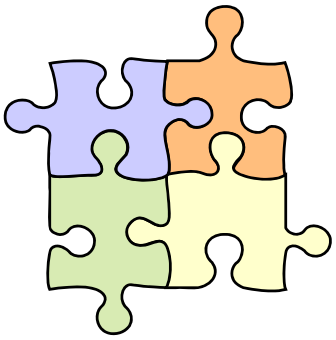
Luca Cabibbo



Architetture Software

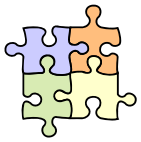
Comunicazione interprocesso e socket

Dispensa ASW 820

ottobre 2014

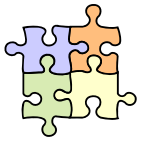
Ricordati di chiedere:
"Qual è la cosa peggiore
che altri elementi potrebbero fare
attraverso l'interfaccia?"

B. Kuchta



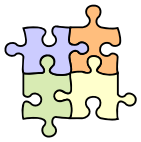
- Fonti

- [CDK/4e] Coulouris, Dollimore, Kindberg, Distributed Systems, Concepts and Design, 4th edition, 2005
  - Chapter 4, Interprocess Communication
- [Liu] Liu, Distributed Computing – Principles and Applications, Pearson Education, 2004
  - Chapter 2, Interprocess Communication
  - Chapter 4, The Socket API
  - Chapter 5, The Client-Server Paradigm

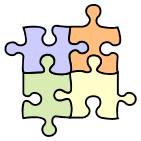


- Obiettivi e argomenti

- Obiettivi
  - richiamare il meccanismo dei socket
  - esemplificare l'uso dei socket nello sviluppo di semplici connettori per sistemi distribuiti
  - comprendere i limiti dei socket, nonché alcuni aspetti e problemi di base della comunicazione nei sistemi distribuiti
- Argomenti
  - comunicazione interprocesso
  - un'applicazione client-server UDP
  - un'applicazione client-server TCP
  - messaggi da scambiare
  - discussione

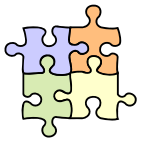


- Wordle



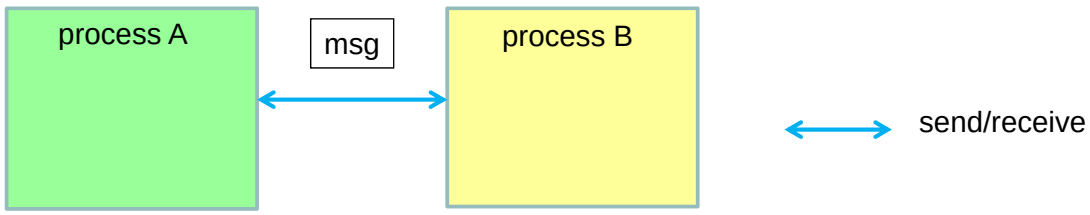
* Comunicazione interprocesso

- La spina dorsale dei sistemi distribuiti è la **comunicazione interprocesso** – **IPC**, *interprocess communication*
  - i sistemi operativi offrono uno o più servizi di base per la comunicazione interprocesso
    - ad esempio, pipe e socket
  - alcuni servizi di IPC sono basati su protocolli di Internet
    - ad esempio, TCP e UDP
  - i servizi di IPC forniti dai sistemi operativi sono alla base della realizzazione di meccanismi di comunicazione più sofisticati, gli strumenti di middleware
    - ciascun servizio di middleware offre un'opportuna astrazione di programmazione di supporto alla realizzazione dei sistemi distribuiti

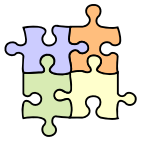


Comunicazione interprocesso di base

- Ci concentriamo sulla comunicazione interprocesso di tipo unicast

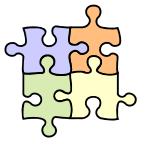


  - una coppia di processi indipendenti – che possono essere sullo stesso computer o su computer diversi – si scambiano dati (*messaggi*) – usando le primitive ***send*** e ***receive*** – ed eventualmente una rete di interconnessione
  - è necessario basare la comunicazione su un *protocollo* – formato da messaggi e da regole, anche di sincronizzazione – accettato da entrambe le parti



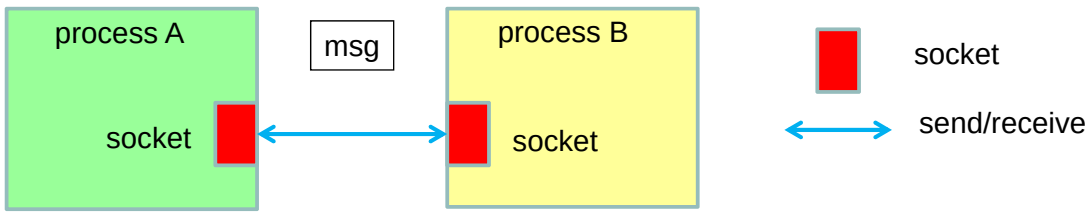
Primitive send e receive

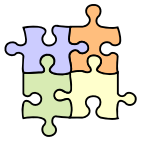
- In generale, lo scambio di messaggi tra processi può essere basato su due operazioni primitive
  - ***send*** – consente a un processo di trasmettere dati a un altro processo
    - ***send(receiving process, data)***
  - ***receive*** – consente a un processo di accettare dati trasmessi da un altro processo
    - ***receive(sending process, *buffer)***
  - entrambe le primitive sono definite in termini di
    - processo che le esegue – mittente o destinazione
    - l'altro processo – destinazione o mittente
    - un messaggio – a questo livello di astrazione, per *messaggio* si intende semplicemente una qualche sequenza di dati, binaria o testuale



