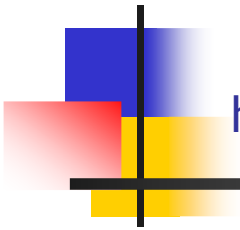
Reti di Calcolatori
AA 2011/2012

UNIVERSITÀ DEGLI STUDI DI TRENTO

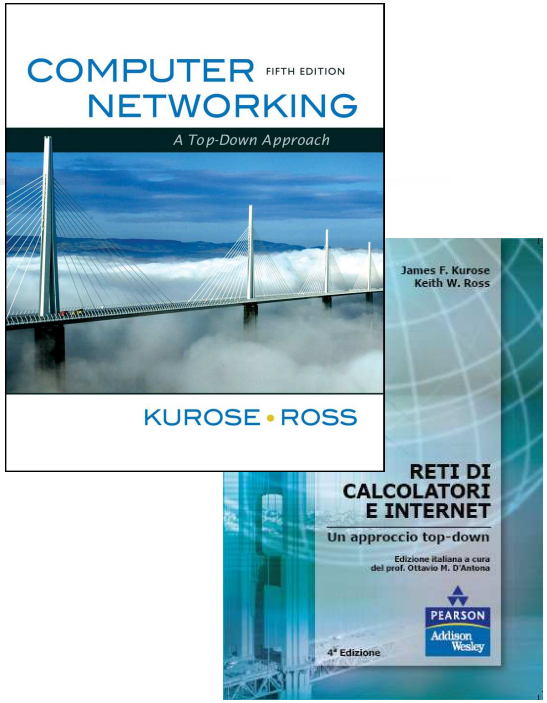


http://disi.unitn.it/locigno/index.php/teaching-duties/computer-networks

Programmazione delle socket

Csaba Kiraly
Renato Lo Cigno

Programmazione delle socket



A note on the use of these slides:

The first part of these slides are an adaptation from the freely available version provided by the book authors to all (faculty, students, readers). The originals are in PowerPoint and English.

The Italian translation is originally form
Gianluca Torta, Stefano Leonardi, Francesco Di Tria

The second part of these slides is from Dan Rubenstein and Vassilis Stachtos.

Adaptation is by Csaba Kiraly and Renato Lo Cigno

Computer Networking: A Top Down Approach,
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

Reti di calcolatori e Internet: Un approccio top-down
4ª edizione
Pearson Paravia Bruno Mondadori Spa
©2008

{kiraly,locigno}@disi.unitn.it

Programmazione delle socket

Obiettivo: imparare a costruire un'applicazione client/server che comunica utilizzando le socket

Socket API

- Introdotto in BSD4.1 UNIX, 1981
- esplicitamente creata, usata, rilasciata dalle applicazioni
- due tipi di servizio di trasporto tramite una socket API:
  - datagramma inaffidabile
  - affidabile, orientata ai byte

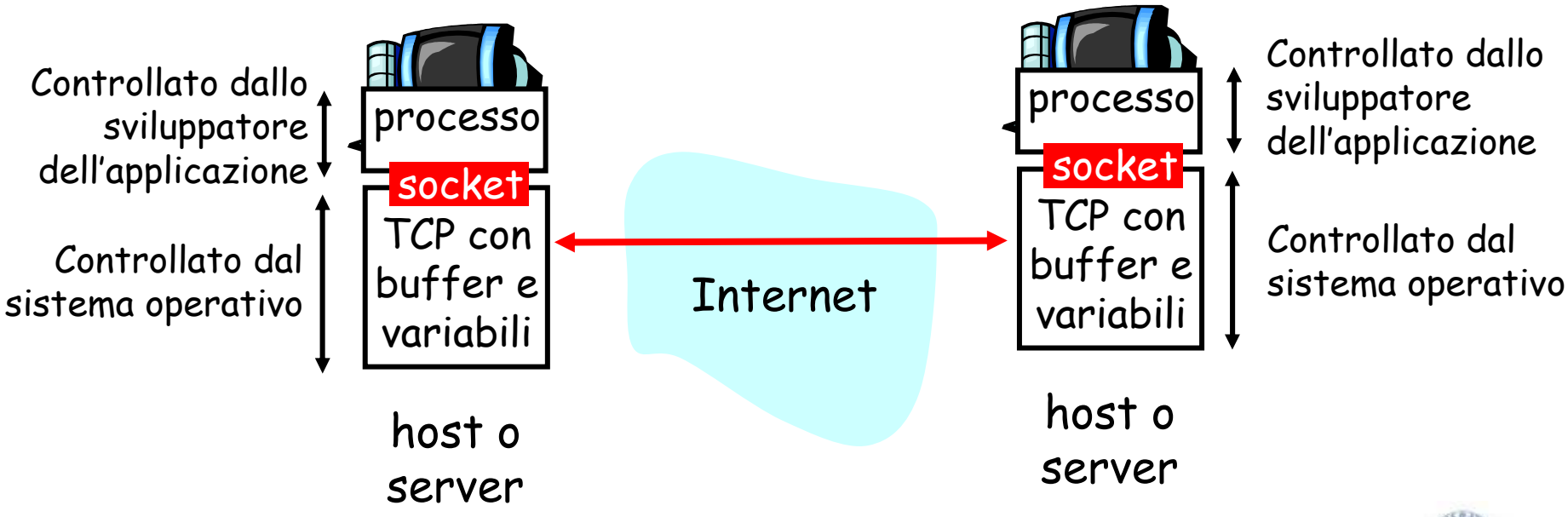
socket

Interfaccia
locale,
fra applicazione
e SO
(una "porta") in cui
il processo di
un'applicazione può
inviare e ricevere
messaggi al/dal processo
di un'altra applicazione

Programmazione delle socket con TCP

Socket: una porta tra il processo di un'applicazione e il protocollo di trasporto end-end (UDP o TCP)

Servizio TCP: trasferimento affidabile di **byte** da un processo all'altro



Programmazione delle socket con TCP

Il client deve contattare il server

- Il processo server deve essere in corso di esecuzione
- Il server deve avere creato una socket (porta) che dà il benvenuto al contatto con il client
- IP e porta deve essere noto al client

