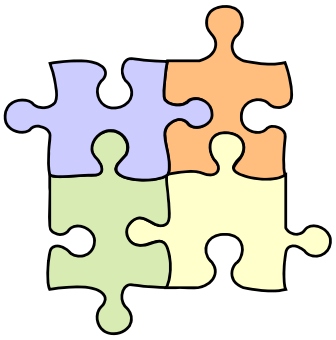
Luca Cabibbo

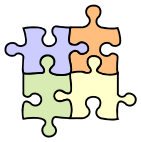


Architetture Software

Comunicazione interprocesso e socket

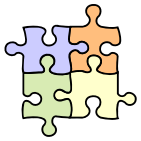
Dispensa MW 2

ottobre 2008



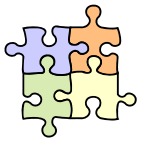
- Fonti

- [CDK/4e] Chapter 4, Interprocess Communication
- [Liu] Chapter 2, Interprocess Communication
- [Liu] Chapter 4, The Socket API



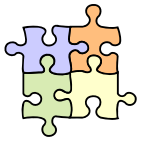
- Obiettivi e argomenti

- Obiettivi
  - richiamare il meccanismo delle socket
  - comprendere i limiti delle socket
  - comprendere alcuni aspetti e problemi di base della comunicazione interprocesso nei sistemi distribuiti
- Argomenti
  - comunicazione interprocesso
  - socket
  - che messaggi scambiare?
  - comunicazione client-server
  - discussione
  - appendice: un'applicazione client-server a strati



* Comunicazione interprocesso

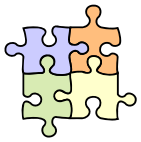
- La spina dorsale dei sistemi distribuiti è la **comunicazione interprocesso** – **IPC**, *interprocess communication*
  - l'IPC è basata sui protocolli di base di Internet
    - ad esempio, TCP e UDP
  - questi protocolli di base possono essere acceduti direttamente mediante servizi del sistema operativo
    - socket
  - oppure indirettamente mediante degli opportuni servizi di middleware
    - ciascun servizio di middleware offre un'opportuna astrazione di programmazione per sistemi distribuiti



Comunicazione interprocesso

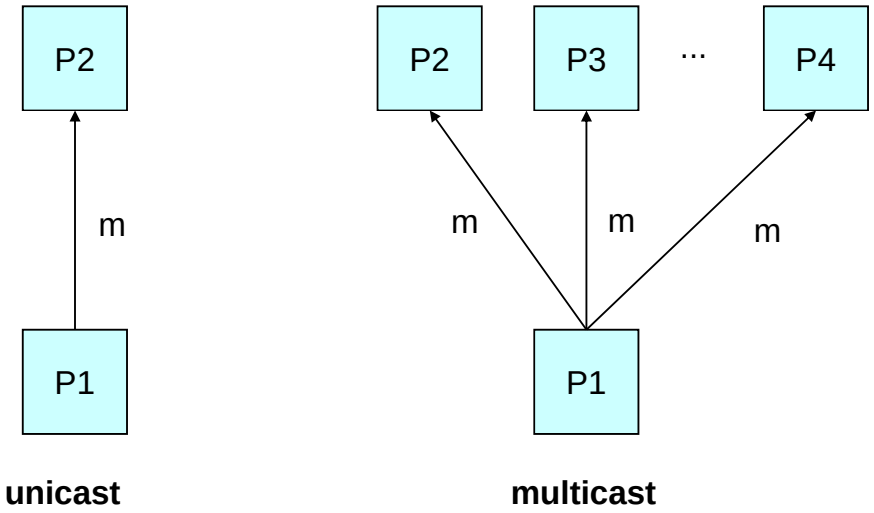
- IPC di base

- due processi indipendenti (che possono essere su calcolatori diversi) si scambiano dati/messaggi mediante una rete di interconnessione
- anche sulla base di un protocollo (formati dei messaggi e regole, anche di sincronizzazione) accettato da entrambe le parti

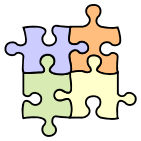


Comunicazione unicast e multicast

- La comunicazione può essere unicast o multicast

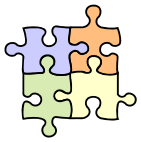


- Ci concentriamo principalmente sulla comunicazione unicast



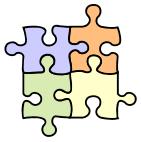
Le primitive send e receive

- Lo scambio di messaggi tra processi può essere basato su due operazioni primitive
  - definite in termini di messaggio e di processo mittente/destinazione
    - ***send*** – consente ad un processo di trasmettere dati ad un altro processo
      - ***send(receiving process, data)***
    - ***receive*** – consente ad un processo di accettare dati trasmessi da un altro processo
      - ***receive(sending process, *buffer)***
  - a questo livello di astrazione, per *messaggio* si intende semplicemente una qualche sequenza, binaria o testuale, di dati

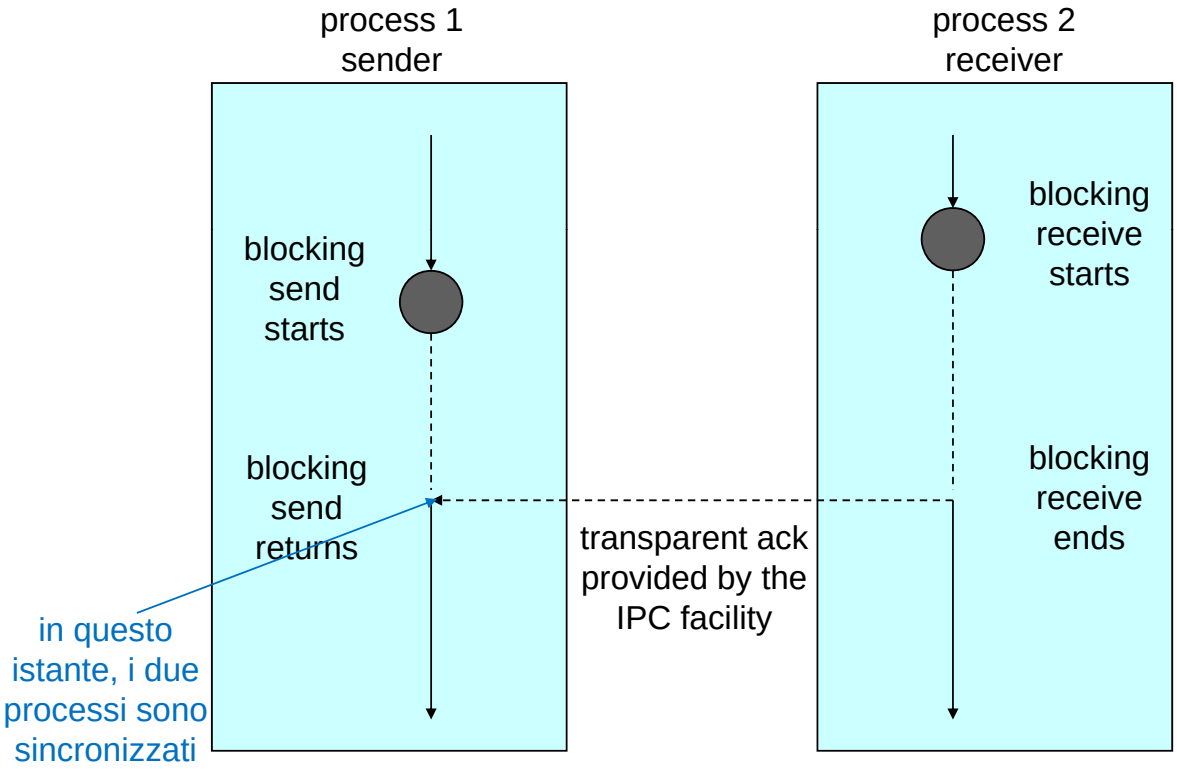


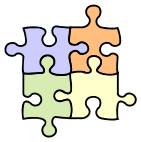
Comunicazione sincrona e asincrona

- La comunicazione tra processi può essere sincrona o asincrona
  - **comunicazione sincrona**
    - lo scambio di un messaggio costituisce un punto di sincronizzazione tra i processi comunicanti
    - le operazioni ***send*** e ***receive*** sono *bloccanti*
    - i dati trasmessi devono essere stati ricevuti prima di poter andare avanti
  - **comunicazione asincrona**
    - nella comunicazione asincrona, l'operazione ***send*** è *non bloccante* – il messaggio viene copiato in un buffer, e poi il processo mittente può proseguire, mentre il messaggio viene trasmesso
    - l'operazione ***receive*** è normalmente bloccante