Socket

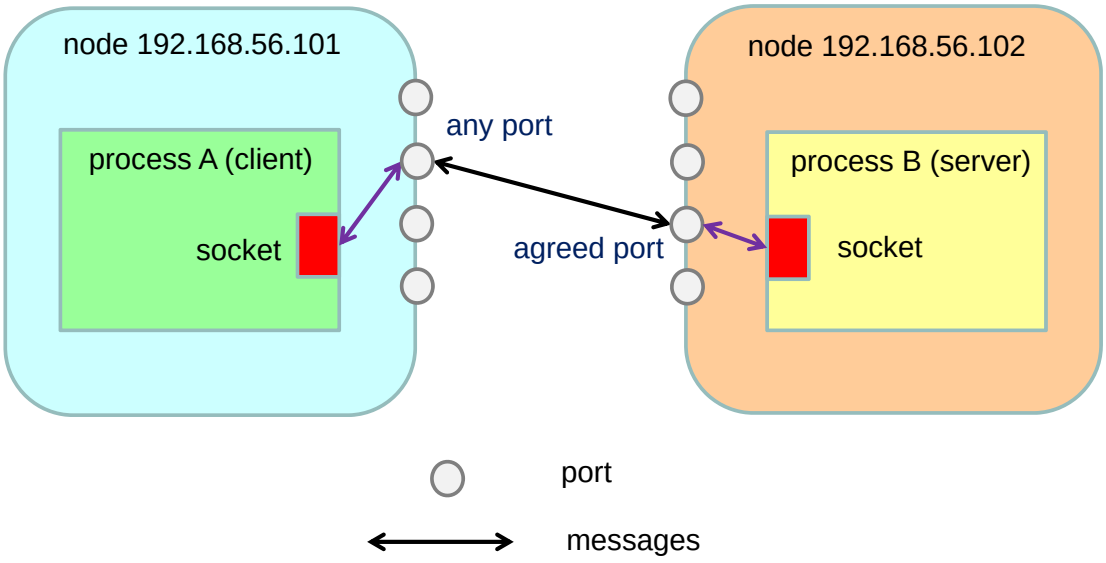
- Nel seguito faremo riferimento ai **socket** – un'astrazione di programmazione (ovvero, un'API standard) per lo scambio di messaggi in rete tramite UDP e/o TCP
  - un socket fornisce un endpoint per la comunicazione tra processi – letteralmente, socket = connettore, presa
  - l'astrazione di comunicazione interprocesso fornita dai socket consiste nella possibilità di inviare un messaggio tramite un socket di un processo e ricevere il messaggio tramite un socket di un altro processo

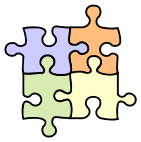




Socket – modello fisico

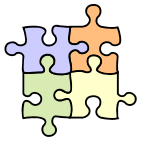
- In pratica, ciascun socket deve essere legato a una porta di un computer (in rete)
  - ogni socket ha un indirizzo (*indirizzo IP*, *numero di porta*)





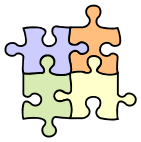
UDP e TCP

- I socket consentono la comunicazione tra processi con riferimento ai protocolli (a livello di trasporto) UDP e TCP
- *UDP* è un protocollo per il trasporto di datagrammi – senza ack e ritrasmissione
  - di tipo “best effort” – ovvero, non offre garanzie di consegna – i datagrammi potrebbero venire persi (a causa di errori di rete o scartati per se la rete è congestionata) o consegnati fuori ordine
  - connectionless – l’overhead è basso – non è richiesto setup
- *TCP* è un servizio di trasporto più sofisticato
  - consegna “affidabile” (?, vedi dopo) di uno stream ordinato di dati – uso di numerazione di pacchetti, checksum, ack e ritrasmissione
  - connection-oriented – richiesto il setup di un canale di comunicazione bidirezionale – l’overhead è più alto



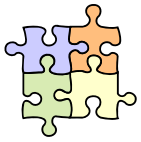
Primitive connect e disconnect

- Due ulteriori primitive per l'IPC connection-oriented
  - ***connect*** – per stabilire una connessione logica tra due processi
    - request-to-connect + accept-connection
    - per allocare delle risorse per gestire la connessione
  - ***disconnect*** – per terminare una connessione logica su entrambi i lati della comunicazione
    - per deallocare le risorse per la gestione della connessione
- Avvertenza
  - la comunicazione connection-oriented è più costosa sia in termini di occupazione di memoria che di consumo di CPU
  - questo può anche limitare la scalabilità di un servizio – ovvero il numero massimo di sessioni/connessioni che possono essere attive nello stesso momento – a meno che non vengano implementate soluzioni opportune, ad es., connection pooling



- Sincronizzazione

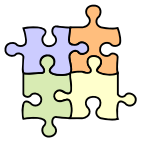
- La comunicazione tra processi può essere sincrona o asincrona
  - **comunicazione sincrona**
    - lo scambio di un messaggio costituisce un punto di sincronizzazione tra i processi comunicanti
    - le operazioni ***send*** e ***receive*** sono *bloccanti*
    - i dati trasmessi devono essere stati 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*