Il client contatta il server:

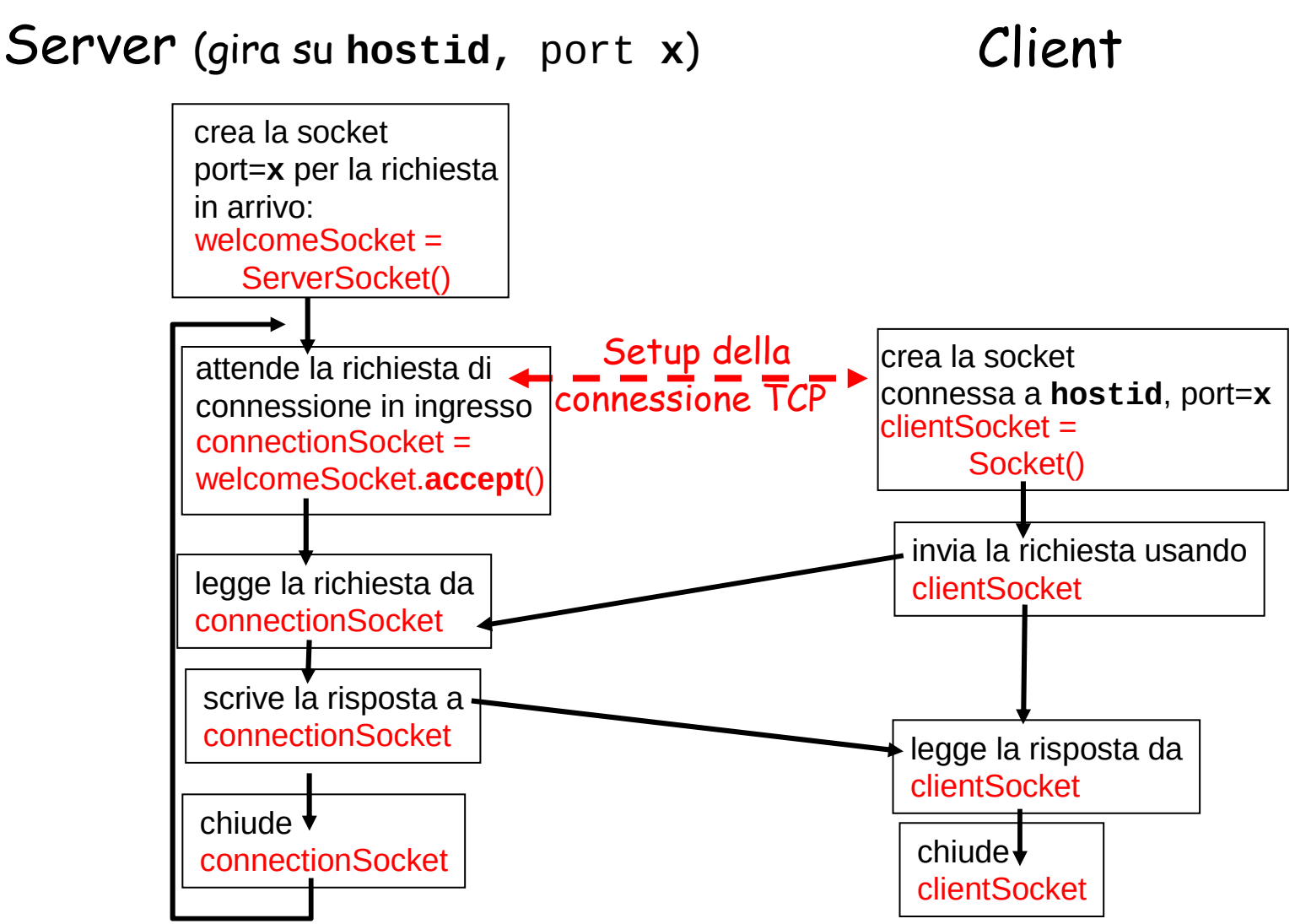
- Creando una socket TCP
- Specificando l'indirizzo IP, il numero di porta del processo server
- Quando il client crea la socket: il client TCP stabilisce una connessione con il server TCP

- Quando viene contattato dal client, il server TCP crea una nuova socket per il processo server per comunicare con il client
  - consente al server di comunicare con (distinguere) più client
- Come vengono differenziati i client?
  - Nel processo: socket diversi
  - "sotto": diversi 5-upla (ip origine, numeri do porta origine)

