sockets conhecidos
        if not addr in addr_list: addr_list.append(addr)

        # Enviar a mensagem para todos
        for dst_addr in addr_list: udp_s.sendto(b_data.upper(), dst_addr)

    udp_s.close()
...
```

Exercício 3.5

Implemente este servidor *chat* designado por **udp_chat_server.py**.

Ele deverá funcionar com o cliente UDP criado anteriormente. Mas como o cliente simples não suporta a nova lógica, não lhe serão apresentadas as mensagens dos outros clientes.

Um cliente *chat* também necessita de pequenas alterações de maneira a permanentemente ouvir novos dados do teclado e do *Socket*. O teclado permite ao utilizador enviar mensagens. O *Socket* permite receber as mensagens dos outros clientes. Isto é possível através da utilização da instrução `select`, que basicamente permite ficar à escuta de informação de múltiplas fontes, indicando depois qual das fontes possui informação para ser consumida.

O método **select** aceita três parâmetros: a lista de *Sockets* (ou outros dispositivos) onde se espera por dados, a lista de *Sockets* onde recentemente foram escritos dados e se espera que estes sejam transmitidos e a lista de *Sockets* onde se querem receber notificações de excepções (p.ex, *Socket* fechado). Irá devolver igualmente três listas com os *Sockets* que tiveram os eventos respetivos. Aqui apenas nos interessa a primeira das listas, a que indica os *Sockets* com informação pronta a ser consumida.

```
import socket
import select
import sys

def main():
    udp_s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_s.bind(("127.0.0.1", 0))
    server_addr = ("127.0.0.1", 1234)

    while True:
        rsocks = select.select([udp_s, sys.stdin, ], [], [])[0]

        for sock in rsocks:
            if sock == udp_s:
                # Informação recebida no socket
                b_data, addr = udp_s.recvfrom(4096)
                sys.stdout.write("%s\n" % b_data.decode("utf-8"))
            elif sock == sys.stdin:
                # Informação recebida do teclado
                str_data = sys.stdin.readline()
                udp_s.sendto(str_data.encode("utf-8"), server_addr)

    udp_s.close()
...
```

Exercício 3.6

Implemente um cliente *chat* (designado por **udp_chat_client.py**) de forma a que possa dialogar com os outros utilizadores ligados ao mesmo servidor.

Para testar o cliente proceda como foi explicado anteriormente para o modelo simples cliente-servidor, mas tendo em atenção que estamos perante um *chat*. Pelo que, deve abrir vários (pelo menos dois) terminais e invocar em cada um deles o cliente. Envie mensagens dos múltiplos clientes e verifique que o servidor envia cada mensagem recebida de um cliente para todos os clientes.

A implementação de um servidor *chat* (ou multi-cliente) com *Sockets* TCP é mais complexa do que para os *Sockets* UDP.

```python
import socket
import select

def main():
    tcp_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcp_s.bind(("0.0.0.0", 1234)) # Clientes multiplos na porta indicada
    tcp_s.listen(10) # Aceitar 10 clientes
    connections = []

    connections.append(tcp_s)
    print("Chat server started")

    while True:
        # Lista de sockets que podem ser lidos pelo select
        read_sockets = select.select(connections, [], [])[0]

        for sock in read_sockets:
            if sock == tcp_s: #Novo cliente?
                # Adicionar o socket do novo cliente à lista de sockets conhecidos
                client_s, addr = tcp_s.accept()
                connections.append(client_s)
                print("Client connected: %s" % str(addr))
            else:
                try: #Verificar se há uma mensagem de um cliente e processá-la
                    data = sock.recv(4096) # Mensagem válida de um cliente
                    if len(data) != 0:
                        print("Fom client: %s" % str(sock.getpeername()))
                        print("Got Data: " + data.decode("utf-8"))
                    else: # O cliente desligou-se
                        print("Client disconnected: %s" % str(sock.getpeername()))
                        sock.close()
                        connections.remove(sock)
                        break

                    # Criar a mensagem com identificação do cliente que a enviou
                    message = "<Fom client: " + str(sock.getpeername()) + "> "
                    message = message.encode("utf-8") + data.upper()