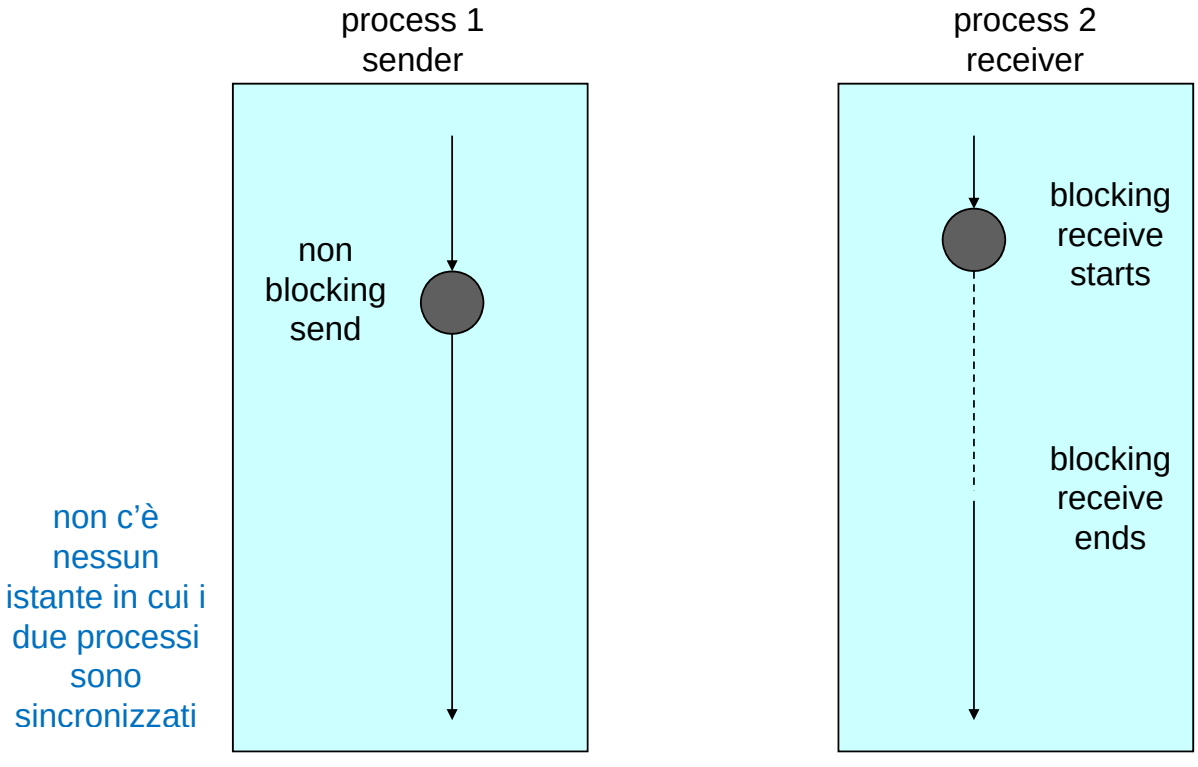


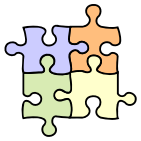
Comunicazione sincrona





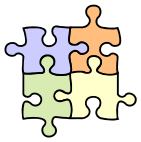
Comunicazione asincrona





- Precauzioni: sincronizzazione

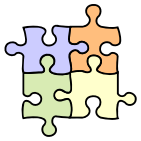
- I processi devono comunque provvedere alla loro *sincronizzazione*
  - ad es., non è accettabile che le operazioni bloccanti blocchino un processo indefinitamente
    - una possibile soluzione è la specifica di un timeout
    - altrimenti, le operazioni bloccanti possono essere eseguite nell'ambito di un processo child oppure di un thread dedicato alla comunicazione
  - un blocco indefinito può essere anche causato da uno stallo
    - ad es., operazioni eseguite in ordine non corretto
  - la mancanza di sincronizzazione può causare la perdita di messaggi importanti
    - ad es., un processo client che fa una richiesta non bloccante ad un processo server non può assumere che una sua richiesta sia stata ricevuta ed elaborata completamente dal processo server



- Precauzioni: sincronizzazione

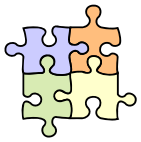
- I processi devono comunque provvedere alla loro *sincronizzazione*
  - ad es., non è a tabile che le operazioni bloccanti blocchino un processo in
    - una
    - altr seguite
      dedicato
  - u tallo
  - la perdita di
    me
    - ad es., esta non bloccante
      ad un processo serv assumere che una sua
      richiesta sia stata ricevuta ed elaborata completamente dal
      processo server

questo non vuol dire che processi concorrenti che devono operare in modo sincronizzato devono necessariamente comunicare solo in modo sincrono...



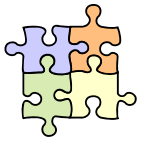
- Destinazione dei messaggi

- In Internet, i messaggi sono inviati ad una coppia (***indirizzo IP***, ***porta***)
  - in IP, la comunicazione è tra calcolatori
    - ciascun calcolatore è identificato da un indirizzo IP
  - nella IPC, la comunicazione è tra processi
    - ciascun processo è caratterizzato da un numero di porta all'interno di un calcolatore – quindi da una coppia (***indirizzo IP***, ***porta***)
    - una porta corrisponde ad un solo destinatario – con l'eccezione delle porte multicast
    - più mittenti possono inviare messaggi sulla stessa porta



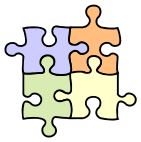
- UDP e TCP

- TCP e UDP forniscono capacità di comunicazione (trasporto) in una forma utile per la scrittura programmi
- *UDP* è un protocollo per il trasporto di datagrammi – senza ack e ritrasmissione
  - non offre garanzie di consegna – i datagrammi potrebbero venire persi
  - overhead basso – non è richiesto setup
- *TCP* è un servizio di trasporto più sofisticato
  - connection-oriented – richiesto il setup di un canale di comunicazione bidirezionale
  - consegna "affidabile" (vedi dopo) di uno stream ordinato di dati – uso di ack e ritrasmissione



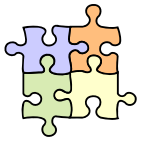
Altre due primitive

- Due ulteriori primitive per l'IPC connection-oriented
  - ***connect*** – per stabilire una connessione logica tra due processi
    - request-to-connect + accept-connection
  - ***disconnect*** – per terminare una connessione logica su entrambi i lati della comunicazione



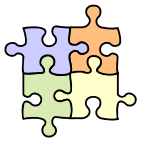
- Precauzioni: affidabilità

- In un sistema distribuito, ci possono essere *fallimenti* nella comunicazione
  - questi fallimenti possono essere causati da
    - guasti nei processi che comunicano
    - guasti nei canali di comunicazione
  - possibili conseguenze
    - messaggi persi
    - messaggi trasmessi male o trasmessi più volte
    - messaggi trasmessi fuori ordine
  - una *network* può diventare una *not-work*



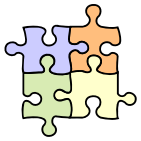
Precauzioni: affidabilità

- Un servizio di comunicazione può richiedere/garantire alcune forme di *affidabilità*
  - *validità*
    - garanzia di consegna dei messaggi – anche a fronte dell'eventuale perdita di alcuni pacchetti
  - *integrità*
    - i messaggi ricevuti sono identici a quelli trasmessi, senza duplicazioni di messaggi
  - *ordine*
    - messaggi consegnati nell'ordine in cui sono stati trasmessi
- Si noti il *può*
  - in alcuni casi non sono richieste tutte queste forme di affidabilità



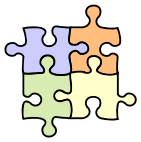
Precauzioni: affidabilità

- Un servizio di comunicazione può mascherare alcuni dei possibili fallimenti – realizzando un canale di comunicazione con un certo livello di affidabilità
  - ma il servizio di comunicazione in uso è affidabile? in che senso?
  - se non è affidabile come vorrei, come posso fornire il livello richiesto di affidabilità?
- Detto in altro modo
  - ciascun servizio di middleware garantisce un qualche livello di affidabilità (in alcuni casi più di uno, selezionabile)
  - ogni volta che si usa uno strumento di middleware è opportuno chiedersi
    - quali garanzie di affidabilità sono possibili?
    - a quale "prezzo"?