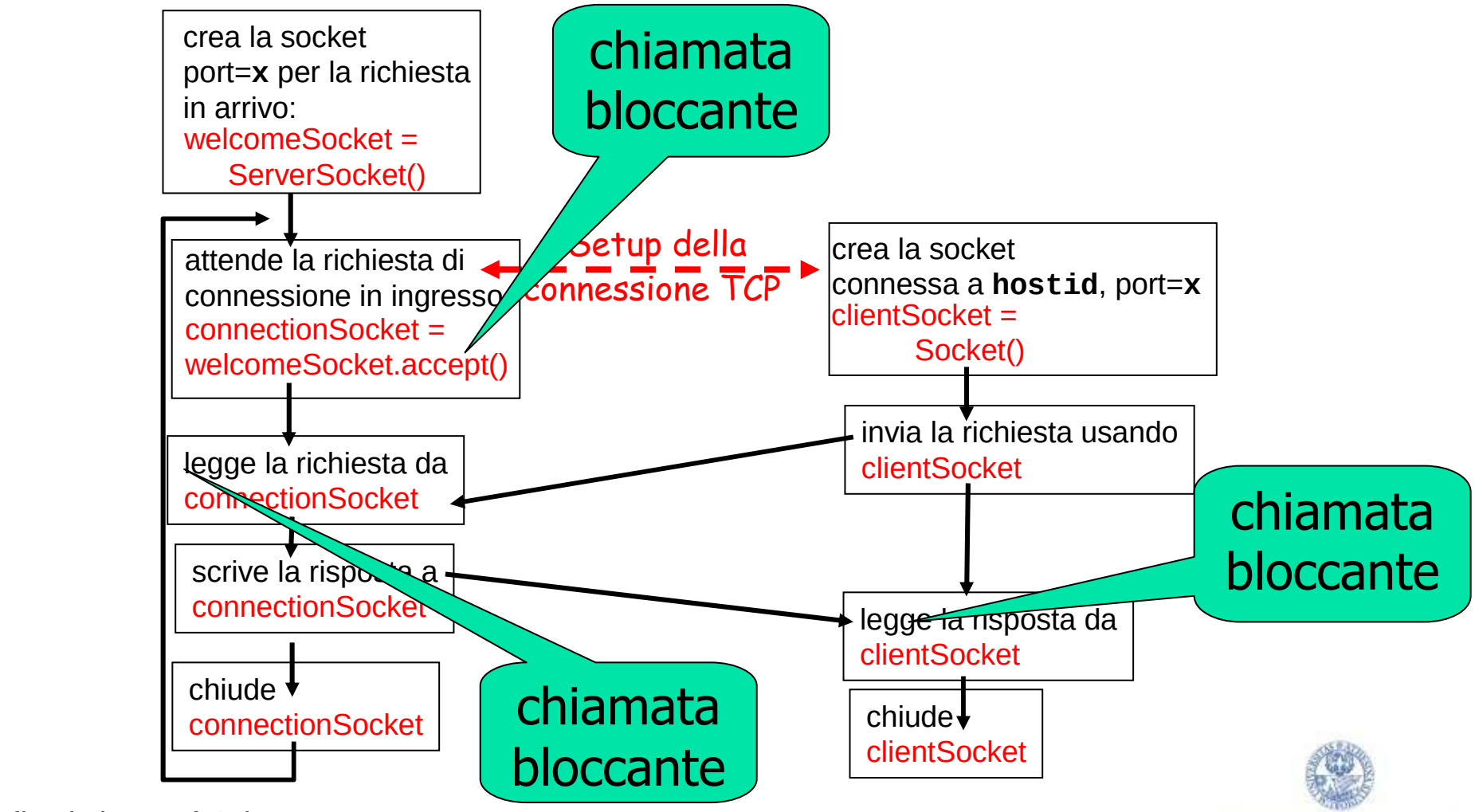
Punto di vista dell'applicazione

TCP fornisce un trasferimento di byte affidabile e ordinato ("pipe") tra client e server

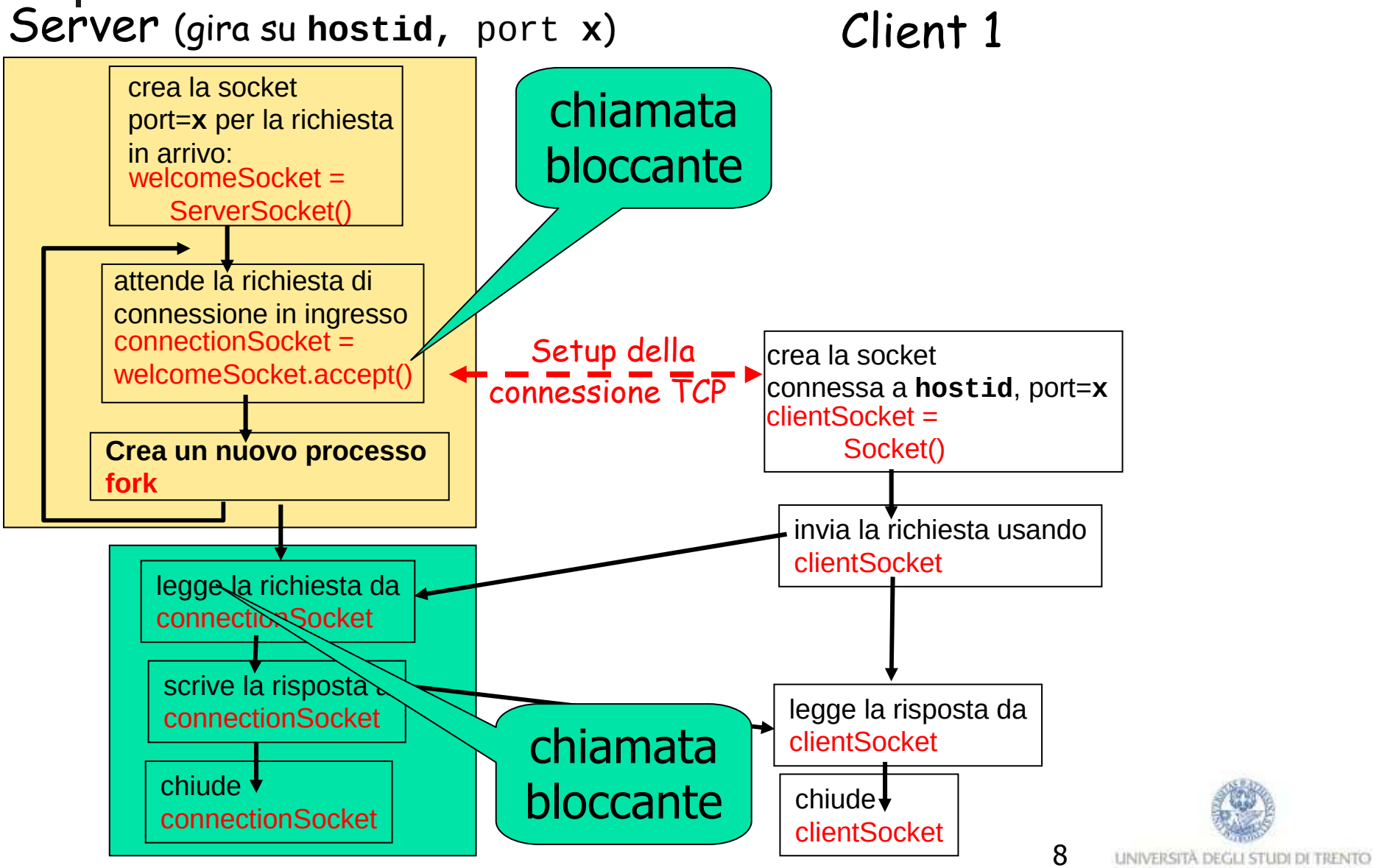
Interazione delle socket client/server: TCP



