Laboratórios de Informática 2014/2015

DETI - Universidade de Aveiro

Docentes
João Paulo Barraca <jpbarraca@ua.pt>
Diogo Gomes <dgomes@ua.pt>
João Manuel Rodrigues <jmr@ua.pt>
Mário Antunes <mario.antunes@ua.pt>

TEMA 15

Comunicação entre aplicações

Objetivos:

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

15.1 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, sendo que este guião irá focar-se na geração, envio e recepção das mensagens pelas aplicações.

15.2 Conceitos de comunicação

15.2.1 Modelo Cliente-Servidor

Nas comunicações entre aplicações considera-se a existência de um modelo de comunicação onde se contemplam dois tipos de intervenientes: o cliente e o servidor. O Cliente é aquele que inicia a comunicação e requer informação, enquanto o servidor é aquele que aceita um pedido e realiza uma ação, podendo emitir uma resposta. Considerando o exemplo anterior, o navegador *Web* é um cliente que emite pedidos enquanto os servidores *Web* (e tal como o nome indica) são aqueles que aceitam pedidos dos clientes, fornecendo depois páginas.

A Figura 15.1 representa uma simples interacção entre vários clientes e um servidor.

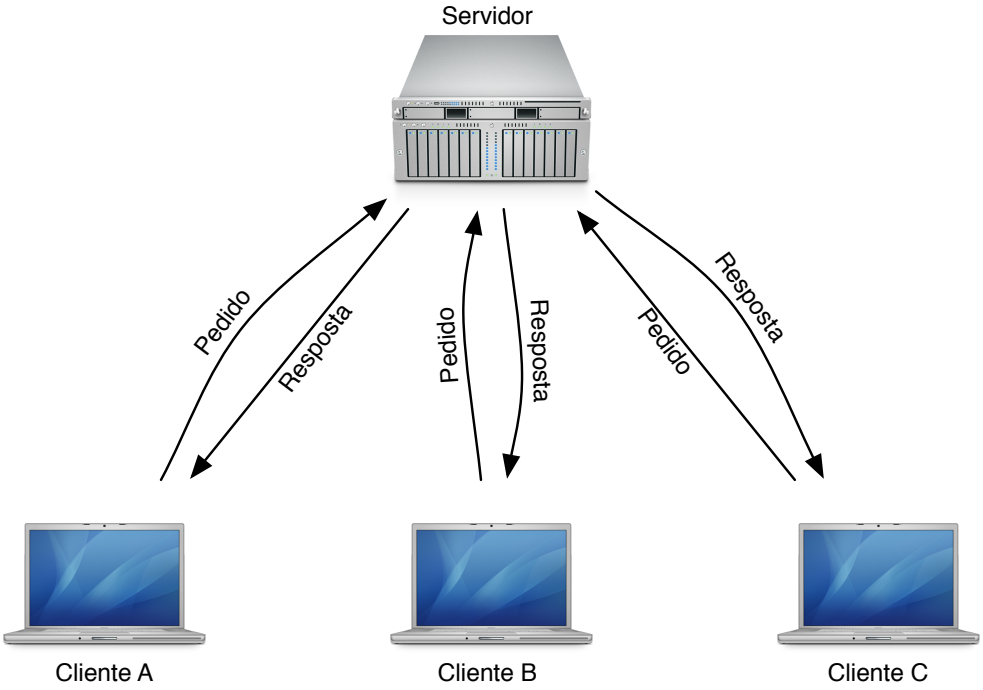

Figura 15.1: Modelo Cliente-Servidor

Esta separação de funções é importante pois as aplicações são desenvolvidas de acordo com uma das funções, sendo que o fluxo lógico (algoritmo) do cliente é diferente do fluxo lógico do servidor. Considerando o mesmo exemplo, os servidores têm de fornecer páginas *Web* e 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. Considera-se igualmente que vários clientes ligam-se 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 isto 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.