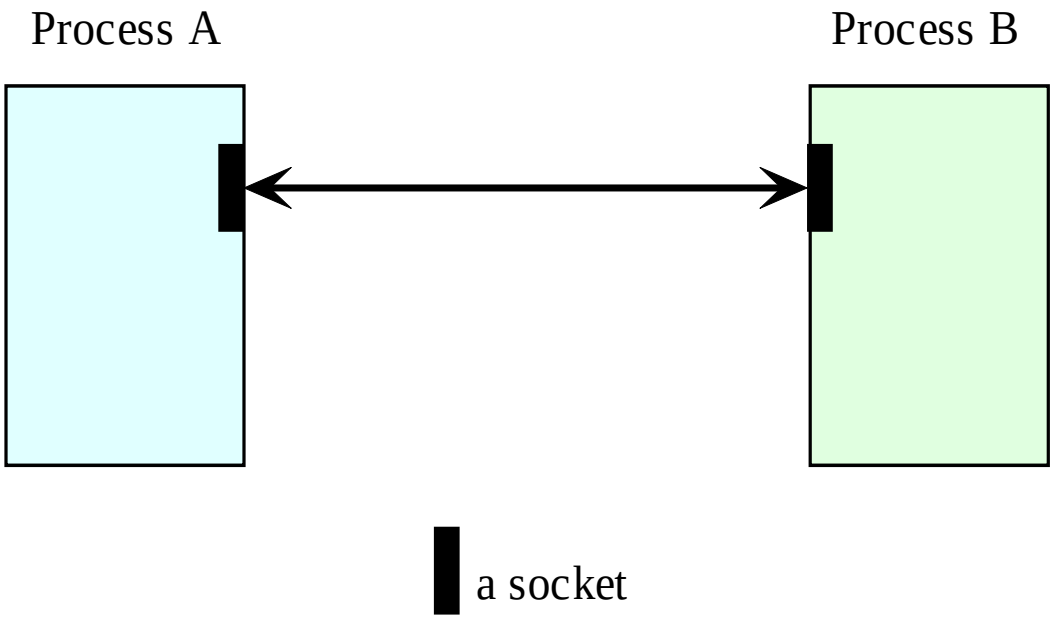


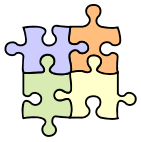
* Socket

- Le **socket** sono un'astrazione di programmazione per TCP e UDP
  - forniscono un endpoint per la comunicazione tra processi
    - letteralmente, socket = connettore, presa
  - l'astrazione di comunicazione interprocesso fornita consiste nella possibilità di inviare un messaggio da una socket di un processo e ricevere il messaggio tramite una socket di un altro processo
- Ci concentriamo sulle API di Java per le socket

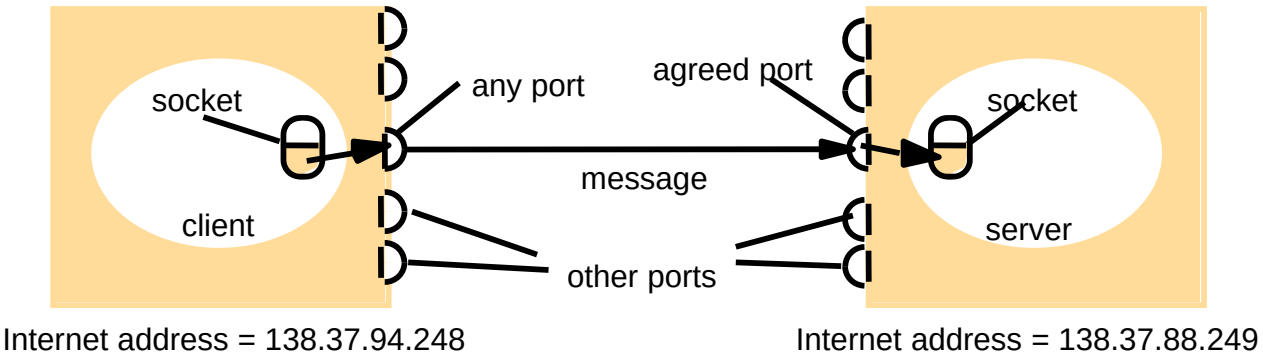


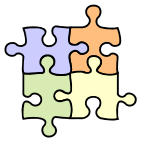
Modello logico





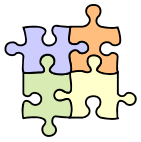
Socket e porte



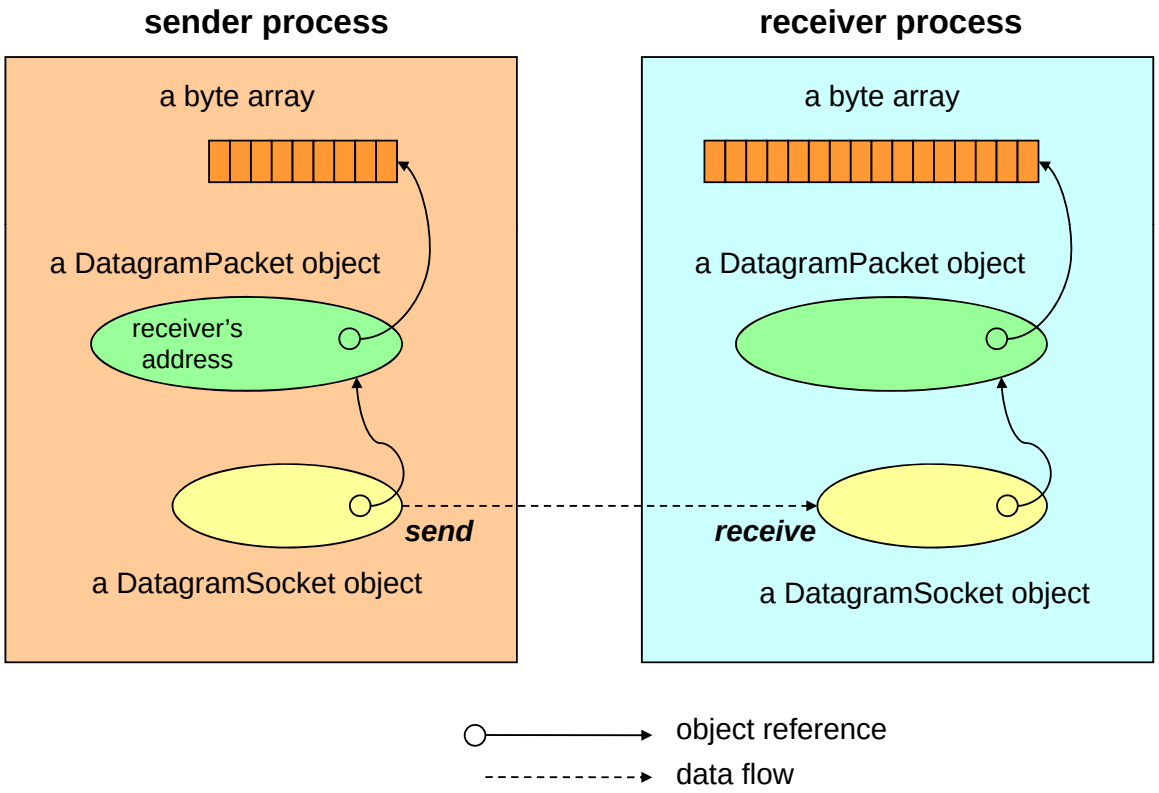


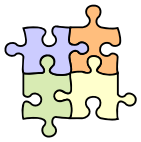
- Socket UDP

- UDP consente la trasmissione di datagrammi tra due processi
  - normalmente, send non bloccante e receive bloccante
  - receive from any – l'operazione receive non specifica il mittente



API di Java – strutture di dati coinvolte





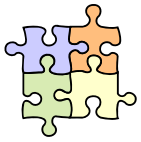
Programmi sender e receiver

sender program

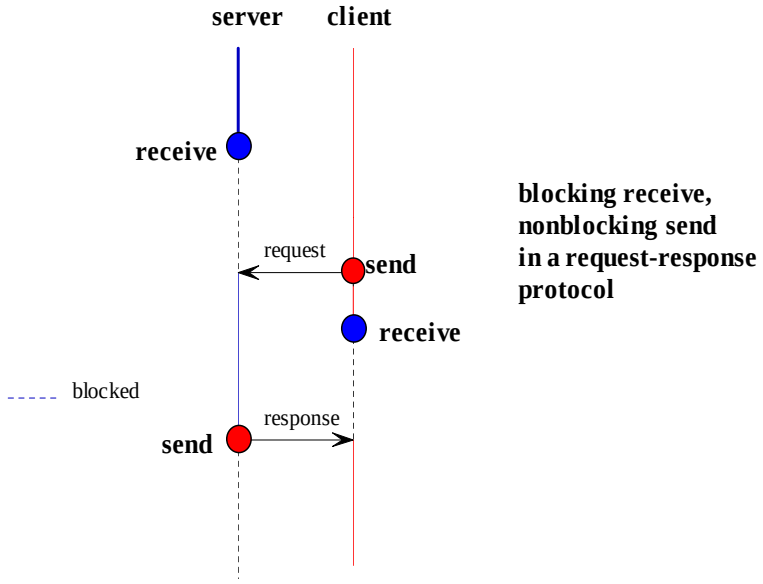
- create a datagram socket and bind it to any local port
- place data in a byte array
- create a datagram packet, specifying the data array and the receiver's address
- invoke the send method of the socket with a reference to the datagram packet

receiver program

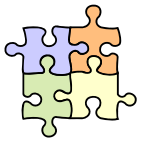
- create a datagram socket and bind it to a specific local port
- create a byte array for receiving the data
- create a datagram packet, specifying the data array
- invoke the receive method of the socket with a reference to the datagram packet



Una possibile modalità di sincronizzazione

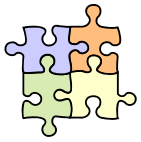


if data is sent before a corresponding receive operation is issued, the data will be discarded by the runtime support and will not be received.