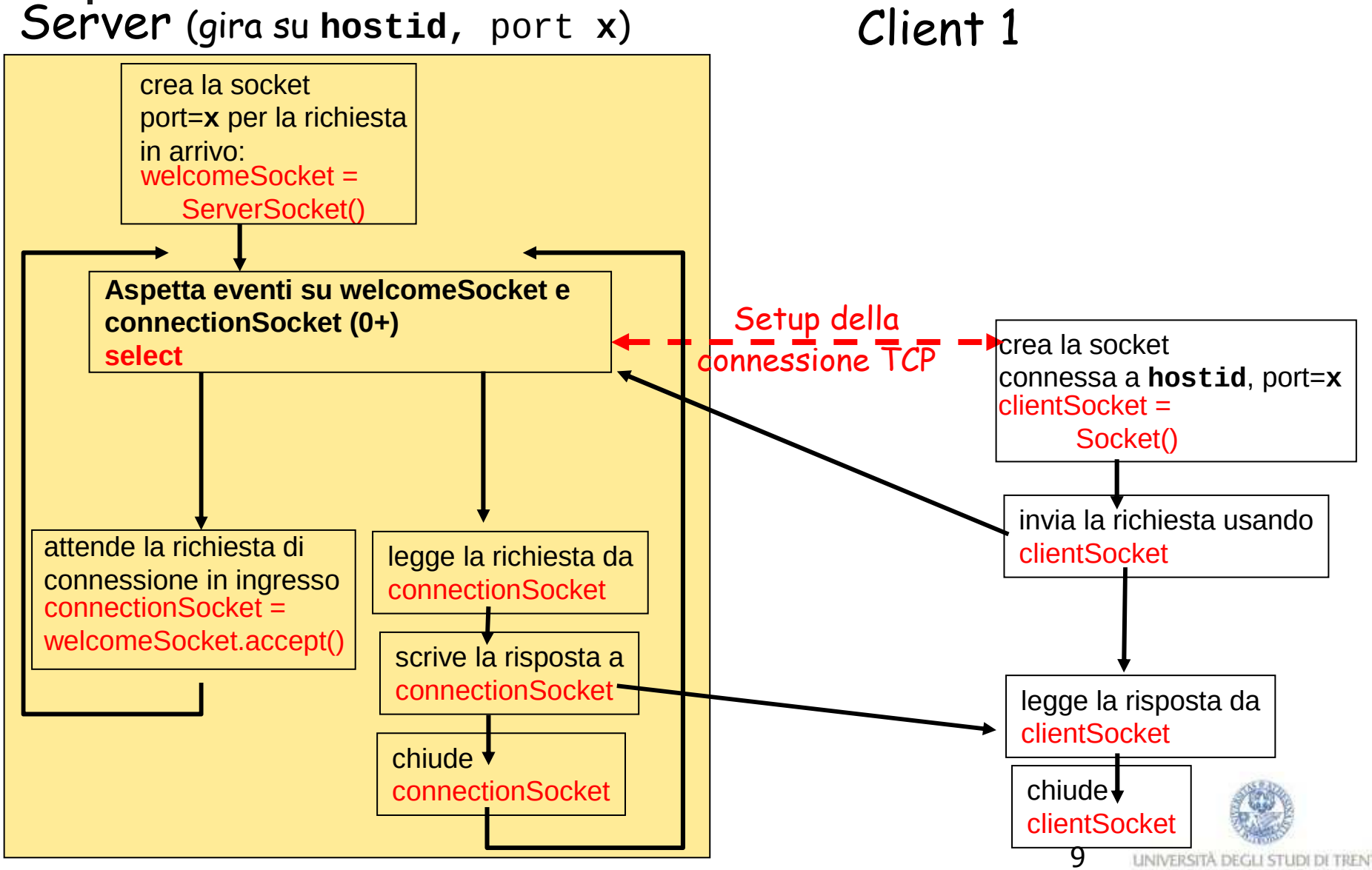
Interazione delle socket client/server: TCP
problema: chiamate bloccante