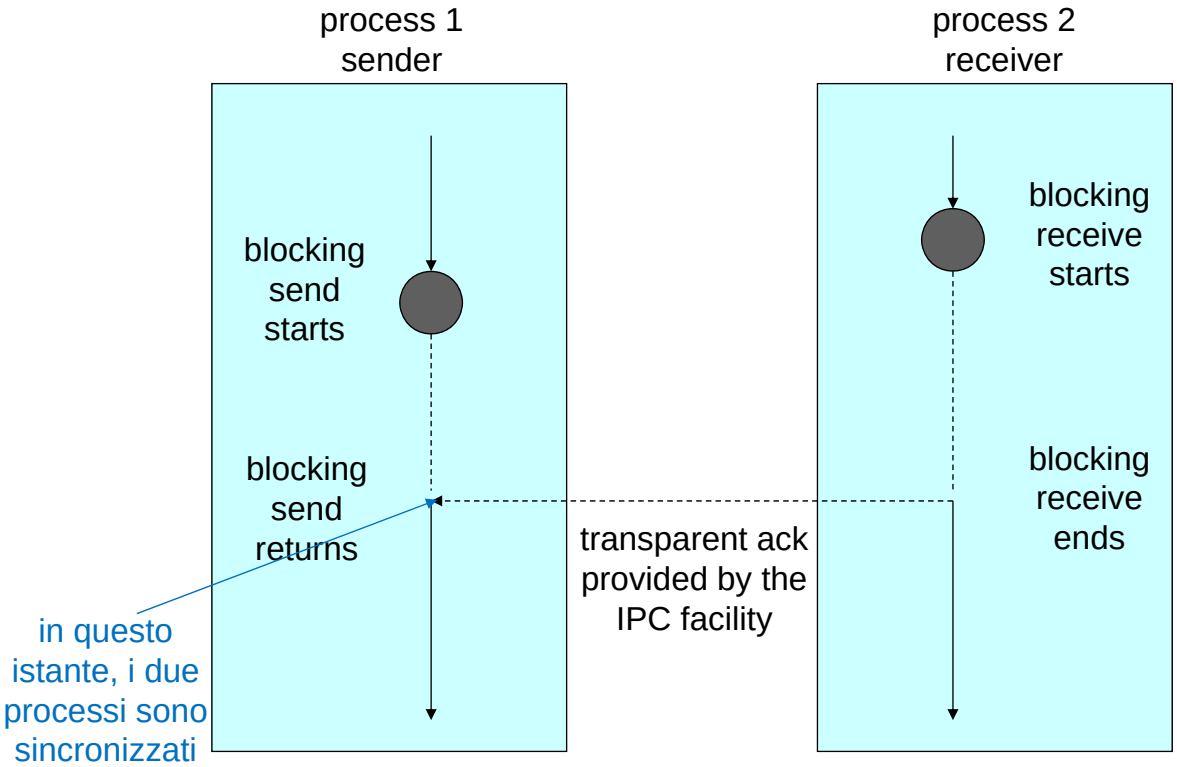
Sincronizzazione

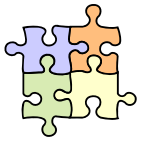
- La comuni… …sincrona
  - **comu…**
    - …
    - i dati t… …ti ricevuti prima di poter andare avanti
  - **comunicazione asincrona**
    - nella comunicazione asincrona, l'operazione ***send*** è *non bloccante* – il messaggio viene copiato in un buffer, e poi il processo mittente può proseguire, mentre il messaggio viene trasmesso
    - l'operazione ***receive*** è normalmente *bloccante*

Attenzione: Più avanti nel corso parleremo di comunicazione asincrona con un significato un po' diverso da questo.