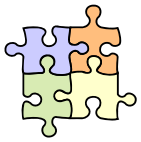
DatagramPacket e DatagramSocket

- DatagramPacket(byte[] buffer, int length)
  - crea un DatagramPacket per la ricezione di pacchetti
- DatagramPacket(byte[] buffer, int length, InetAddress address, int port)
  - crea un DatagramPacket per l'invio di pacchetti alla socket specificata (address rappresenta un indirizzo IP)
- DatagramSocket()
  - crea una DatagramSocket legata ad una porta qualsiasi (ok per inviare pacchetti ma non per riceverli)
- DatagramSocket(int port)
  - crea una DatagramSocket legata alla porta specificata (ok anche per ricevere pacchetti)
- void close()
  - chiude questo oggetto DatagramSocket
- void receive(DatagramPacket p)
  - riceve un pacchetto usando questa socket e buffer
- void send(DatagramPacket p)
  - invia questo pacchetto



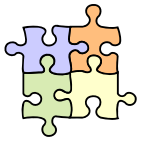
Client UDP – manda messaggio e riceve risposta

```
import java.net.*;
import java.io.*;

/** Client UDP che invia un messaggio e ottiene una risposta. */
public class UDPClient {

    /**
     * @param args[0] contenuto del messaggio
     * @param args[1] nome del server
     */
    public static void main(String[] args) {
        ...
    }
}
```

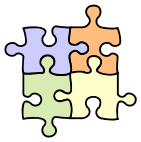
Client UDP – manda messaggio e riceve risposta

```
DatagramSocket aSocket = null;
try {
        aSocket = new DatagramSocket();
        byte[] m = args[0].getBytes();
        InetAddress aHost = InetAddress.getByName(args[1]);
        int serverPort = 6789;
        DatagramPacket request =
                new DatagramPacket(m, args[0].length(), aHost, serverPort);
        /* invia una richiesta */
        aSocket.send(request);    // send non bloccante
        byte[] buffer = new byte[1000];
        DatagramPacket reply =
                new DatagramPacket(buffer, buffer.length);
        // il server gli risponderà sulla stessa porta
        /* riceve una richiesta */
        aSocket.receive(reply);    // receive bloccante
        System.out.println("Reply: " + new String(reply.getData()));
} catch (SocketException e) {
        System.out.println("Socket: " + e.getMessage());
} catch (IOException e) {
        System.out.println("IO: " + e.getMessage());
} finally {
        if (aSocket!=null)    aSocket.close();
}
```



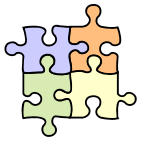
Server UDP – ripetutamente riceve una richiesta e risponde alla richiesta

```
import java.net.*;
import java.io.*;

/** Server UDP che riceve ripetutamente delle richieste e le rimanda
 *  indietro al client. */
public class UDPServer {

    public static void main(String[] args) {
        ...
    }
}
```

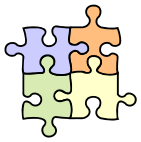
Server UDP – ripetutamente riceve una richiesta e risponde alla richiesta

```
DatagramSocket aSocket = null;
try {
        aSocket = new DatagramSocket(6789);
        byte[] buffer = new byte[1000];
        while (true) {
                DatagramPacket request =
                        new DatagramPacket(buffer, buffer.length);
                /* aspetta una richiesta */
                aSocket.receive(request);    // receive bloccante
                System.out.println("Ricevuti " + request.getLength() + " byte(s).");
                System.out.println(new String(request.getData()));
                DatagramPacket reply =
                        new DatagramPacket(request.getData(), request.getLength(),
                                        request.getAddress(), request.getPort());
                /* invia una risposta */
                aSocket.send(reply);    // send non bloccante
        }
} catch (SocketException e) {
        System.out.println("Socket: " + e.getMessage());
} catch (IOException e) {
        System.out.println("IO: " + e.getMessage());
} finally {
        if (aSocket!=null)    aSocket.close();
}
```



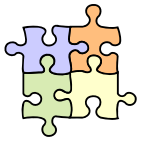
UDP – caratteristiche e discussione

- Dimensione dei messaggi
  - il destinatario deve specificare la dimensione del buffer – se il messaggio trasmesso è troppo grande, viene troncato
  - spesso datagrammi fino a 8kbyte
- Receive from any
  - l'operazione receive non specifica il mittente
  - da un messaggio ricevuto, è possibile risalire all'IP e porta del mittente
- Blocco
  - normalmente, la send è non bloccante e la receive è bloccante
    - la send termina dopo aver copiato il messaggio in un buffer in uscita
    - la receive è normalmente bloccante – è possibile introdurre un timeout



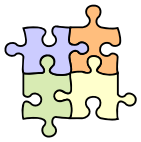
UDP – caratteristiche e discussione

- Fallimenti
  - UDP è senza ack né ritrasmissione
  - ci possono essere problemi di
    - integrità
    - omissione – si possono perdere messaggi
    - ordine – i messaggi possono arrivare fuori ordine
    - duplicazione – possono arrivare messaggi duplicati
  - ...
- Affidabilità
  - UDP può essere usato per definire un proprio protocollo di comunicazione affidabile
    - come? come gestire i fallimenti di cui sopra?



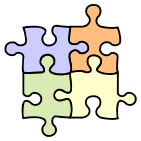
UDP – caratteristiche e discussione

- Come usare UDP per per definire un proprio protocollo di comunicazione affidabile?
  - problemi di integrità
    - posso usare checksum – ma anche cifratura, firme, ...
  - problemi di omissione – perdita di messaggi
    - se non torna una risposta, ripeto la richiesta
  - problemi di ordine – i messaggi possono arrivare fuori ordine
    - gestisco esplicitamente l'ordine dei messaggi (ad es., li numero)
  - problemi di duplicazione – possono arrivare messaggi duplicati
    - posso associare un identificatore ai messaggi, e quindi accorgermi di messaggi duplicati
  - ...



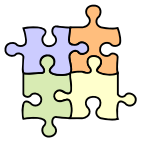
UDP – caratteristiche e discussione

- Uso di UDP nella definizione di un servizio client-server
  - nell'esempio mostrato, il server è in grado di soddisfare richieste dai suoi client – che nell'esempio sono costituite da un singolo messaggio
  - possibile per un server gestire delle "richieste" che siano composte da una sequenza di più messaggi?
    - no (almeno, non facilmente) con UDP connectionless
      - si ricordi che l'ordine di ricezione non è garantito, e che la receive è una receive from any, quindi un altro client si potrebbe "intromettere" in una interazione tra un client ed il server
    - forse, con UDP connection-oriented – ma il suo uso è poco comune, essendogli preferite le socket TCP



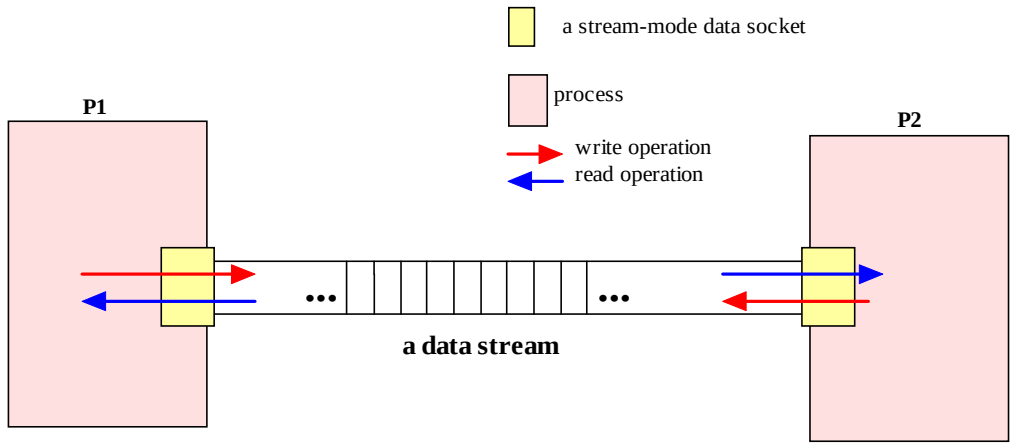
UDP – caratteristiche e discussione

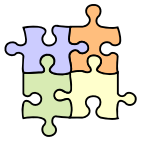
- In conclusione, possibili usi di UDP
  - per server stateless
  - per ridurre l'overhead della comunicazione
  - se è accettabile una comunicazione non affidabile
  - se posso permettermi di gestire esplicitamente l'affidabilità