Interazione delle socket client/server: TCP
versione multi-thread



Interazione delle socket client/server: TCP
versione single-thread



Programmazione delle socket *con UDP*

UDP: non c'è "connessione" tra client e server

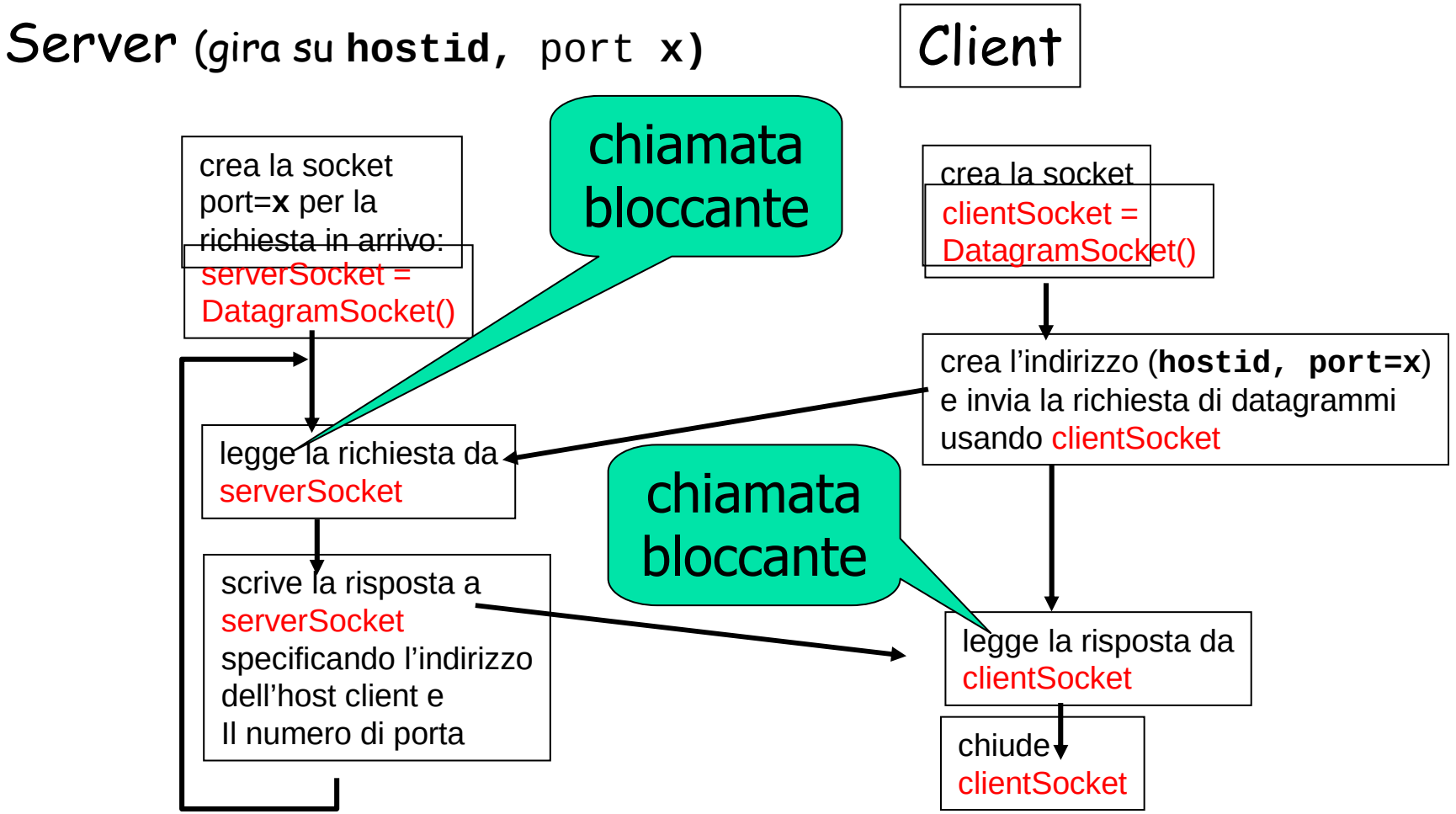
- Non c'è handshaking
- Il mittente allega esplicitamente ad ogni *messaggio* l'indirizzo della destinazione (IP e porta)
- Il server deve estrarre l'indirizzo IP e la porta del mittente dal pacchetto ricevuto

UDP: i dati trasmessi possono perdersi o arrivare a destinazione in un ordine diverso da quello d'invio

Punto di vista dell'applicazione

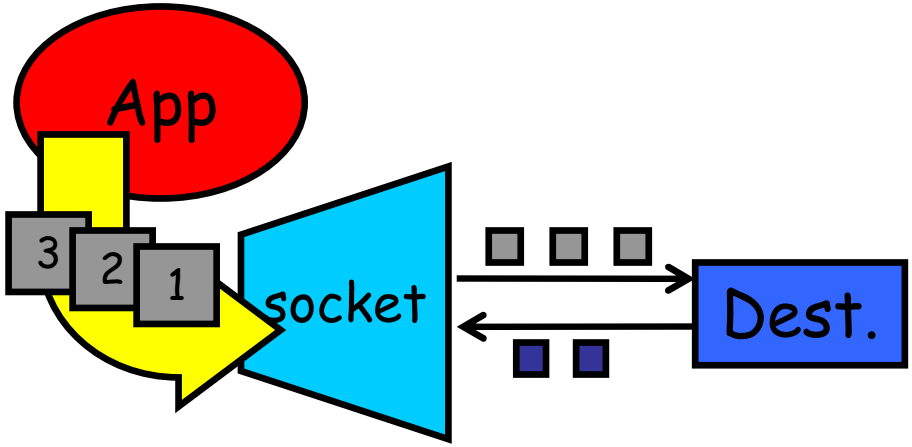
UDP fornisce un trasferimento inaffidabile di gruppi di byte ("datagrammi") tra client e server

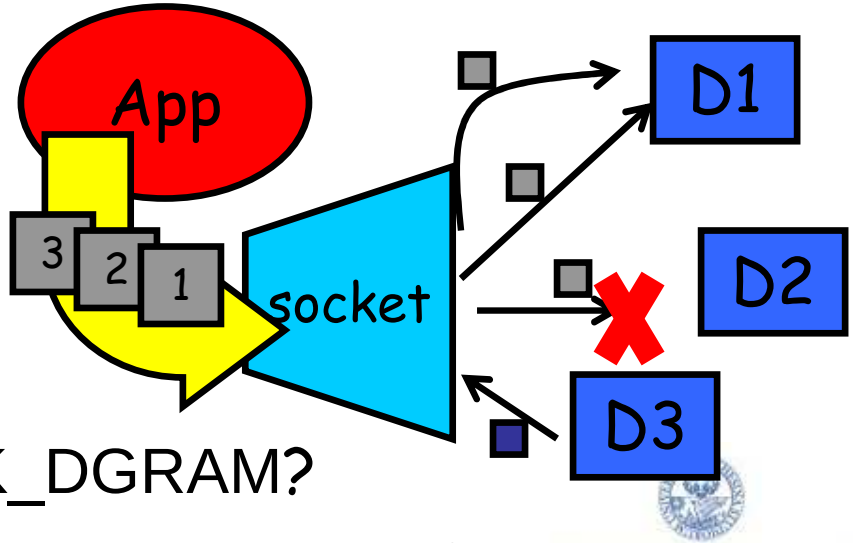
Interazione delle socket client/server: UDP



Two essential types of sockets

- SOCK_STREAM
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional
- SOCK_DGRAM
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of "connection" – app indicates dest. for each packet
  - can send or receive





Q: why have type SOCK_DGRAM?