15.2.2 Sockets

As aplicações na consola podem interagir com o utilizador através de três conceitos básicos: `stdin`, `stdout` e `stderr`. Estes mecanismos têm sido extensivamente utilizados quando se lêem teclas do teclado ou se escreve para a consola. As aplicações também podem criar representações internas de ficheiros e diretórios, que depois podem ser utilizados para ler, escrever, criar ou apagar estes elementos. Do ponto de vista de comunicação entre processos, ou Inter Process Communication (IPC), consideram-se outros mecanismos como o *Socket*. Um *Socket* possui um significado de uma ficha onde se pode ligar algo, tal como uma ficha eléctrica ou uma ficha de rede. No âmbito das aplicações, um *Socket* é um mecanismos pelo qual as aplicações podem criar fichas "virtuais" para o qual outras aplicações pode realizar ligações. Uma aplicação pode criar quantos *Sockets* quiser, tal como uma casa pode ter quantas fichas forem necessárias. A Figura 15.2 demonstra uma aplicação com vários pontos de comunicação, sendo que alguns são *Sockets*.

Tal como uma ficha possui um dada especificação, do ponto de vista de forma a contactos, um *Socket* também possui algumas definições elementares, das quais se destacam a família e o tipo. Um *Socket* de uma dada família e tipo só irá permitir ligações com essas características e não com outras. Como família, podem ser considerados os seguintes valores (entre outros):

- **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.

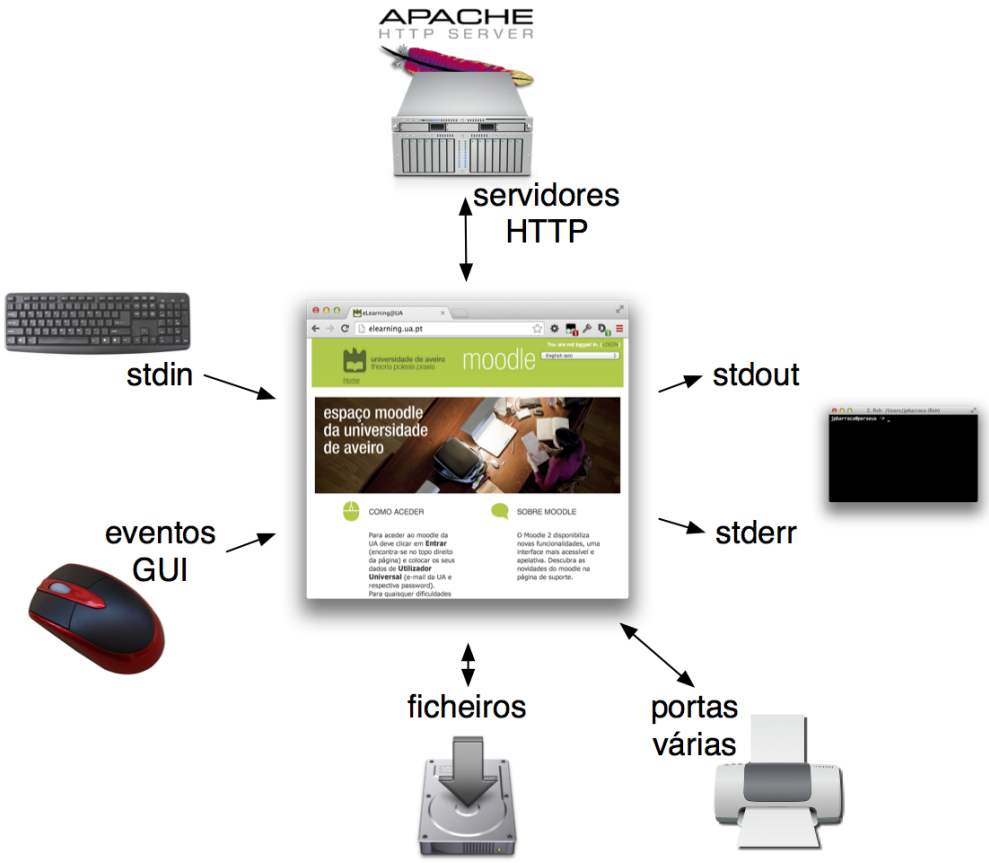

Figura 15.2: Vários pontos de comunicação com um navegador *Web*.