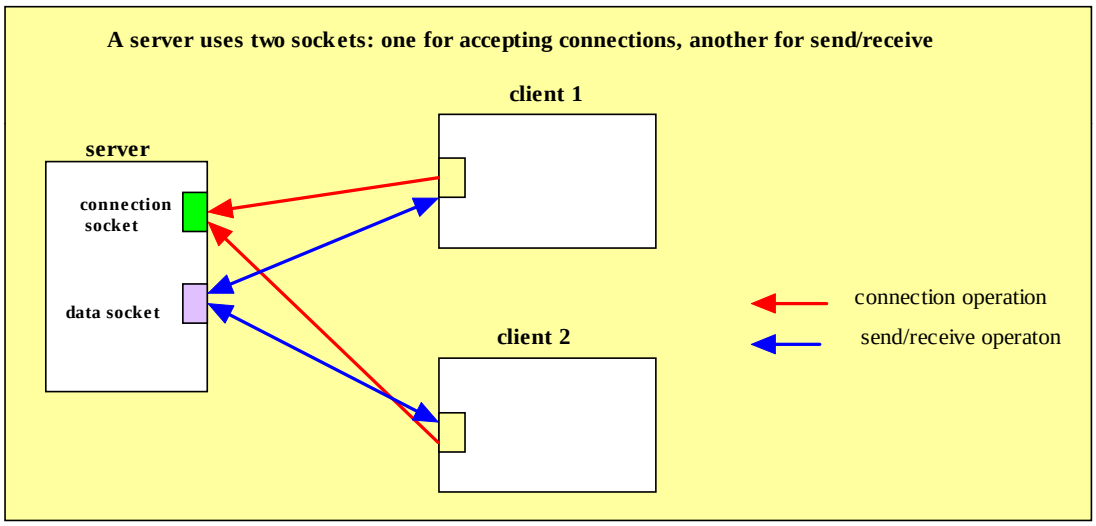
- Socket TCP

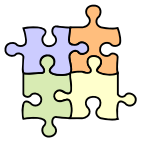
- TCP consente la trasmissione di flussi di dati (bidirezionali) tra due processi





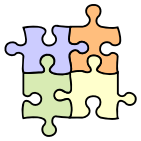
Socket TCP





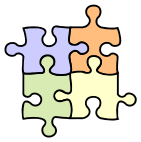
Socket TCP

- Le API assumono che, nel momento in cui una coppia di processi devono stabilire una connessione
  - uno abbia il ruolo di *client* – fa una richiesta di ***connect***
  - l'altro quello di *server* – a fronte di una richiesta di connect risponde con una ***accept*** – bloccante
- Da quel momento in poi, possono agire come pari (*peer*), con operazioni
  - ***read*** – bloccanti
  - ***write*** – non bloccanti
- Due tipi di socket
  - *server socket* – per accettare connessioni
  - (*data*) *socket* – per lo scambio di dati



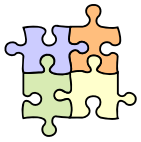
ServerSocket e Socket

- ServerSocket(int port)
  - crea una ServerSocket legata alla porta specificata – per accettare connessioni
- Socket accept() throws IOException
  - attende una richiesta di connessione e l'accetta – restirtuisce la Socket per gestire la connessione
- void close()
  - chiude questa socket
- Socket(InetAddress address, int port)
  - crea una stream socket (TCP) e richiede una connessione alla server socket con l'indirizzo e la porta specificata
- void close()
  - chiude questa socket
- InputStream getInputStream() throws IOException
  - l'input stream della socket – per effettuare letture
- OutputStream getOutputStream() throws IOException
  - l'output stream della socket – per effettuare scritture



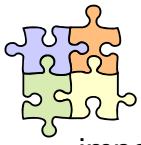
Client TCP – si connette al server, invia richiesta e riceve risposta

```
import java.net.*;
import java.io.*;

/** Client TCP che invia un messaggio e ottiene una risposta. */
public class TCPClient {

    /**
     * @param args[0] contenuto del messaggio
     * @param args[1] nome del server
     */
    public static void main(String[] args) {
        ...
}
```



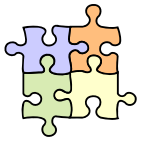
Client TCP – si connette al server, invia richiesta e riceve risposta

```
Socket s = null;
try {
        int serverPort = 7896;
        s = new Socket(args[1], serverPort);
        DataInputStream in = new DataInputStream(s.getInputStream());
        DataOutputStream out = new DataOutputStream(s.getOutputStream());
        /* invia una richiesta */
        out.writeUTF(args[0]);    // non bloccante
        /* riceve una risposta */
        String data = in.readUTF();    // bloccante
        System.out.println("Reply: " + data);
} catch (UnknownHostException e) {
        System.out.println("Socket: " + e.getMessage());
} catch (EOFException e) {
        System.out.println("EOF: " + e.getMessage());
} catch (IOException e) {
        System.out.println("IO: " + e.getMessage());
} finally {
        if (s!=null)
                try { s.close(); }
                catch (IOException e) { System.out.println("IO: close failed"); }
}
```

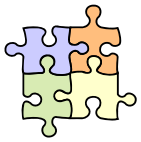
Server TCP (1) – effettua una connessione con ciascun client

```
import java.net.*;
import java.io.*;

/** Server TCP che effettua una connessione per ciascun cliente
 *  e rimanda indietro le richieste dei client. */
public class TCPServer {

    public static void main(String[] args) {
        try {
                int serverPort = 7896;
                ServerSocket listenSocket = new ServerSocket(serverPort);
                while (true) {
                        /* accetta una richiesta */
                        Socket clientSocket = listenSocket.accept();    // bloccante
                        // la richiesta sarà gestita in un nuovo thread, separato
                        TcpConnection c = new TcpConnection(clientSocket);
                }
        } catch (IOException e) {
                System.out.println("IO: " + e.getMessage());
        }
    }
}
```



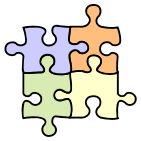
Server TCP (2) – in ciascuna connessione, risponde alla richiesta

```
import java.net.*;
import java.io.*;

public class TcpConnection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;

    public TcpConnection(Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream(clientSocket.getInputStream());
            out = new DataOutputStream(clientSocket.getOutputStream());
            this.start();    // esegue run() in un nuovo thread
        } catch (IOException e) {
            System.out.println("IO: " + e.getMessage());
        }
    }

    ...
```

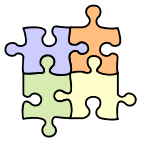
Server TCP (3) – in ciascuna connessione, risponde alla richiesta

```
    /* run eseguito in un nuovo thread */
    public void run() {
        try {
            /* riceve una richiesta */
            String data = in.readUTF();  // bloccante
            System.out.println("Ricevuto: " + data);
            /* invia una risposta */
            out.writeUTF(data);    // non bloccante
        } catch (EOFException e) {
            System.out.println("EOF: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("IO: " + e.getMessage());
        } finally {
            try {
                clientSocket.close();
            } catch (IOException e) {
                System.out.println("IO: close failed");
            }
        }
    }
}
```



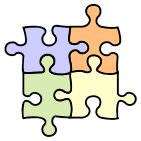
TCP – caratteristiche e discussione

- Dimensione dei messaggi
  - flussi di dati di dimensione illimitata
  - comunicazione – a buffer “pieno”, oppure forzata (flush)
- Flusso dei messaggi
  - pacchetti inizialmente persi vengono automaticamente ritrasmessi – sulla base di uno schema di acknowledgement
  - pacchetti ricevuti duplicati vengono scaricati
  - i messaggi vengono ricevuti in ordine
- Flusso di controllo
  - il protocollo prova a regolare automaticamente la velocità di comunicazione (nei due versi)



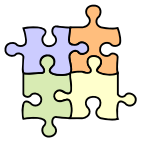
TCP – caratteristiche e discussione

- Uso di TCP
  - molti servizi comunemente usati sono basati su TCP – ad es., HTTP, FTP, telnet, SMTP, ...
- Affidabilità
  - in caso di congestione della rete, una connessione TCP può essere dichiarata broken – quando una connessione è broken
    - un processo non può sapere se i messaggi che ha inviato sono stati consegnati
    - non è possibile distinguere tra fallimento della rete e fallimento del processo remoto
  - pertanto, la comunicazione TCP non è completamente affidabile
    - non garantisce la consegna dei messaggi a fronte di ogni possibile difficoltà



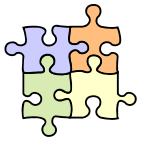
Socket – discussione

- In generale, la comunicazione basata su socket (sia UDP che TCP) soffre di diverse limitazioni
  - il programmatore può prendersi carico direttamente di sopperire alle limitazioni effettive
  - oppure può decidere di usare uno strumento specifico di middleware, che
    - supera le limitazioni riscontrate
    - offre un paradigma di programmazione più semplice da utilizzare – nascondendo la complessità della comunicazione
    - a costo, probabilmente, di un qualche overhead



* Che messaggi scambiare?