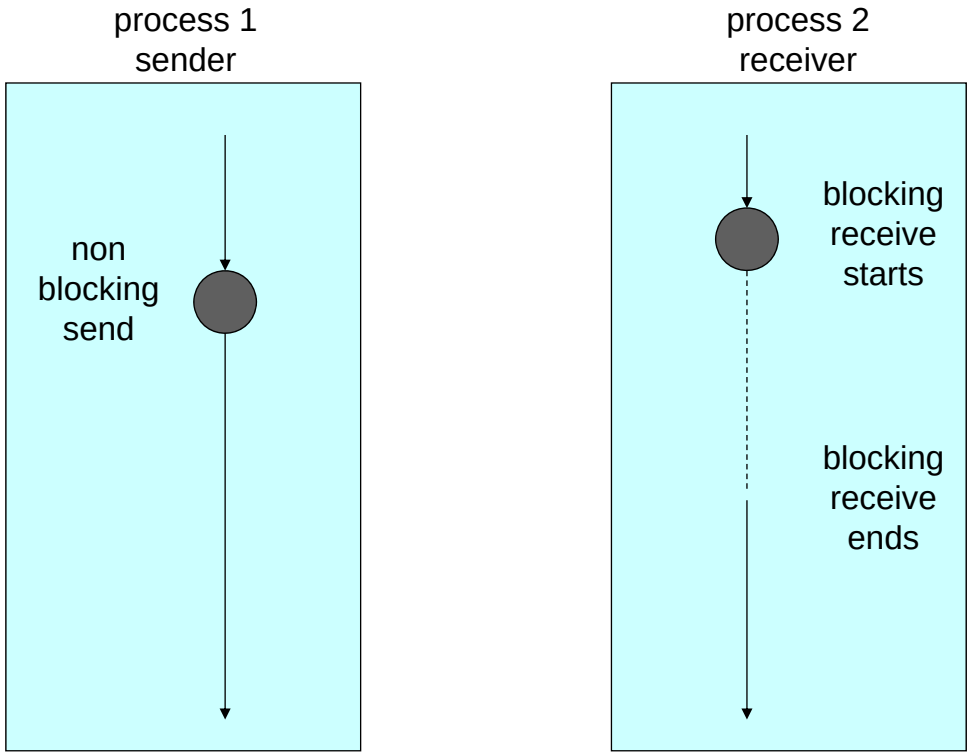


Comunicazione sincrona

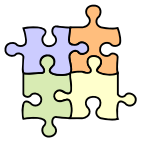




Comunicazione asincrona

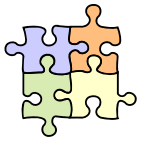


non c'è nessun istante in cui i due processi sono sincronizzati



Sincronizzazione

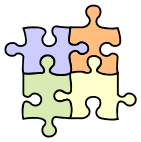
- Processi che operano in modo concorrente e che comunicano devono in genere provvedere a una loro opportuna *sincronizzazione*
  - ad es., un processo client che fa una richiesta ad un processo server vuole normalmente sapere se la richiesta è stata elaborata o meno
    - che fare se la richiesta viene fatta con una ***send*** non bloccante?
  - in genere, i programmi concorrenti devono sincronizzarsi per fornire garanzie di
    - *safety* – non succede niente di negativo (ad es., stalli)
    - *liveness* – succede qualcosa di positivo (la computazione avanza)



Sincronizzazione

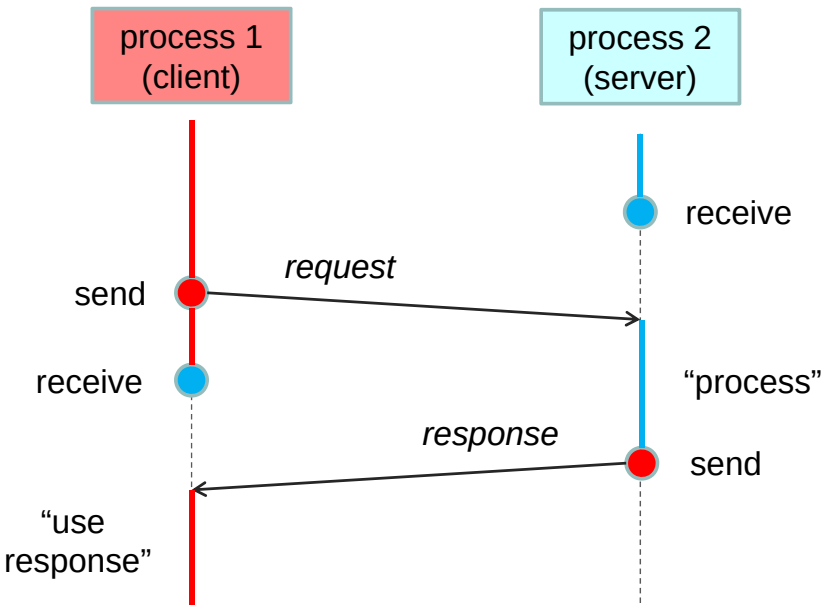
- Processi che operano in modo concorrente e che comunicano devono in genere provvedere a una loro opportuna *sincronizzazione*
  - ad es., un proc… un processo server v… stata elabor…
    - che… n
  - i… si per fo…
    - …stalli)
    - …mputazione avanz…

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



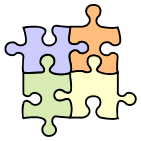
Protocollo richiesta-risposta

- Una possibile (e comune) modalità di sincronizzazione (parziale) è data dai protocolli *richiesta-risposta*



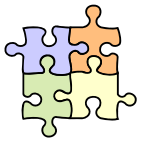
protocollo richiesta-risposta, con receive bloccante e send non bloccante

la sincronizzazione è parziale: il server non sa se il client ha ricevuto la sua risposta