Socket Creation in C: socket

- int s = socket(domain, type, protocol);
  - s: socket descriptor, an integer (like a file-handle)
  - domain: integer, communication domain
    - e.g., PF_INET (IPv4 protocol) – typically used
  - type: communication type
    - SOCK_STREAM: reliable, 2-way, connection-based service
    - SOCK_DGRAM: unreliable, connectionless,
    - other values: need root permission, rarely used, or obsolete
  - protocol: specifies protocol (IPPROTO_TCP, IPPROTO_UDP) - usually set to 0 for default
- NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

A Socket-eye view of the Internet

medellin.cs.columbia.edu

(128.59.21.14)

newworld.cs.umass.edu

(128.119.245.93)

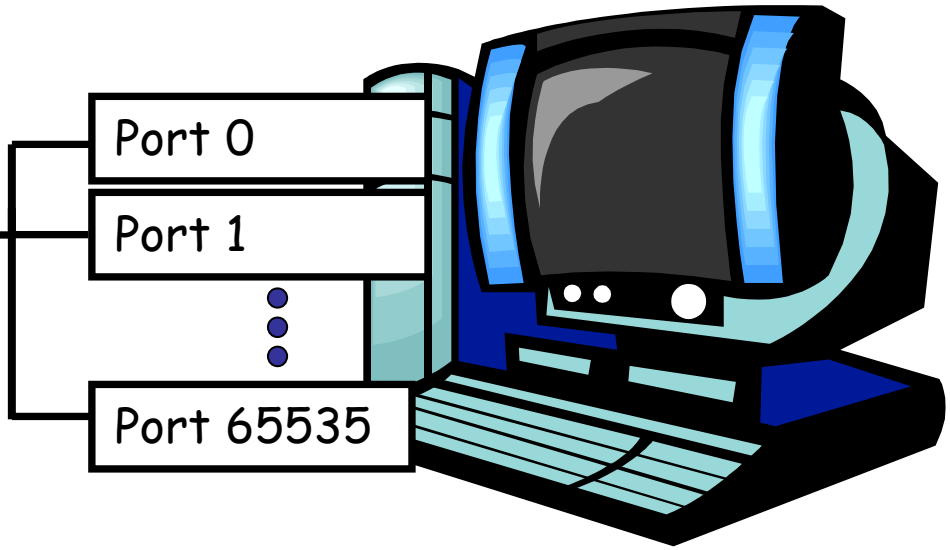
cluster.cs.columbia.edu

(128.59.21.14, 128.59.16.7, 128.59.16.5, 128.59.16.4)

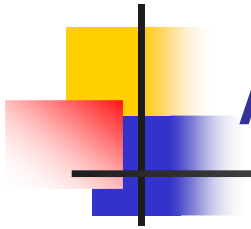
- Each host machine has an IP address
- When a packet arrives at a host

Ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)



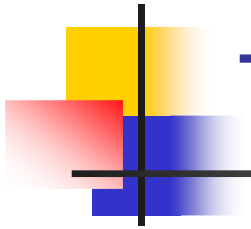
A socket provides an interface to send data to/from the network through a port from/to a process



Addresses, Ports and Sockets

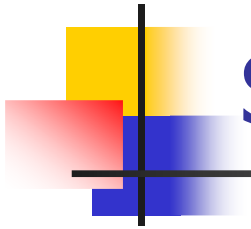
- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the key that gives you access to the right mailbox (one difference: assume outgoing mail is placed by you in your mailbox)

- Q: How do you choose which port should belong to a socket?
  - Note: not the remote end, but the **local port**



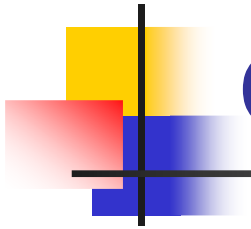
The bind function

- associates and (can exclusively) reserves a port for use by the socket
- int status = bind(sockid, &addrport, size);
  - status: error status, = -1 if bind failed
  - sockid: integer, socket descriptor
  - addrport: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY – chooses a local address)
  - size: the size (in bytes) of the addrport structure
- bind can be skipped for both types of sockets. When and why?



Skipping the bind

- SOCK_DGRAM:
  - if only sending, no need to bind. The OS finds a port each time the socket sends a pkt
  - if receiving, need to bind
- SOCK_STREAM:
  - destination determined during conn. setup
  - don't need to know port sending from (during connection setup, receiving end is informed of port)



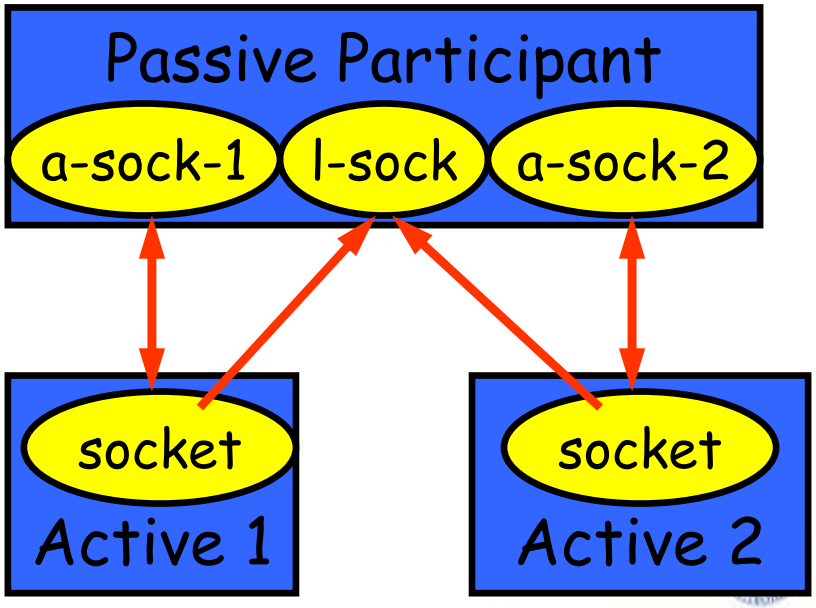
Connection Setup (SOCK_STREAM)

- Recall: no connection setup for SOCK_DGRAM
- A connection occurs between two kinds of participants
  - passive: waits for an active participant to request connection
  - active: initiates connection request to passive side
- Once connection is established, passive and active participants are "similar"
  - both can send & receive data
  - either can terminate the connection

Connection setup cont'd

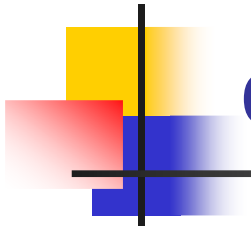
- Passive participant
  - step 1: listen (for incoming requests)
  - step 2/3: accept (a request)
  - step 4: data transfer
- The accepted connection is on a new socket
- The old socket continues to listen for other active participants
- Why?