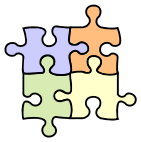
- Le socket offrono un'astrazione di programmazione che consente di scambiare messaggi o flussi di dati tra processi distribuiti
  - come detto, a questo livello di astrazione, per messaggio si intende semplicemente una qualche sequenza, binaria o testuale, di dati
  - ma quali sono i tipi di messaggi che un gruppo di processi possono scambiarsi utilmente ?
  - che cosa rappresentano/possono rappresentare questi messaggi?



Chiamata di procedute remote

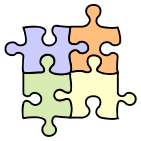
- Un caso comune – *chiamata di procedure remote* (aka *invocazione di operazioni remote*)
  - un server espone, mediante un opportuno protocollo, un insieme di operazioni/procedure la cui esecuzione può essere richiesta remotamente
    - il protocollo definisce, per ciascuna operazione
      - il formato del messaggio per richiamare la procedura
      - il formato del messaggio con cui invierà la risposta
    - il protocollo può anche definire l'ordine con cui è possibile richiedere le varie procedure
  - i client possono chiedere al server l'esecuzione di procedure remote adeguandosi a questo protocollo e questi formati

è un caso comune – ma non è l'unica possibilità



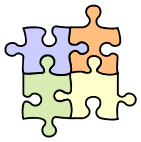
Messaggi per chiamate di procedure remote

- Tipi di messaggi coinvolti nella chiamata di procedure remote
  - *richiesta*
    - codifica l'operazione richiesta e i parametri attuali
  - *risposta*
    - codifica i risultati restituiti

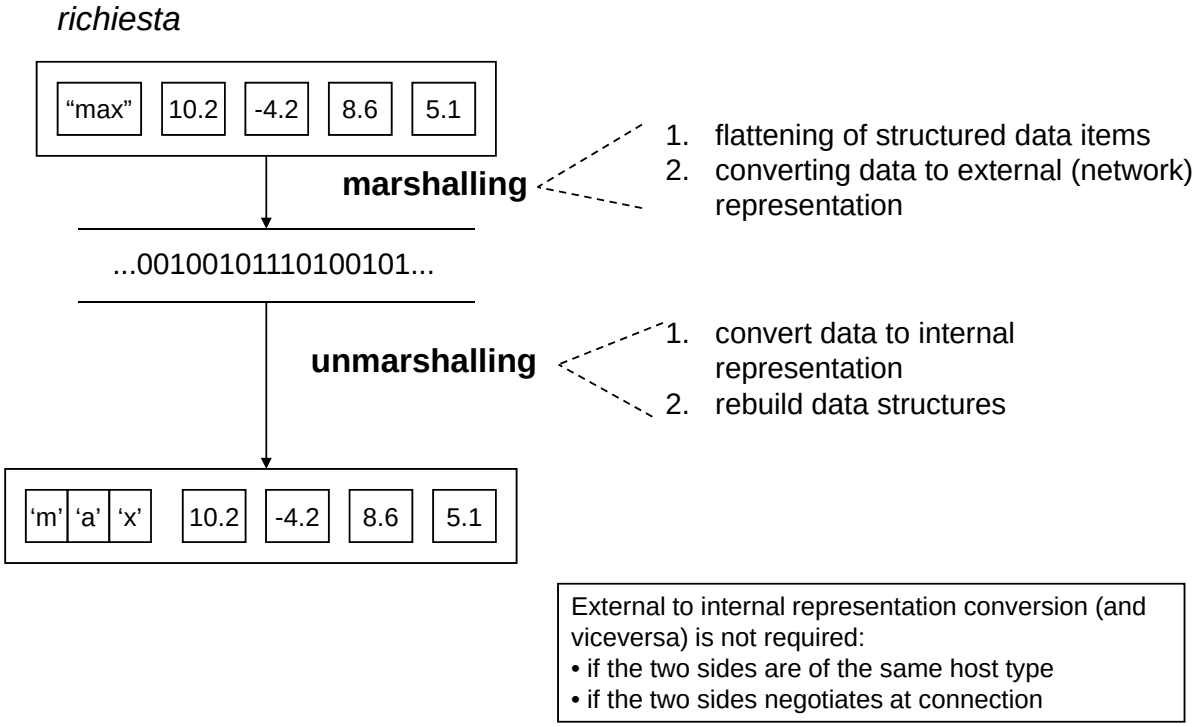


Marshalling e unmarshalling

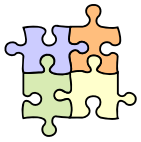
- Le due parti devono essere d'accordo sul formato (sintassi e semantica) dei messaggi scambiati
  - trascuriamo qui dettagli di basso livello (che vanno comunque considerati)
    - ad es., le rappresentazioni "little endian" e "big endian"
- Le due parti devono inoltre svolgere attività di
  - *marshalling*
    - assemblare un gruppo di dati in una forma adatta ad essere trasmessa come messaggio
  - *unmarshalling*
    - disassemblare un messaggio – per estrarre i dati in esso contenuti



Marshalling e unmarshalling



External to internal representation conversion (and viceversa) is not required:
- if the two sides are of the same host type
- if the two sides negotiates at connection



Comunicazione richiesta-risposta

programma client

ho bisogno che il server soddisfi una mia richiesta:

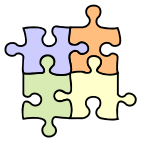
- effettuo il marshalling della richiesta (operazioni e dati)
- invio il messaggio di richiesta

- ricevo il messaggio di risposta
- effettuo l'unmarshalling della risposta (risultati)

programma server

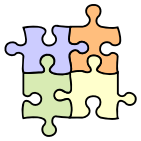
sono in attesa che un client mi faccia qualche richiesta:

- ricevo il messaggio di richiesta
- effettuo l'unmarshalling della richiesta (dati e operazioni)
- eseguo la richiesta (calcolo i risultati a partire dai dati)
- effettuo il marshalling della risposta (risultati)
- invio il messaggio di risposta



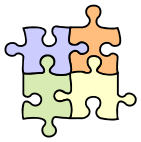
Esempio

- Si noti che è necessario uno schema di codifica per
  - le operazioni
    - ad es., un numero naturale, usato come primo elemento/byte della richiesta
  - dati e risultati



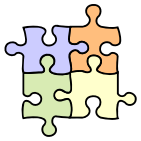
Alcuni scenari

- I dati scambiati potrebbero essere strutturati
  - in formato binario
  - in formato ASCII
  - in formato XML
    - formato testuale, auto-descrivente
    - offre flessibilità nella possibilità di estendere i messaggi scambiati
- Scambio di oggetti serializzati
  - Java consente di serializzare un oggetto/un grafo di oggetti collegati come una sequenza binaria – e di trasmetterli e deserializzarli



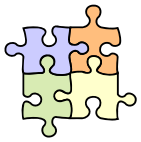
* Comunicazione client/server

- Si supponga si voler realizzare un'applicazione *client-server* usando la comunicazione basata su socket
  - la comunicazione client-server è basata su un protocollo (normalmente sincrono) di tipo richiesta-risposta
    - richiesta – il client chiede al server di eseguire un'operazione – poi si blocca in attesa di una risposta
    - risposta – il server restituisce il controllo e i risultati dell'esecuzione dell'operazione
- Quali gli (alcuni) aspetti da prendere in considerazione?
  - quanti/quali i servizi erogati dal server?
  - il server (i servizi) sono stateless o stateful?
  - se ci sono client multipli concorrenti, lo stato della sessione è gestito dai client o dal server?
  - TCP o UDP?