Neste caso, um *Socket* da família **AF_INET** podes 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. Consideram-se dois tipos relevantes:

- **SOCK_DGRAM**: As comunicações deverão utilizar o protocolo User Datagram Protocol (UDP)[6]. Com este tipo de *Socket*, é possível a perda ou reordenamento das mensagens.
- **SOCK_STREAM**: As comunicações deverão utilizar o protocolo Transmission

Control Protocol (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.

15.3 Sockets não orientados à ligação (UDP)

Um tipo de *Sockets* permite o envio de mensagens entre aplicações, sem que estas estabeleçam uma ligação explícita. Entre muitos outros cenários, são indispensáveis para envio de mensagens de *broadcast*, para sistemas com baixas capacidades computacionais, ou mesmo para comunicações de Voice Over IP (VoIP). Aplicações comuns que utilizam este tipo de *Sockets* são as redes IP TeleVision (IPTV) que temos em casa, as dedicadas a VoIP como o *Skype*, ou o 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, tal como uma ficha eléctrica não transmite energia de forma mágica para os dispositivos que temos em casa. É necessário um conjunto de primitivas ou acções realizadas sobre o *Socket* para que ele seja útil. No caso de uma ficha, é necessário ter um cabo correto, com a tomada correta e depois encaixar o cabo. No caso de um *Socket* são também definidas várias acções, nomeadamente:

- **Criação**: Um *Socket* é criado, definindo-se a família e tipo. Utiliza-se para isso a instrução `socket`.
- **Nomeação**: É necessário dar um nome ao *Socket*, usando-se a primitiva `bind`. Considerando que um sistema terá vários *Sockets* das várias aplicações, o nome identifica **unicamente** um dado *Socket*. As aplicações trocam informação a partir de e para um *Socket* em particular. Neste caso o nome é composto por um caminho, normalmente um endereço IPv4 e um porto (ou porta).
- **Enviar Informação**: Usa-se a acção de enviar (`send`) para enviar informação para o *Socket*, que será colocada numa memória à espera que a aplicação de destino a receba.
- **Receber Informação**: Usa-se a acção de receber (`receive`) para receber informação enviada para um *Socket* e presente na memória de receção.
- **Fechar**: O oposto da criação de um *Socket* efetivamente destruindo-o.

Estas primitivas são utilizadas da forma descrita na Figura 15.3. Como pode ser visto, neste tipo de *Sockets*, não existe uma declaração formal de que *Socket* é o servidor ou o cliente, mas a lógica das aplicações é diferenciada. Uma espera por pedidos, outra

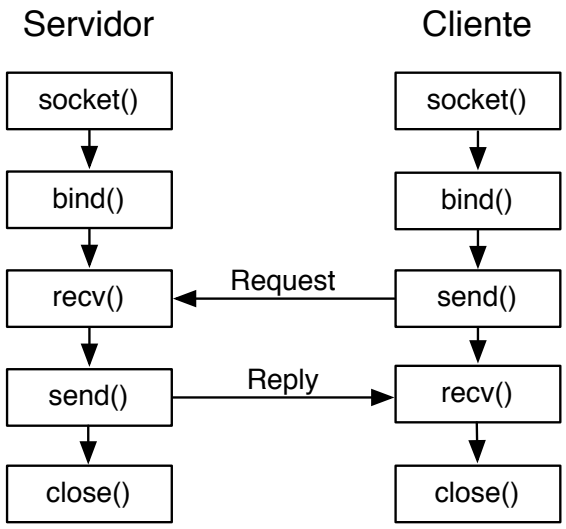

Figura 15.3: Sequência de primitivas utilizadas num *Socket* UDP.

efetua pedidos.