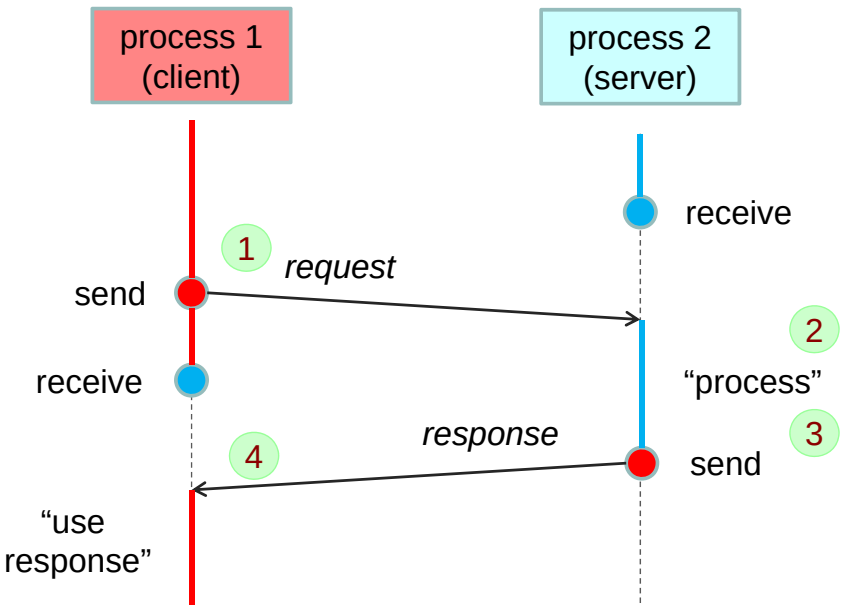
- Affidabilità

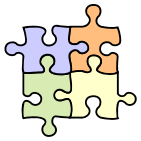
- Un problema comune nei sistemi distribuiti è legata al fatto che si possono verificare dei *fallimenti* nella comunicazione
  - i fallimenti possono essere causati da
    - guasti (crash) nei processi che comunicano
    - guasti nei canali di comunicazione
  - alcune possibili conseguenze
    - messaggi persi
    - messaggi trasmessi male o trasmessi più volte
    - messaggi trasmessi fuori ordine
  - una *network* può diventare una *not-work*



Affidabilità

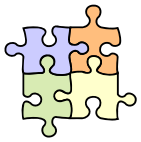
- Il client e il server possono fallire – e possono fallire in momenti e modi diversi – con conseguenze differenti





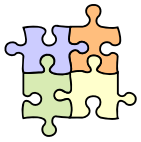
Affidabilità

- Il client e il server possono fallire – e possono fallire in in momenti modi diversi – con conseguenze differenti
  - se il fallimento avviene prima della richiesta 1, niente è ancora successo
  - il fallimento può avvenire tra la richiesta 1 e la sua ricezione 2 – richiesta persa
  - il fallimento può avvenire tra la ricezione 2 e la generazione della risposta 3 – l'azione richiesta potrebbe essere stata eseguita parzialmente, con effetti collaterali (parziali) sul server e inconsistenze con il client
  - il fallimento può avvenire tra la generazione della risposta 3 e la sua ricezione 4 – azione eseguita, ma risposta persa
- Che fare in ciascuno di questi casi?
  - chi è responsabile di capire che cosa è successo? come gestire le diverse situazioni?



Affidabilità

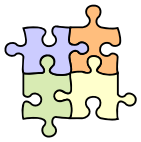
- Alcune proprietà/forme di *affidabilità* che *potremmo* volere
  - *validità*
    - garanzia di consegna dei messaggi – anche a fronte dell'eventuale perdita di alcuni pacchetti
  - *integrità*
    - i messaggi ricevuti sono identici a quelli trasmessi, senza duplicazioni di messaggi
  - *ordine*
    - messaggi consegnati nell'ordine in cui sono stati trasmessi
- Va notato che non tutti i servizi di comunicazione garantiscono tutte queste forme di affidabilità
  - ma anche che non sempre sono richieste tutte queste proprietà



Affidabilità

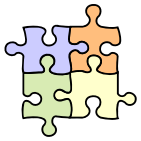
- Alcune proprietà/forme di *affidabilità* che *potremmo* volere
  - *validità*
    - garanzia di consegna messaggi – anche a fronte dell'eventuale pe
  - *integrità*
    - i m senza
  - c messi
- Va n antiscono tutte q
  - ma anch te queste proprietà

Anche se UDP non garantisce direttamente tutte queste proprietà, questo non vuol dire che UDP non possa essere utilizzato nella comunicazione interprocesso, anche se queste proprietà fossero richieste...



Affidabilità di TCP

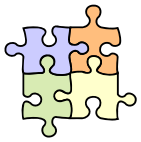
- TCP offre garanzie di affidabilità maggiori rispetto a UDP
  - messaggi ricevuti in ordine, ritrasmissione automatica di pacchetti corrotti (checksum) oppure persi (ack), pacchetti ricevuti duplicati vengono scaricati
- Tuttavia, in alcune situazioni – la perdita di pacchetti è eccessiva, oppure la rete è partizionata oppure è molto congestionata – è possibile che il software per TCP non riceva degli ack e dichiari la connessione broken [CDK/4e]
  - i processi comunicanti non possono sapere se i messaggi che hanno inviato sono stati consegnati o meno – e non sanno distinguere tra fallimento della rete e fallimento dell'altro processo remoto
  - pertanto, nemmeno la comunicazione TCP è completamente affidabile – ovvero, non garantisce la consegna dei messaggi a fronte di ogni possibile difficoltà



Affidabilità di TCP

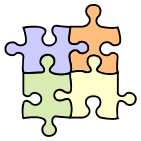
- TCP offre garanzie di affidabilità maggiori rispetto a UDP
  - messaggi ricevuti in ordine, [illegible]asmissione automatica di pacchetti corrotti (checksun[illegible]ppure persi (ack), pacchetti ricevuti duplicati ver[illegible]
- Tuttavia, [illegible]cessiva, oppu[illegible]a – è pos[illegible]hiari la conne[illegible]
  - i p[illegible]messaggi che hanno in[illegible]e non sanno distinguere tra fallimen[illegible]e fallimento dell'altro processo remoto
  - pertanto, nemmeno la comunicazione TCP è completamente affidabile – ovvero, non garantisce la consegna dei messaggi a fronte di ogni possibile difficoltà

Dunque, valuta con attenzione quanto viene detto – e poniti dei dubbi sull'apparenza delle cose e su quanto non viene detto…



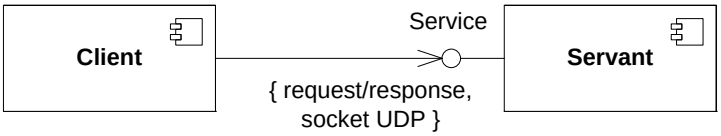
Affidabilità

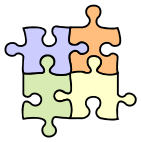
- Un servizio di comunicazione – in particolare, un servizio di middleware
  - può mascherare alcuni dei possibili fallimenti – realizzando un canale di comunicazione che garantisce un certo livello di affidabilità – ma probabilmente mai un'affidabilità completa
- Dunque, quando usi un servizio di comunicazione/middleware, chiediti
  - il servizio di comunicazione in uso è affidabile? in che senso?
  - quali garanzie di affidabilità sono possibili? come le ottengo?
  - se non è affidabile come vorrei, posso comunque ottenere il livello di affidabilità necessario?
  - a quale "prezzo"?