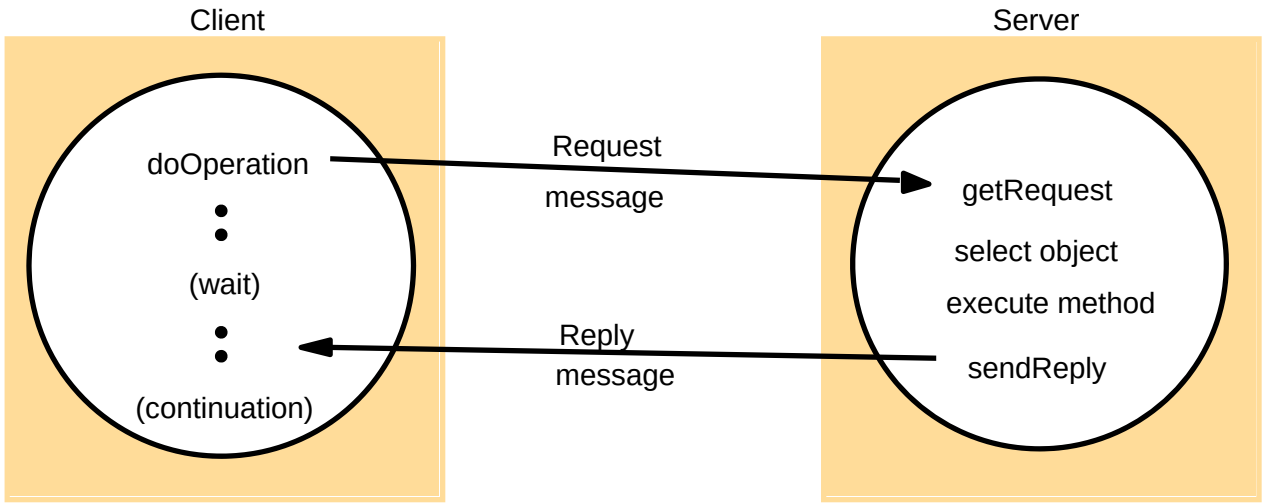


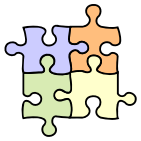
Un protocollo richiesta-risposta

- Scenario – comunicazione remota tra oggetti distribuiti
  - le richieste fatte ad oggetti remoti – identificati da un riferimento remoto
- Una possibile soluzione è basata su tre operazioni primitive (da implementare)
  - ***doOperation*** – usata dal client per specificare l'oggetto destinatario, l'operazione richiesta e i parametri – e per ottenere la risposta
  - ***getRequest*** – usata dal server per sapere qual è la prossima richiesta da soddisfare
  - ***sendReply*** – usata dal server per inviare la risposta alla richiesta corrente



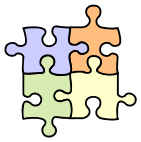
Protocollo richiesta-risposta





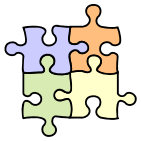
Legame dei parametri

- Il legame dei parametri avviene comunemente per valore/risultato
  - normalmente, l'esecuzione di un metodo remoto non può provocare effetti collaterali (sul client)
  - i parametri vengono allora legati per valore (*in*), per risultato (*out*) o per valore-risultato (*in-out*)
- Talvolta, i parametri/risultati di un'operazione sono oggetti
  - comunicare il riferimento all'oggetto parametro non è sufficiente – i riferimenti possono essere seguiti solo localmente
  - può essere opportuno codificare e comunicare l'oggetto parametro e il grafo degli oggetti da esso raggiungibile – serializzazione



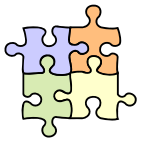
Che succede se uso UDP?

- Attenzione alla dimensione dei messaggi
  - in particolare, se è necessaria la serializzazione di oggetti
- Se le primitive **doOperation**, **getRequest** e **sendReply** sono implementate su UDP, sono possibili dei fallimenti
  - gestione di fallimenti di omissione
    - un client potrebbe decidere di ritrasmettere una richiesta a cui non ha ricevuto risposta entro un certo timeout
  - gestione di messaggi fuori ordine



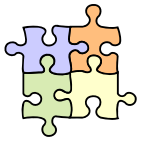
Che succede se uso UDP?

- Se messaggi persi vengono ritrasmessi
  - è possibile che il server riceva più volte la stessa richiesta
    - richieste duplicate possono essere riconosciute – ad es., associando identificatori alle richieste
  - un client potrebbe ripetere una richiesta perché non ha ricevuto la risposta
    - se il server non ha ancora risposto, scarica la richiesta duplicata e risponde a quella iniziale
    - se invece il server ha già inviato la risposta, allora potrebbe ritrasmettere la risposta senza eseguire nuovamente l'operazione – se la risposta è stata memorizzata
    - oppure il server potrebbe ripetere l'esecuzione dell'operazione – questo non crea problemi se l'operazione è idempotente – ma se non lo è?



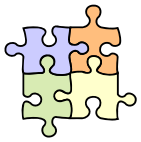
Che cambia se uso TCP?

- L'uso di TCP risolve molti dei problemi legati a UDP
- Tuttavia, talvolta viene usato UDP per implementare protocolli più efficienti
  - se non tutte le caratteristiche offerte da TCP sono richieste



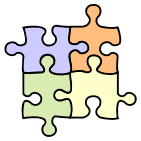
- Esercizi – per pensare

- Realizzare delle applicazioni client-server nei seguenti casi
  - server Daytime – per la consultazione dell'ora corrente
    - la richiesta non contiene dati
    - la risposta è una stringa – ad es., new Date().toString()
  - server Echo
    - la richiesta è una stringa
    - la risposta è la stessa stringa
  - server Math
    - la richiesta è composta da una stringa che denota un'operazione (ad es., *sqrt* o *max*) e da un certo numero di numeri reali (ad es., uno per *sqrt*, uno o più per *max*)
    - la risposta è normalmente un solo numero



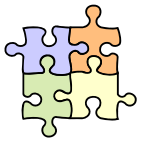
Esercizi – per pensare

- Realizzare delle applicazioni client-server nei seguenti casi
  - server Counter
    - la richiesta non contiene dati
    - la risposta è un numero progressivo *assoluto* – quante richieste sono state fatte finora al server?
  - server SessionCounter
    - la risposta è un numero progressivo *relativo* – quante richieste sono state fatte finora al server da questo client?
  - server DoubleCounter
    - quante richieste sono state fatte finora, complessivamente, al server? e quante richieste sono state fatte finora al server da questo client?
- Quali le possibili modalità di realizzazione di queste applicazioni?
  - ce ne sono più di una...



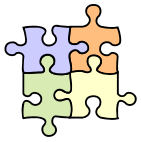
* Discussione

- Le socket consentono la comunicazione tra processi
  - ma ad un livello di astrazione basso (troppo basso?)
  - bisogna implementare (quasi) tutti gli aspetti della comunicazione
- Per questo, sono stati realizzati degli strumenti di middleware
  - per facilitare la comunicazione tra processi
  - per nascondere l'eterogeneità
    - nella posizione delle parti – stesso processo, processo diverso sullo stesso computer, computer diverso
    - nel protocollo di comunicazione – TCP, UDP
    - nella piattaforma hardware/sistema operativo
    - nel linguaggio di programmazione

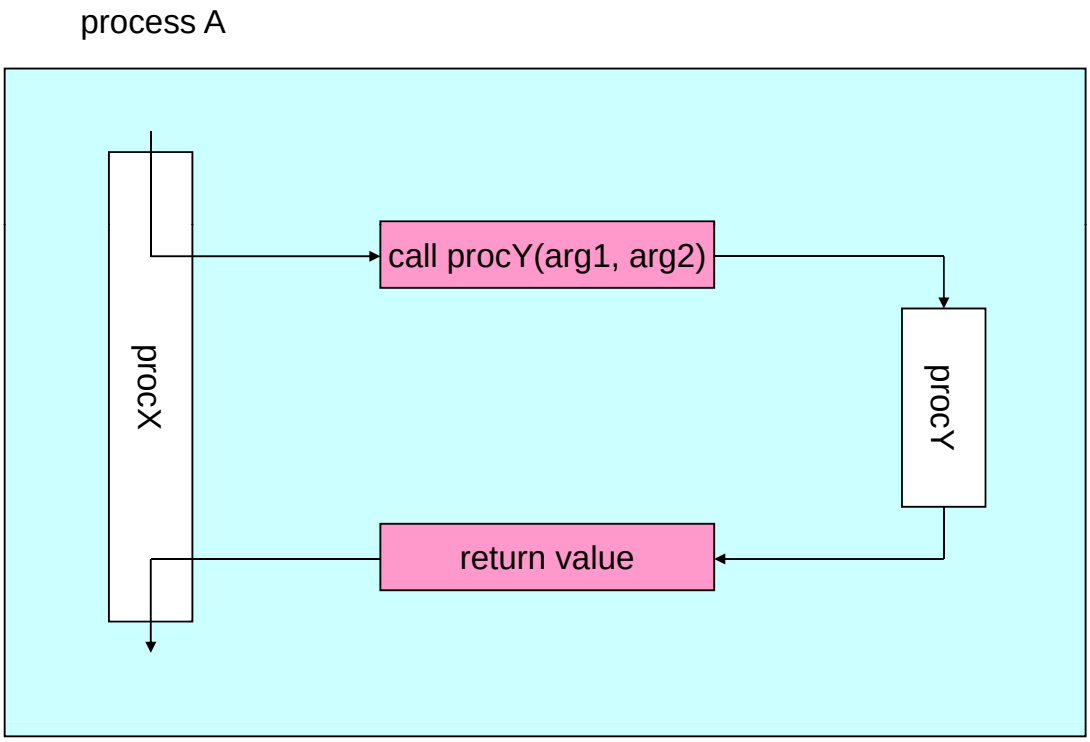


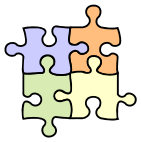
Esempio: RPC

- RPC (Remote Procedure Call)
  - Remote Procedure Call (RPC) is a technology that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer) without the programmer explicitly coding the details for this remote interaction [en.wikipedia.org]