Utilizando *Python* é possível utilizar *Sockets* através da inclusão do módulo **`socket`**, sendo que as acções descritas anteriormente ficam disponíveis de forma imediata. Assim, um *Socket* pode ser criado e nomeado através das seguintes instruções:

```
# encoding=utf-8
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 especifica em que interface de rede o *Socket* comunicar, sendo que o valor `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. O exemplo seguinte 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[1].

```
...
def main():

    ...

    while 1:
        data, addr = udp_s.recvfrom(4096)
        udp_s.sendto(data.upper(), addr)

    udp_s.close()
...
```

Exercício 15.1

Utilizando os exemplos anteriores, implemente um servidor de mensagens UDP. Pode testar o servidor utilizando o comando **`nc -u localhost 1234`**.

O cliente para comunicar com esta aplicação, seria programado de maneira muito semelhante, sendo que teria as instruções de envio e recepção trocadas de ordem (primeiro envia e depois recebe a resposta). O trecho seguinte demonstra um cliente que lida uma frase do teclado a envia para o servidor, espera uma resposta e imprime-a.

```
...

def main():

    ...
    udp_s.bind(("127.0.0.1", 0))
    server_addr = ("127.0.0.1", 1234)
    while 1:
        data = raw_input("<-:")
```

[1] Ao contrário de *Java*, *Python* permite que a funções devolvam um número qualquer de valores.

```
        udp_s.sendto(data, server_addr)
        data, addr = udp_s.recvfrom(4096)

        print "->: %s \n" % data

    udp_s.close()
```

...

Exercício 15.2

Utilizando os exemplos fornecidos implemente um cliente 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.

Exercício 15.3

Coordene com o grupo ao seu lado de forma a executarem um servidor com um *Socket* no endereço `0.0.0.0`. Deverá conseguir enviar mensagens para este servidor se o seu cliente se ligar ao endereço do computador onde reside o servidor.

15.4 Sockets orientados à ligação (TCP)

Além destas primitivas, um *Socket* que seja 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 15.4. Como pode ser visto, neste tipo de *Sockets*, **existe** uma declaração formal de que *Socket* é servidor

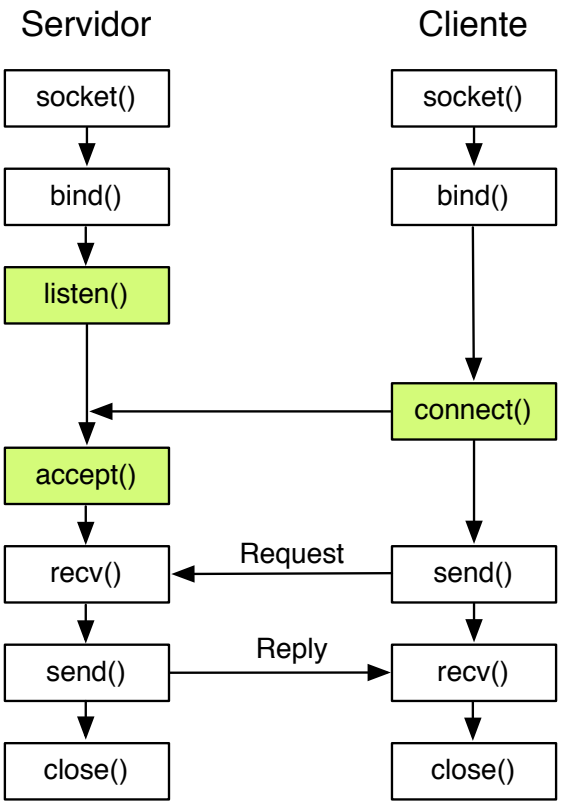

Figura 15.4: Sequência de primitivas utilizadas num *Socket* TCP.

ou cliente e lógica das aplicações é também diferenciada. Uma aplicação espera por pedidos, outra efetua pedidos. A cor são indicadas as primitivas adicionais, necessárias ao estabelecimento da sessão.

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:

```
# encoding=utf-8
import socket

def main():
    tcp_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    tcp_s.bind( ("127.0.0.1", 1234) )

main()
```

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. De notar 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.