- Active participant
  - step 2/3: request & establish connection
  - step 4: data transfer



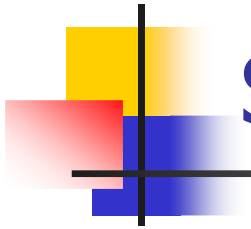
Connection setup: listen & accept

- Called by passive participant
- int status = listen(sock, queuelen);
  - status: 0 if listening, -1 if error
  - sock: integer, socket descriptor
  - queuelen: integer, # of active participants that can "wait" for a connection
  - listen is **non-blocking**: returns immediately
- int s = accept(sock, &name, &namelen);
  - s: integer, the new socket (used for data-transfer)
  - sock: integer, the orig. socket (being listened on)
  - name: struct sockaddr, address of the active participant
  - namelen: sizeof(name): value/result parameter
    - must be set appropriately before call
    - adjusted by OS upon return
  - accept is **blocking**: waits for connection before returning



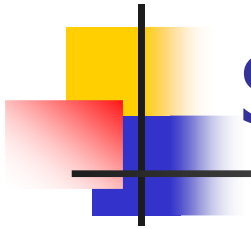
connect call

- int status = connect(sock, &name, namelen);
  - status: 0 if successful connect, -1 otherwise
  - sock: integer, socket to be used in connection
  - name: struct sockaddr: address of passive participant
  - namelen: integer, sizeof(name)
- connect is **blocking**



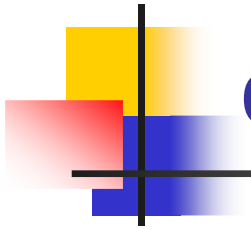
Sending / Receiving Data

- With a connection (SOCK_STREAM):
  - int count = send(sock, &buf, len, flags);
    - count: # bytes transmitted (-1 if error)
    - buf: char[], buffer to be transmitted
    - len: integer, length of buffer (in bytes) to transmit
    - flags: integer, special options, usually just 0
  - int count = recv(sock, &buf, len, flags);
    - count: # bytes received (-1 if error)
    - buf: void[], stores received bytes
    - len: # bytes received
    - flags: integer, special options, usually just 0
  - Calls are **blocking** [returns only after data is sent (to socket buf) / received]



Sending / Receiving Data (cont'd)

- Without a connection (SOCK_DGRAM):
  - int count = sendto(sock, &buf, len, flags, &addr, addrlen);
    - count, sock, buf, len, flags: same as send
    - addr: struct sockaddr, address of the destination
    - addrlen: sizeof(addr)
  - int count = recvfrom(sock, &buf, len, flags, &addr, &addrlen);
    - count, sock, buf, len, flags: same as recv
    - name: struct sockaddr, address of the source
    - namelen: sizeof(name): value/result parameter
- Calls are **blocking** [returns only after data is sent (to socket buf) / received]



close

- When finished using a socket, the socket should be closed:
- status = close(s);
  - status: 0 if successful, -1 if error
  - s: the file descriptor (socket being closed)
- Closing a socket
  - closes a connection (for SOCK_STREAM)
  - frees up the port used by the socket

The struct sockaddr

- The generic:

```
struct sockaddr {
        u_short sa_family;
        char sa_data[14];
};
```

  - sa_family
    - specifies which address family is being used
    - determines how the remaining 14 bytes are used

- The Internet-specific:

```
struct sockaddr_in {
        short sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

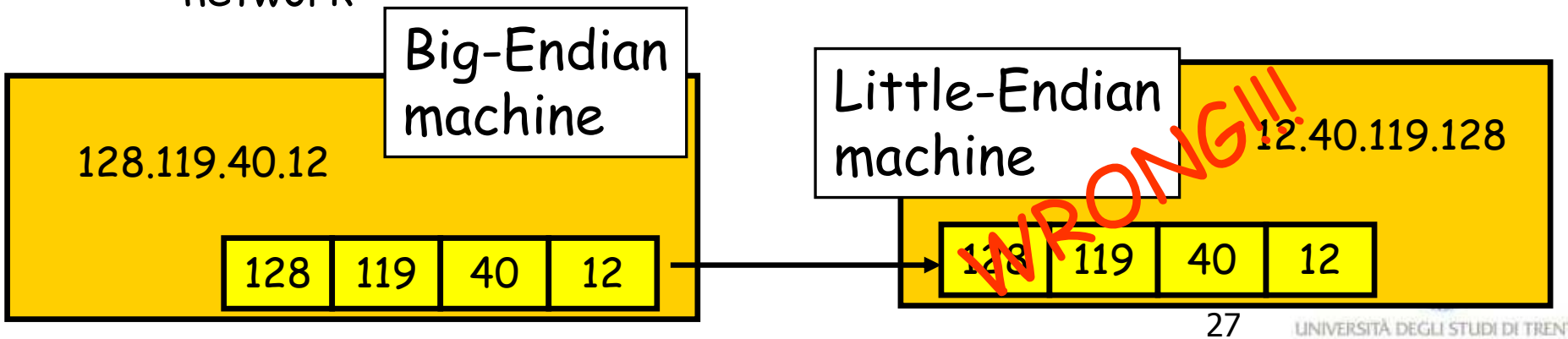
  - sin_family = AF_INET
  - sin_port: port # (0-65535)
  - sin_addr: IP-address
  - sin_zero: unused

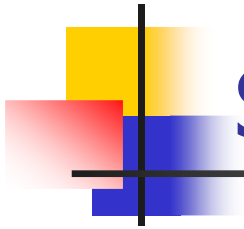
Address and port byte-ordering

- Address and port are stored as integers
  - u_short sin_port; (16 bit)
  - in_addr sin_addr; (32 bit)

