Reti & Socket

Concetti TCP/IP, API di Berkeley/WinSock e esempi in C/Java/Python

Corso: Sistemi di Calcolo 2
Docente: Riccardo Lazzeretti
Slide basate su: W. Stallings, *Operating Systems: Internals and Design Principles*, cap. 17

Indice

1 Richiami di rete e architettura di protocollo **2**
1.1 Perch
'e serve un'architettura di protocollo . . . 2
1.2 Protocollo: definizione ed elementi . . . 2
1.3 Architettura TCP/IP a 5 livelli . . . 2

2 Concetti TCP/IP fondamentali **2**
2.1 Indirizzamento a due livelli . . . 2
2.2 Header TCP e IP . . . 3
2.3 Applicazioni standard sopra TCP . . . 3

3 Socket: interfaccia e modelli **3**
3.1 Concetto di socket . . . 3
3.2 Servizi verso gli strati superiori . . . 3
3.3 Il "socket" come coppia (IP, porta) . . . 3
3.4 Approccio client–server e porte . . . 3

4 API di Berkeley Sockets (C) **3**
4.1 Strutture di indirizzo IPv4 . . . 3
4.2 Creazione del socket: `socket()` . . . 4
4.3 Assegnazione indirizzo locale: `bind()` . . . 4
4.4 Argomenti *valore-risultato* (dalle slide) . . . 4
4.5 Server TCP: `listen()` e `accept()` . . . 4
4.6 Concorrenza: `accept()+fork()` . . . 4
4.7 Chiusura: `close()` e `SO_LINGER` . . . 4
4.8 Client TCP: `connect()` . . . 5
4.9 Invio/ricezione TCP . . . 5
4.10 UDP: API senza connessione . . . 5
4.11 Risoluzione nomi e conversioni . . . 5

5 Esempi in Java **5**
5.1 Stream sockets (server e client) . . . 5
5.2 UDP in Java . . . 6

6 Esempi in Python **6**
6.1 TCP (stream sockets) . . . 6
6.2 UDP . . . 7

7 Adaptor di rete e driver (richiami dalle slide) 7
7.1 Architettura di un network adaptor . . . 7
7.2 Vista dal sistema host . . . 8
7.3 DMA vs Programmed I/O . . . 8
7.4 Buffer Descriptor (BD) list . . . 8
7.5 Viaggio di un messaggio nello stack . . . 8
7.6 Schema driver: richiesta di trasmissione . . . 8
7.7 Interrupt handler (caso trasmissione completata) . . . 8

8 WinSock: differenze principali 9
8.1 Berkeley vs WinSock . . . 9
8.2 Modalità WinSock 1.1 . . . 9
8.3 WinSock 2 . . . 9

9 Note finali 9

1 Richiami di rete e architettura di protocollo

1.1 Perch 'e serve un'architettura di protocollo

Lo scambio di dati tra calcolatori richiede più procedure: instaurare un percorso (diretto o via rete), annunciare l'identità del destinatario, verificare la prontezza del destinatario, coordinare applicazioni (es. trasferimento file) e tradurre formati se necessario. Serve cooperazione ad alto grado tra i sistemi, ottenuta scomponendo i compiti in **sottolivelli** con interfacce ben definite (*protocol architecture*).

1.2 Protocollo: definizione ed elementi

Un **protocollo** è un insieme di regole che governa lo scambio dati tra due entità (potenzialmente su sistemi diversi). Elementi chiave: **sintassi** (formato dati, livelli di segnale), **semantica** (informazioni di controllo per coordinamento e gestione errori), **timing** (sequenziamento e velocità).

1.3 Architettura TCP/IP a 5 livelli

Applicazione, **Trasporto** (host-to-host), **Internet** (IP), **Accesso alla rete**, **Fisico**. L'IP è implementato anche nei router; il Trasporto (TCP/UDP) realizza affidabilità e multiplex/demultiplex tra applicazioni.

2 Concetti TCP/IP fondamentali

2.1 Indirizzamento a due livelli

Ogni entità deve avere un indirizzo univoco. Due livelli:

1. **Indirizzo Internet (IP)** univoco per ogni host; usato da IP per instradare.
2. **Porte** univoche per processo/applicazione nell'host; usate dal Trasporto (TCP/UDP) per consegnare al processo giusto.