                    # Eventualmente não mandar a mensagem para o próprio
                    for client in connections: if client != tcp_s: client.send(message)

                except: # Erro no socket do cliente
                    print("Client socket error: %s" % str(addr))
                    sock.close()
                    connections.remove(sock) # retirar este socket da lista
                    continue

    for sock in connections: sock.close()

main()
```

Tal como foi explicado anteriormente o método `select` permite monitorizar a lista dos *Sockets* dos clientes ativos e também o *Socket* do servidor pois é ele que recebe a notificação de um cliente que quer comunicar com o servidor (método `connect` invocado pelo cliente).

Se o método `select` falhar isso significa que o *Socket* de algum cliente foi desligado. Nessa situação de exceção é necessário recorrer ao método `fileno` que devolve -1 como sinal de fracasso na tentativa de contato com o *Socket* do cliente, confirmando que ele foi desligado e consequentemente ele é removido da lista de clientes ativos.

Quando o método `select` indica que há informação válida nos *Sockets* analisados podemos ter uma de duas situações possíveis. Ou foi o *Socket* do servidor que recebeu dados e nesse caso estamos perante um novo cliente cujo *Socket* tem que ser adicionado à lista de clientes ativos. Se pelo contrário foi o *Socket* de um cliente ativo então temos que verificar se existe mesmo uma mensagem. E, neste caso ela é processada, ou seja é enviada com a identificação do autor, aos restantes clientes.

Exercício 3.7

Implemente este servidor *chat* designado por **tcp_chat_server.py**.

Ele deverá funcionar com o cliente TCP criado anteriormente. Mas como o cliente simples não suporta a nova lógica, por vezes não lhe serão apresentadas as mensagens enviadas pelos outros clientes.

Exercício 3.8

Tendo em atenção a lógica do cliente *chat* para os *Sockets* UDP, implemente um cliente *chat* para *Sockets* TCP designado por **tcp_chat_client.py**.

Para testar o cliente proceda como foi explicado anteriormente para os *Sockets* UDP.

Altere a implementação do servidor de forma que o cliente não receba as mensagens por ele enviadas, recebendo apenas as mensagens dos outros clientes.

3.6 Para Aprofundar

Exercício 3.9

Implemente um programa que dado um endereço HyperText Transfer Protocol (HTTP)[9] como primeiro argumento e um nome de ficheiro como segundo argumento, crie uma ligação TCP e envie os cabeçalhos HTTP de forma a obter o recurso (ficheiro) indicado. A resposta deverá ser escrita para o ficheiro indicado no segundo argumento. (Este programa é um cliente HTTP simples.)

Glossário

CSS	Cascading Style Sheets
DHCP	Dynamic Host Configuration Protocol
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IPC	Inter Process Communication
IPTV	IP TeleVision
IPv4	Internet Protocol v4
IPv6	Internet Protocol v6
JS	JavaScript
P2P	Peer to Peer
TCP	Transmission Control Protocol (protocolo de transporte da Internet orientado à ligação)
UDP	User Datagram Protocol (protocolo de transporte da Internet sem ligação, orientado ao datagrama)
VoIP	Voice Over IP

Referências

[1] W3C. «HTML 4.01 Specification». (1999), URL: `http://www.w3.org/TR/1999/REC-html401-19991224/`.

[2] ECMA International, *Standard ECMA-262 – ECMAScript Language Specification*, Padrão, dez. de 1999. URL: `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.

[3] W3C. «Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification». (2001), URL: `http://www.w3.org/TR/2011/REC-CSS2-20110607/`.

[4] J. Postel, *Internet Protocol*, RFC 791 (Standard), Updated by RFC 1349, Internet Engineering Task Force, set. de 1981.

[5] S. Deering e R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460 (Draft Standard), Updated by RFCs 5095, 5722, 5871, Internet Engineering Task Force, dez. de 1998.

[6] J. Postel, *User Datagram Protocol*, RFC 768 (Standard), Internet Engineering Task Force, ago. de 1980.

[7] J. Postel, *Transmission Control Protocol*, RFC 793 (Standard), Updated by RFCs 1122, 3168, 6093, Internet Engineering Task Force, set. de 1981.

[8] R. Droms, *Dynamic Host Configuration Protocol*, RFC 2131 (Draft Standard), Updated by RFCs 3396, 4361, 5494, Internet Engineering Task Force, mar. de 1997.

[9] R. Fielding, J. Gettys, J. Mogul et al., *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616 (Draft Standard), Updated by RFCs 2817, 5785, 6266, Internet Engineering Task Force, jun. de 1999.