* Un'applicazione client/server UDP

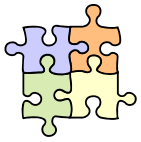
- Viene ora mostrata un'applicazione basata su socket UDP – di tipo client/server, basata su un protocollo richiesta/risposta
  - la struttura dell'applicazione è simile (ma un po' più complessa) a quella mostrata nel contesto dell' "Introduzione ai connettori e al middleware" – con
    - un servizio ***Service***
    - un elemento ***Servant*** in grado di erogare il servizio ***Service***
    - un elemento ***Client***, che richiede l'erogazione del servizio ***Service***



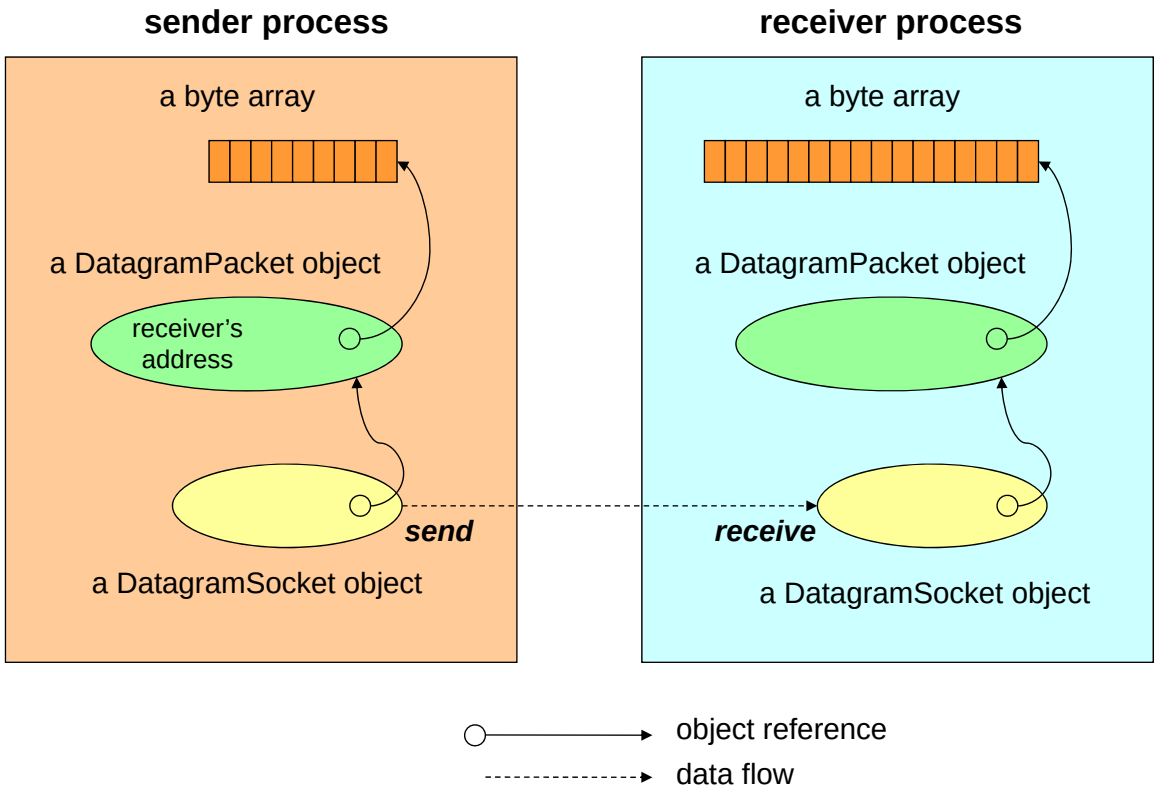


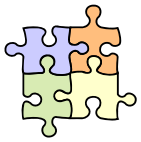
- API di Java per socket UDP

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



API di Java – strutture di dati coinvolte





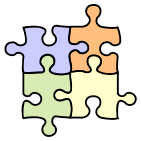
Programmi sender e receiver

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

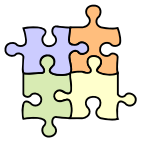
receiver program

- 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



API: DatagramPacket e DatagramSocket

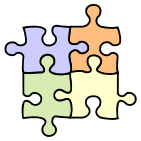
- DatagramPacket(byte[] buffer, int length)
  - crea un DatagramPacket per la ricezione di pacchetti
- DatagramPacket(byte[] buffer, int length, InetAddress address, int port)
  - crea un DatagramPacket per l'invio di pacchetti alla socket specificata (address rappresenta un indirizzo IP)
- DatagramSocket()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