2.2 Header TCP e IP

Descrizione testuale: layout dell'intestazione TCP (porte sorgente/destinazione, numeri di sequenza/ack, flags, finestra, checksum) e dell'intestazione IPv4 (versione, IHL, ToS/DS, lunghezza, ID/flag/offset, TTL, protocollo, checksum, indirizzi).

2.3 Applicazioni standard sopra TCP

Esempi: **SMTP** (mail, liste, ricevute, forward), **FTP** (trasferimento file, accesso controllato), **SSH** (login remoto sicuro, file transfer). **UDP**: protocollo minimale, senza connessione, nessuna garanzia su consegna/ordine/duplicati; utile per transazioni e multicast.

3 Socket: interfaccia e modelli

3.1 Concetto di socket

Interfaccia originata nell'ambiente UNIX (Berkeley Sockets Interface, BSI); è un **endpoint** di comunicazione. Tipi: **stream** (TCP), **datagram** (UDP), **raw**. Standard de facto anche su Windows (WinSock).

3.2 Servizi verso gli strati superiori

La *socket API* fornisce servizi di comunicazione all'applicazione, mascherando dettagli di rete/-trasporto.

3.3 Il "socket" come coppia (IP, porta)

La concatenazione di *indirizzo IP* e *porta* forma un **socket** univoco nell'Internet. Una connessione è identificata da una *coppia di socket* (*socket pair*).

3.4 Approccio client–server e porte

Modello **request/response** con il server in ascolto su **porta ben nota**. Classi di porte (BSD/Linux/Solaris):

- **0–1023**: riservate (privilegiate).
- **1024–∼5000**: *ephemeral* (assegnate automaticamente ai client; sui sistemi moderni l'intervallo effettivo può essere esteso, es. decine di migliaia).
- **/etc/services**: mappa servizi/porte (es. `ftp 21/tcp`, `telnet 23/tcp`, `finger 79/tcp`, `snmp 161/udp`).

Alcuni client legacy necessitano porte privilegiate (`512-1024`) per meccanismi di autenticazione (`rlogin`, `rsh`).

4 API di Berkeley Sockets (C)

4.1 Strutture di indirizzo IPv4

```
1 struct in_addr {
2   in_addr_t s_addr;        /* IPv4 (32 bit, network byte order) */
3 };
4 struct sockaddr_in {
5   uint8_t        sin_len;    /* lunghezza struttura (BSD) */
6   sa_family_t    sin_family; /* AF_INET */
```

```
  in_port_t      sin_port;   /* porta TCP/UDP (network byte order) */
  struct in_addr sin_addr;   /* indirizzo IPv4 */
  char           sin_zero[8];/* inutilizzato (padding) */
};
/* Inizializzare a zero (bzero/memset) prima dell'uso. */
```

4.2 Creazione del socket: `socket()`

```
int socket(int family, int type, int protocol);
/* family: AF_INET/AF_INET6/AF_LOCAL/...; type: SOCK_STREAM/SOCK_DGRAM/SOCK_RAW;
   protocol: 0 (di solito) o specifico; ritorna descrittore o -1. */
```

4.3 Assegnazione indirizzo locale: `bind()`

```
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
/* Se port=0, l'OS sceglie una porta effimera. EADDRINUSE se occupata. */
```

4.4 Argomenti *valore-risultato* (dalle slide)

Per chiamate che passano strutture indirizzo *dal kernel all'utente* (es. `accept()`), si passa un puntatore alla struttura *e* un puntatore a un intero che contiene la dimensione disponibile: il kernel scrive fino a quel limite e, al ritorno, aggiorna la dimensione effettiva scritta. Per le chiamate *dall'utente al kernel* (es. `bind()`), si passa il puntatore e la dimensione (`sizeof`) in modo che il kernel sappia quanti byte copiare.

4.5 Server TCP: `listen()` e `accept()`

```
int listen(int sockfd, int backlog);        /* massimo numero di connessioni
    incomplete */
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
/* accept() crea un nuovo descrittore per la connessione stabilita */
```

4.6 Concorrenza: `accept()+fork()`

```
lisfd = socket(...);
bind(lisfd, ...);
listen(lisfd, 5);
while (1) {
  connfd = accept(lisfd, ...);
  if ((pid = fork()) == 0) {
    close(lisfd); doit(connfd); close(connfd); _exit(0);
  }
  close(connfd);
}
```

Descrizione: dopo il `fork()`, il figlio usa `connfd` per servire il client; il padre richiude `connfd` e torna ad `accept()`.