```
struct in_addr {
  u_long s_addr;
};
```

- Problem:
  - different machines / OS's use different word orderings
    - little-endian: lower bytes first
    - big-endian: higher bytes first
  - these machines may communicate with one another over the network





Solution: Network Byte-Ordering

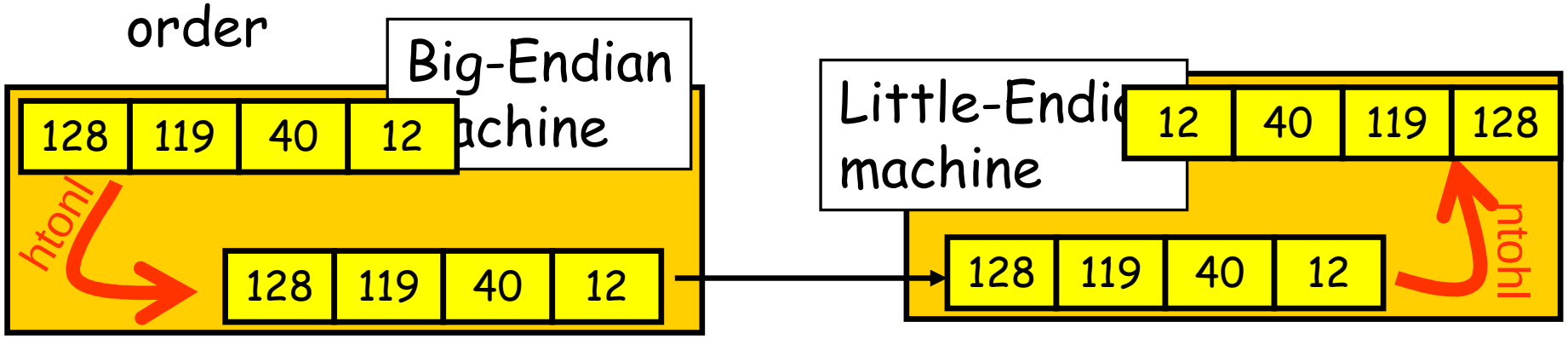
- Defs:
  - Host Byte-Ordering: the byte ordering used by a host (big or little)
  - Network Byte-Ordering: the byte ordering used by the network – always big-endian
- Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)
- Q: should the socket perform the conversion automatically?

- Q: Given big-endian machines don't need conversion routines and little-endian machines do, how do we avoid writing two versions of code?

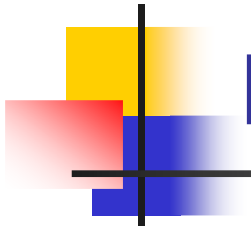
UNIX's byte-ordering funcs

- u_long htonl(u_long x);
- u_short htons(u_short x);
- u_long ntohl(u_long x);
- u_short ntohs(u_short x);

- On big-endian machines, these routines do nothing
- On little-endian machines, they reverse the byte order

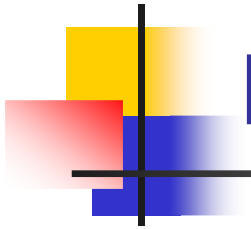


- Same code would have worked regardless of endian-ness of the two machines



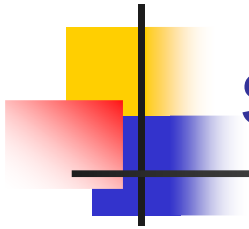
Dealing with blocking calls

- Many of the functions we saw block until a certain event
  - accept: until a connection comes in
  - connect: until the connection is established
  - recv, recvfrom: until a packet (of data) is received
  - send, sendto: until data is pushed into socket's buffer
    - Q: why not until received?
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing



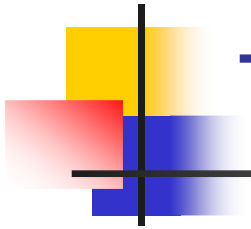
Dealing w/ blocking (cont'd)

- Options:
  - create multi-process or multi-threaded code
  - turn off the blocking feature (e.g., using the fcntl file-descriptor control function)
  - use the select function call.
- What does select do?
  - can be permanent blocking, time-limited blocking or non-blocking
  - input: a set of file-descriptors
  - output: info on the file-descriptors' status
  - i.e., can identify sockets that are "ready for use": calls involving that socket will return immediately



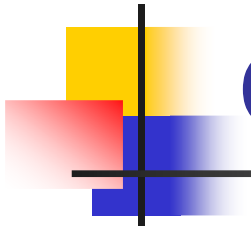
select function call

- int status = select(nfds, &readfds, &writefds, &exceptfds, &timeout);
  - status: # of ready objects, -1 if error
  - nfds: 1 + largest file descriptor to check
  - readfds: list of descriptors to check if read-ready
  - writefds: list of descriptors to check if write-ready
  - exceptfds: list of descriptors to check if an exception is registered
  - timeout: time after which select returns, even if nothing ready - can be 0 or ∞
    (point timeout parameter to NULL for ∞)



To be used with select:

- Recall select uses a structure, struct fd_set
  - it is just a bit-vector
  - if bit *i* is set in [readfds, writefds, exceptfds], select will check if file descriptor (i.e. socket) *i* is ready for [reading, writing, exception]
- Before calling select:
  - FD_ZERO(&fdvar): clears the structure
  - FD_SET(i, &fdvar): to check file desc. *i*
- After calling select:
  - int FD_ISSET(i, &fdvar): boolean returns TRUE iff *i* is "ready"



Other useful functions

- bzero(char* c, int n): 0's n bytes starting at c
- gethostname(char *name, int len): gets the name of the current host
- gethostbyaddr(char *addr, int len, int type): converts IP hostname to structure containing long integer
- inet_addr(const char *cp): converts dotted-decimal char-string to long integer
- inet_ntoa(const struct in_addr in): converts long to dotted-decimal notation

- Warning: check function assumptions about byte-ordering (host or network). Often, they assume parameters / return solutions in network byte-order