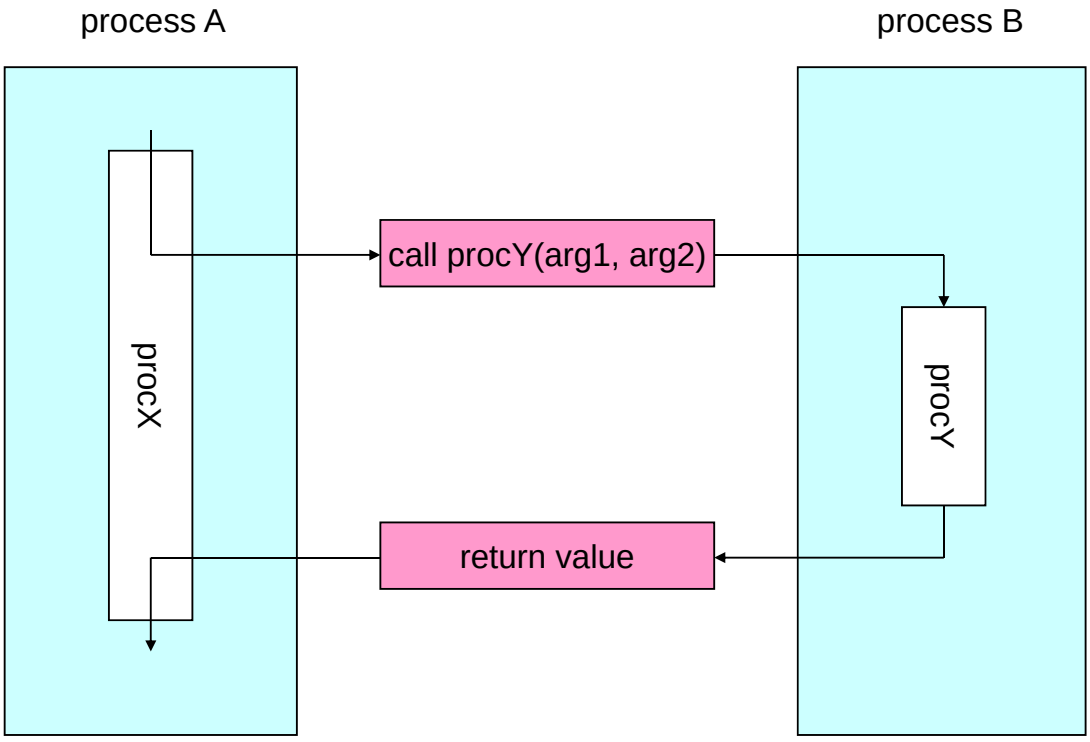


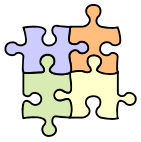
Chiamata di procedure (locale)





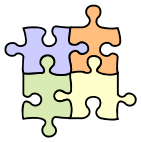
Paradigma dell'RPC





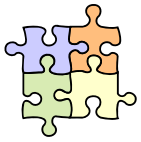
Esempio: RPC

- Uso di RPC
  - scrivo l'interfaccia del server
  - un compilatore d'interfacce produce del codice lato client (stub) e del codice lato server (skeleton) per le varie procedure
    - il client effettua una richiesta rivolgendosi allo stub – che fa marshalling ed inoltra il messaggio di richiesta
    - lo skeleton riceve il messaggio di richiesta, fa l'unmarshalling, e gira la richiesta al server
    - il server genera la risposta e la restituisce allo skeleton – che fa il marshalling e invia il messaggio di risposta
    - lo stub riceve il messaggio di risposta, fa l'unmarshalling, e gira la risposta al client



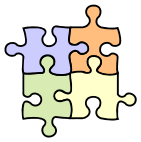
* Un'applicazione client/server a strati

- Un'applicazione Echo client-server basata su socket TCP con un'architettura a strati
  - lato client
    - presentazione – legge stringhe, chiama logica applicativa, visualizza risultato
    - logica applicativa – chiamata del servizio
    - comunicazione
  - lato server
    - logica applicativa – servizio
    - comunicazione



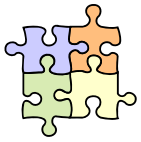
Client – presentazione

```
import java.util.Scanner;
import java.io.*;

/* Legge dalla tastiera una sequenza di stringhe e le invia al server Echo. */
public class EchoTcpClient {
    public static void main(String[] args) throws IOException {
        EchoTcpHelper helper = new EchoTcpHelper();
        Scanner in = new Scanner(System.in);
        System.out.println("Scrivi una sequenza di linee di testo, " +
                           "terminata da CTRL-C.");
        while (in.hasNextLine()) {
            System.out.print("# In: ");
            String line = in.nextLine();
            String result = helper.echo(line);
            System.out.println("Echo: " + result);
        }
        helper.done();
        System.out.println("Bye.");
    }
}
```

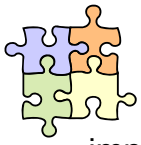
Client – logica applicativa (1)

```
import java.io.*;

/* Classe di supporto per l'accesso al servizio Echo. */
public class EchoTcpHelper {
    private static final String HOST = "localhost";
    private static final int PORT = 5687;
    private static final String END_MESSAGE = "##@@##";

    private MyTcpSocket mySocket;

    public EchoTcpHelper() throws IOException {
        mySocket = new MyTcpSocket(HOST, PORT);
    }

    /* invia una stringa e riceve la risposta dal server echo */
    public String echo(String s) throws IOException {
        mySocket.sendMessage(s);
        String result = mySocket.receiveMessage();
        return result;
    }
```



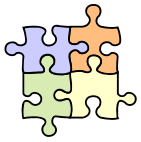
Client – logica applicativa (2)

```
        public void done() {
            try {
                mySocket.sendMessage(END_MESSAGE);
                mySocket.close();
            } catch(IOException e) {
                System.out.println("IO: " + e.getMessage());
            }
        }
    }
```

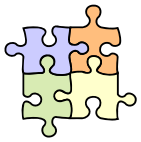
Client e server – comunicazione (1)

```
import java.net.*;
import java.io.*;

/* Socket TCP che consente l'invio e la ricezione di messaggi. */
public class MyTcpSocket {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;

    public MyTcpSocket(String host, int port) throws IOException {
        socket = new Socket(host, port);
        setStreams();
    }
    public MyTcpSocket(Socket socket) throws IOException {
        this.socket = socket;
        setStreams();
    }
    private void setStreams() throws IOException {
        in = new BufferedReader( new InputStreamReader( socket.getInputStream() ) );
        out = new PrintWriter( new OutputStreamWriter( socket.getOutputStream() ) );
    }
```



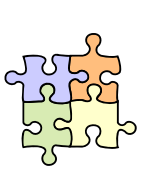
Client e server – comunicazione (2)

```
    public void sendMessage(String message) throws IOException {
        out.println(message);
        out.flush();
    }

    public String receiveMessage() throws IOException {
        String message = in.readLine();
        return message;
    }

    public void close() throws IOException {
        socket.close();
    }

}
```

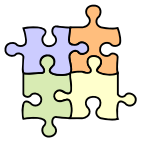
Server (1)

```
import java.net.*;
import java.io.*;

/* Server Echo TCP. */
public class EchoTcpServer {
    private static final String HOST = "localhost";
    private static final int PORT = 5687;

    public static void main(String[] args) {
        try {
            ServerSocket connectionSocket = new ServerSocket(PORT);
            System.out.println("Echo Server ready.");
            while (true) {
                MyTcpSocket dataSocket =
                        new MyTcpSocket( connectionSocket.accept() );
                EchoServerConnection c =
                        new EchoServerConnection(dataSocket);
            }
        } catch (IOException e) { System.out.println("IO: " + e.getMessage()); }
    }
}
```



Server (2)

```
import java.io.*;

public class EchoServerConnection extends Thread {
    private MyTcpSocket clientSocket;

    private static final String END_MESSAGE = "##@@##";

    public EchoServerConnection(MyTcpSocket clientSocket) {
        this.clientSocket = clientSocket;
        this.start();    // esegue run() in un nuovo thread
    }
```

Server (3)

```
    public void run() {
        boolean done = false;
        try {
            while (!done) {
                String data = clientSocket.receiveMessage();
                if (END_MESSAGE.equals(data)) {
                    clientSocket.close();
                    System.out.println("Connection closed.");
                    done = true;
                } else {
                    System.out.println("Received: " + data);
                    String result = data;
                    clientSocket.sendMessage(result);
                    System.out.println("Sent: " + result);
                }
            }
        } catch (IOException e) {
            System.out.println("IO: " + e.getMessage());
        }
    }
}
```