4.7 Chiusura: `close()` e `SO_LINGER`

```
int close(int sockfd);   /* chiude per lettura/scrittura; tenta d'inviare i dati
    pendenti */
```

Opzione `SO_LINGER`: bloccare fino all'invio, oppure scartare residui.

4.8 Client TCP: `connect()`

```c
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
/* Il client non ha bisogno di bind(): l'OS assegna IP/porta locali effimeri. */
```

4.9 Invio/ricezione TCP

```c
int send(int sockfd, const void *buf, size_t n, int flags);
int recv(int sockfd, void *buf, size_t n, int flags);
/* Flag comuni: MSG_DONTWAIT, MSG_OOB, MSG_PEEK, MSG_WAITALL, MSG_DONTROUTE */
```

4.10 UDP: API senza connessione

```c
int sendto(int sockfd, const void *buf, size_t n, int flags,
           const struct sockaddr *to, socklen_t addrlen);
int recvfrom(int sockfd, void *buf, size_t n, int flags,
             struct sockaddr *from, socklen_t *addrlen);
```

Nessun *handshake*, nessuna *close* simultanea, e i server concorrenti non usano `fork()` come per TCP.

4.11 Risoluzione nomi e conversioni

```c
int gethostname(char *hostname, int buflen);
unsigned long inet_addr(const char *cp);   /* puntinata -> 32 bit (deprecata: usare
    inet_pton) */
char *inet_ntoa(struct in_addr in);        /* 32 bit -> puntinata (non thread-safe;
    usare inet_ntop) */
struct hostent *gethostbyname(const char *name);   /* legacy; usare getaddrinfo */
struct hostent *gethostbyaddr(const void *addr, int len, int type);
```

5 Esempi in Java

5.1 Stream sockets (server e client)

```java
// Server minimale (solo per demo: mancano gestione errori e loop)
public class GreetServer {
  private ServerSocket serverSocket;
  private Socket clientSocket;
  private PrintWriter out;
  private BufferedReader in;
  public void start(int port) throws Exception {
    serverSocket = new ServerSocket(port);
    clientSocket = serverSocket.accept();
    out = new PrintWriter(clientSocket.getOutputStream(), true);
    in  = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
    String greeting = in.readLine();
    if ("hello server".equals(greeting)) out.println("hello client");
    else out.println("unrecognised greeting");
  }
  public void stop() throws Exception {
    in.close(); out.close(); clientSocket.close(); serverSocket.close();
  }
```

```
  public static void main(String[] args) throws Exception {
    GreetServer s = new GreetServer(); s.start(6666); s.stop();
  }
}
```

```
// Client minimale
public class GreetClient {
  private Socket clientSocket;
  private PrintWriter out;
  private BufferedReader in;
  public void startConnection(String ip, int port) throws Exception {
    clientSocket = new Socket(ip, port);
    out = new PrintWriter(clientSocket.getOutputStream(), true);
    in  = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
  }
  public String sendMessage(String msg) throws Exception {
    out.println(msg);
    return in.readLine();
  }
  public void stopConnection() throws Exception {
    in.close(); out.close(); clientSocket.close();
  }
  public static void main(String[] args) throws Exception {
    GreetClient c = new GreetClient();
    c.startConnection("127.0.0.1", 6666);
    String response = c.sendMessage("hello server");
    System.out.println(response);
    c.stopConnection();
  }
}
```

Nota: un server "ben fatto" include generazione di processi/thread, ascolto continuo e comunicazioni ripetute.

5.2 UDP in Java

```
// UDP Server
DatagramSocket serverSocket = new DatagramSocket(9876);
byte[] receiveData = new byte[1024], sendData;
while (true) {
  DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
  serverSocket.receive(receivePacket);
  String sentence = new String(receivePacket.getData());
  InetAddress IPAddress = receivePacket.getAddress();
  int port = receivePacket.getPort();
  String capitalized = sentence.toUpperCase();
  sendData = capitalized.getBytes();
  DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress,
     port);
  serverSocket.send(sendPacket);
}
```

6 Esempi in Python

6.1 TCP (stream sockets)

```python
# server.py
import socket
HOST, PORT = '127.0.0.1', 65432
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT)); s.listen()
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.sendall(data)
s.close()
```

```python
# client.py
import socket
HOST, PORT = '127.0.0.1', 65432
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall(b'Hello, world')
data = s.recv(1024)
print('Received', repr(data))
s.close()
```