- Il servizio

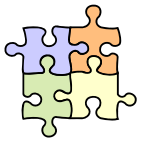
- La seguente definizione del ***Service*** caratterizza l'interfaccia funzionale del servente

```
package asw.asw820.service;

/* Interfaccia del servizio Service. */
public interface Service {
    public String alpha(String arg) throws ServiceException, RemoteException;
    public String beta(String arg) throws ServiceException, RemoteException;
}
```



Eccezioni

- Nella definizione di un servizio remoto è di solito necessario far riferimento a due tipologie di eccezioni
  - eccezioni legate al servizio, di natura “funzionale”
    - ad es., se la precondizione di un’operazione non è soddisfatta, allora non è possibile erogare il servizio – si solleva un’eccezione per segnalarlo
  - eccezioni legate alla natura distribuita del servizio, ma non di natura “funzionale”
    - ad es., non è possibile fruire del servizio, ma per un problema sul canale di comunicazione

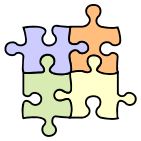


Eccezioni

- Le eccezioni per ***Service***

```
package asw.asw820.service;

/** ServiceException rappresenta un'eccezione "funzionale" legata al servizio. */
public class ServiceException extends Exception {
    public ServiceException(String message) {
        super(message);
    }
}
```

```
package asw.asw820.service;

/** RemoteException indica un problema nell'accesso remoto al servizio Service. */
public class RemoteException extends Exception {
    public RemoteException(String message) {
        super(message);
    }
}
```



Il servizio

- Ecco la definizione del ***Servant***, che completa gli aspetti funzionali

```
package asw.asw820.server;

import asw.asw820.service.*;

/* Implementazione del servizio Service. */
public class Servant implements Service {
    public String alpha(String arg) throws ServiceException { ... fa qualcosa ... }
    public String beta(String arg) throws ServiceException { ... fa qualcosa ... }
}
```

  - notare che l'implementazione del servente non solleva mai l'eccezione remota ***RemoteException***
  - come vedremo, questa eccezione può essere invece sollevata da un remote proxy, se questo rileva un problema nella comunicazione remota

Un client del servizio

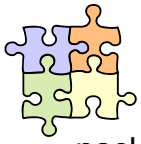
```
package asw.asw820.client;

import asw.asw820.service.*;

/* client del servizio */
public class Client {

    private Service service;

    public Client(Service service) {  this.service = service;  }

    public void run(...) {
        try {
            ... service.alpha(...) ...
        } catch (ServiceException e) { ... gestisci e ... }
        } catch (RemoteException e) { ... gestisci e ... }
    }

}
```

Un client del servizio

```
package asw.asw820.client;

import asw.asw820.service.*;
import asw.asw820.client.connector.*;

/* Applicazione client: ottiene e avvia il client. */
public class Main {

    /* Crea e avvia un oggetto Client. */
    public static void main(String[] args) {
        Service service = ServiceFactory.getInstance().getService();
        /* crea il client e gli inietta il servizio da cui dipende */
        Client client = new Client(service);
        client.run();
    }

}
```

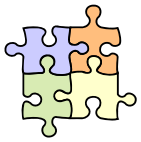

Factory per il proxy lato client

```
package asw.asw820.client.connector;

import asw.asw820.service.*;
import java.net.*;

/* Factory per il servizio service (lato client). */
public class ServiceFactory {
   ... variabile e metodo per singleton ...

   /* Factory method per il servizio Service. */
   public Service getService() {
      Service service = null;
      try {
         InetAddress address = InetAddress.getByName("192.168.56.99");
         int port = 6789;
         service = new ServiceClientUDPProxy(address, port);
      } catch(Exception e) { e.printStackTrace(); }
      return service;
   }
}
```



Il proxy lato client (1)

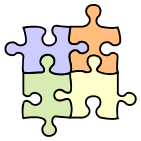
```
package asw.asw820.client.connector;

import asw.asw820.service.*;
import java.net.*;

/* Remote proxy lato client per il servizio Service. */
public class ServiceClientUDPProxy implements Service {
    private InetAddress address;    // indirizzo del server
    private int port;               // porta per il servizio

    public ServiceClientUDPProxy(InetAddress address, int port) {
        this.address = address;    this.port = port;
    }

    public String alpha(String arg) throws ServiceException, RemoteException {
        ... vedi dopo ...
    }
    public String beta(String arg) throws ServiceException, RemoteException {
        ... simile ad alpha ...
    }
}
```

Il proxy lato client (2)

- I metodi ***alpha*** e ***beta*** del “remote proxy” lato client
  - stiamo assumendo che il servizio sia definito in termini di più operazioni – ma, per semplicità, che queste operazioni abbiano tutte un solo parametro (una stringa) e che restituiscano tutte un solo valore (una stringa)

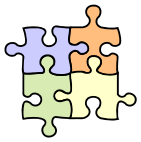
```
/* questo è proprio il metodo alpha invocato dal client
 * (anche se il client pensa di parlare direttamente con il servant) */
public String alpha(String arg) throws ServiceException, RemoteException {
    return doOperation("alpha", arg);
}

/* questo è proprio il metodo beta invocato dal client
 * (anche se il client pensa di parlare direttamente con il servant) */
public String beta(String arg) throws ServiceException, RemoteException {
    return doOperation("beta", arg);
}
```



Il proxy lato client (3)