```
...

def main():

    ...

    # máximo de 1 cliente à espera de
    # aceitação
    tcp_s.listen(1)

    # esperar por novos clientes
    client_s, client_addr = tcp_s.accept()

    while 1:
        data = client_s.recv(4096)
        client_s.send(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 15.4

Implemente um servidor TCP utilizando o exemplo fornecido. Pode testar o servidor utilizando o comando **`telnet localhost 1234`**, ou o comando **`nc localhost 1234`**.

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

```
...

def main():
    ...

    # Ligar ao servidor
    tcp_s.connect( ("127.0.0.1", 1234) )
    while 1:
        data = raw_input("Mensagem: ")
        tcp_s.send(data)
        data = tcp_s.recv(4096)
        print 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 15.5

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

15.5 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 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.

```
...

def main():

    ...

    # Lista de sockets conhecidos
    addr_list = []

    while 1:
        data, addr = udp_s.recvfrom(4096)
        print data

        # 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(data.upper(), dst_addr)
```

Exercício 15.6

Implemente um servidor como o indicado no exemplo. Este deverá funcionar com os clientes UDP criados anteriormente. No entanto o cliente não irá suportar a nova lógica, pelo que não serão apresentadas as mensagens de todos os clientes.

Um cliente de *chat* também necessitaria de pequenas alterações de forma 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.

De notar que o método **`select`** aceita três parâmetros: a lista de *Sockets* onde se espera por dados, a lista de *Sockets* onde recentemente foram escritos dados e se espera que estes sejam transmitidos, 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 é interessante a primeira das listas, indicando

esta que *Sockets* possuem informação pronta a ser consumida.

```
import select
...

def main():
    ...

    while 1:
        rsocks = select.select([udp_s, sys.stdin, ], [], [])[0]

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

...
```

Exercício 15.7

Implemente um cliente de *chat* de forma a que possa dialogar com os outros utilizadores ligados ao mesmo servidor.

Exercício 15.8

Implemente o mesmo cliente e servidor mas utilizando TCP. Neste caso tenha em consideração que o servidor irá possuir um *Socket* por cliente, armazenando estes *Sockets* na lista de clientes (e não o endereço e porta utilizados pelo cliente).

15.6 Para Aprofundar

Exercício 15.9

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.

Exercício 15.10

Implemente um programa que escute por mensagens no *Socket* TCP no formato:

```
GET /dir/fname
```

O programa deve responder com o conteúdo do ficheiro que se encontra no local indicado. Considere que os ficheiros pedidos encontram-se em diretórios e subdiretórios de um diretório raiz específico. A título de exemplo, se o diretório raiz for `/tmp` e for pedido o ficheiro `/labi/aula/a.txt`, o programa deve fornecer o conteúdo do ficheiro `/tmp/labi/aula/a.txt`.

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

UDP User Datagram Protocol

VoIP Voice Over IP

Referências

[1] W3C. (1999). Html 4.01 specification, endereço: `http://www.w3.org/TR/1999/REC-html401-19991224/`.

[2] ECMA International, *Standard ecma-262 – ecmascript language specification*, Padrão, dez. de 1999. endereço: `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.

[3] W3C. (2001). Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, endereço: `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] ——, *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, H. Frystyk, L. Masinter, P. Leach e T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616 (Draft Standard), Updated by RFCs 2817, 5785, 6266, Internet Engineering Task Force, jun. de 1999.