6.2 UDP

```python
# udp_server.py
import socket
UDP_IP, UDP_PORT = "127.0.0.1", 5005
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((UDP_IP, UDP_PORT))
while True:
    data, addr = sock.recvfrom(1024)
    print("received message:", data)
    sock.sendto(data, addr)
```

```python
# udp_client.py
import socket
UDP_IP, UDP_PORT = "127.0.0.1", 5005
MESSAGE = b"Hello, World!"
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(MESSAGE, (UDP_IP, UDP_PORT))
data, addr = sock.recvfrom(1024)
sock.close()
```

7 Adaptor di rete e driver (richiami dalle slide)

7.1 Architettura di un network adaptor

Descrizione: l'adaptor ha una parte verso l'host (bus/CSR) e una verso la rete (livello fisico/-collegamento), con FIFO tra le due per mascherare l'asincronia. Una **SCO** (sottosistema di controllo) governa l'insieme.

7.2 Vista dal sistema host

L'adaptor esporta registri **CSR** (Control Status Register). Esempio (leggenda: RO, RC, W1, RW, RW1): bit come `LE_RINT` (richiesta interruzione per ricezione), `LE_TINT` (trasmissione completata), `LE_INEA` (abilitazione interrupt), `LE_TDMD` (richiesta di trasmissione).
Modalità di gestione: *busy waiting* oppure *interrupt*.

7.3 DMA vs Programmed I/O

DMA: CPU non coinvolta nel trasferimento; OS alloca aree di memoria; frame scritti direttamente in memoria host. **Programmed I/O**: il trasferimento passa per la CPU; serve buffering sull'adaptor; possibile memoria *dual-port*.

7.4 Buffer Descriptor (BD) list

Memoria organizzata come vettore di descrittori che puntano a *buffers*. Tecniche: *scatter read / gather write*. In Ethernet tipicamente 64 buffer da 1500 B preallocati.

7.5 Viaggio di un messaggio nello stack

1. Il SO copia il messaggio dal buffer utente ad un BD.
2. Il messaggio attraversa gli strati (TCP/IP/driver), che aggiungono header e aggiornano puntatori nel BD.
3. Il driver segnala alla SCO usando i bit `LE_TDMD`/`LE_INEA`.
4. La SCO trasmette.
5. A fine trasmissione, la SCO setta `LE_TINT` e genera un'interruzione.
6. L'interrupt handler azzera i bit, libera risorse e sblocca eventuali processi (es. `semsignal` su `xmit_queue`).

7.6 Schema driver: richiesta di trasmissione

```
#define csr ((u_int)0xffff3579) /* indirizzo CSR */
Transmit(Msg *msg) {
  descriptor *d;
  semwait(xmit_queue);
  d = next_desc();
  prepare_desc(d, msg);
  semwait(mutex);
  disable_interrupts();
  /* Invita la SCO a trasmettere, abilita gli interrupt */
  csr = LE_TDMD | LE_INEA;
  enable_interrupts();
  semsignal(mutex);
}
```

7.7 Interrupt handler (caso trasmissione completata)

- Disabilita interruzioni; legge **CSR** per capire l'origine (errore, TX completata, RX arrivata).
- Per TX: reset di `LE_TINT` (bit RC), ammette un nuovo processo nel BD, riabilita interruzioni.

8 WinSock: differenze principali

8.1 Berkeley vs WinSock

Berkeley	WinSock
`bzero()`	`memset()`
`close()`	`closesocket()`
`ioctl()`	`ioctlsocket()`
`read()/write()`	(non richieste, usare `recv()/send()`)

8.2 Modalità WinSock 1.1

Blocking (come Berkeley), **Non-blocking** (le chiamate ritornano subito), **Asynchronous** (notifiche via messaggi Windows: `FD_ACCEPT`, `FD_CONNECT`, ...).

8.3 WinSock 2

Supporto per altri stack (DecNet, IPX/SPX, OSI), applicazioni indipendenti dal protocollo, retrocompatibilità. Cambi di API (`WSAAccept()`, `WSAConnect()`, `WSAAddressToString()`, ...).

9 Note finali

Queste note sintetizzano i concetti e le API fondamentali per programmare applicazioni di rete con socket in C/Java/Python, e forniscono un richiamo sull'hardware di rete e sui driver, come presentato nelle slide originali.