- Il metodo ***doOperation*** per un client UDP

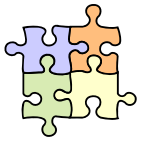
```
/* metodo di supporto che gestisce la richiesta remota di un'operazione */
private String doOperation(String op, String arg)
        throws throws ServiceException, RemoteException {
    String result = null;
    try {
        ... crea un datagramma che codifica la richiesta di servizio
            ed i relativi parametri ...
        ... invia il datagramma di richiesta ...
        ... ricevi il datagramma di risposta ...
        ... estrai il risultato dal datagramma di risposta ...
    } catch(Exception e) {
        prova a gestire l'eccezione, oppure, più semplicemente:
        throw new RemoteException("Client Proxy: " + e.getMessage());
    }
    return result;
}
```

Il proxy lato client (4)

- Una possibile implementazione di ***doOperation***

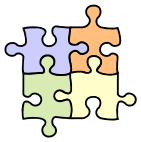
```
/* crea il socket per la comunicazione remota */
DatagramSocket socket = new DatagramSocket();
/* crea il datagramma per codificare la richiesta di servizio
 * e i suoi parametri – nella forma "operazione$parametro" */
String request = op + "$" + arg;
byte[] requestMessage = request.getBytes();
DatagramPacket requestPacket =
    new DatagramPacket(requestMessage, requestMessage.length,
                       this.address, this.port);
/* invia il datagramma di richiesta */
socket.send(requestPacket);    // non bloccante
```



Il proxy lato client (5)

- Una possibile implementazione di ***doOperation***

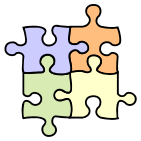
```
/* ricevi il datagramma di risposta */
byte[] buffer = new byte[1000];
DatagramPacket replyPacket = new DatagramPacket(buffer, buffer.length);
socket.receive(replyPacket);    // bloccante
/* estrai la risposta dal datagramma di risposta */
String reply = new String( replyPacket.getData(), replyPacket.getOffset(),
                                        replyPacket.getLength() );
```

Il proxy lato client (6)

- Una possibile implementazione di ***doOperation***

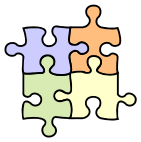
```
/* elabora la risposta, che può avere
 * le seguenti forme (vedi proxy lato server):
 * "#risultato" oppure "@messaggio per eccezione" */
if ( reply.startsWith("#") ) { /* è un risultato */
        result = reply.substring(1);
} else if ( reply.startsWith("@") ) { /* si è verificata una ServiceException */
        String message = reply.substring(1);
        throw new ServiceException(message);
} else { /* risposta malformata, solleva una RemoteException */
        throw new RemoteException("Malformed reply: " + reply);
}
return result;
```



Il server

- L'oggetto ***Server*** – in esecuzione su 192.168.56.99 – è responsabile di
  - creare il ***Servant***
  - creare e avviare il proxy lato server

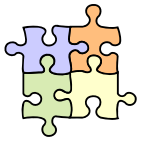
```
package asw.asw820.server.connector;

import asw.asw820.service.*;
import asw.asw820.server.Servant;

/* server per il servizio */
public class Server {
    public static void main(String[] args) {
        Service service = new Servant();
        int port = 6789;
        ServiceServerUDPProxy server =
                new ServiceServerUDPProxy(service, port);
        server.run();
    }
}
```

Il proxy lato server (1)

- Struttura del "remote proxy" lato server

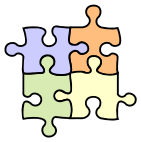
```
package asw.asw820.server.connector;

import asw.asw820.service.*;

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

/* Remote proxy lato server per il servizio Service. */
public class ServiceServerUDPProxy {
    private Service service;          // il vero servizio
    private int port;                 // porta per il servizio

    public ServiceServerUDPProxy(Service service, int port) {
        this.service = service;    this.port = port;
    }

    public void run() {  ... segue ...  }
}
```



Il proxy lato server (2)

- Il metodo ***run*** del “remote proxy” lato server
  - ora, per semplicità, un server “sequenziale”

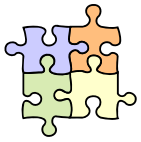
```
public void run() {
    try {
        /* crea il socket su cui ricevere le richieste */
        DatagramSocket socket = new DatagramSocket(this.port);
        byte[] buffer = new byte[1000];
        while (true) {
            getRequestAndSendReply(socket, buffer);
        }
    } catch(Exception e) {
        ... gestisci eccezione e ...
    }
}
```

Il proxy lato server (3)

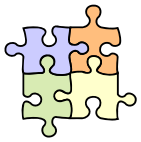
- Il metodo ***getRequestAndSendReply*** del “remote proxy” lato server

```
private void getRequestAndSendReply(DatagramSocket socket, byte[] buffer);
    try {
        ... aspetta un datagramma di richiesta ...
        ... estrai la richiesta dal datagramma di richiesta ...
        ... chiedi l'erogazione del servizio, ottieni il risultati, calcola la risposta ...
        ... crea il datagramma di risposta ...
        ... invia il datagramma di risposta ...
    } catch(Exception e) {
        ... gestisci eccezione e ...
    }
```



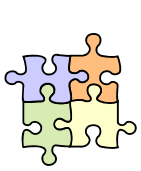
Il proxy lato server (4)

```
/* aspetta un datagramma di richiesta */
DatagramPacket requestPacket =
    new DatagramPacket(buffer, buffer.length);
socket.receive(requestPacket);    // bloccante

/* estrai la richiesta dal datagramma di richiesta */
String request =
    new String( requestPacket.getData(),
                requestPacket.getOffset(), requestPacket.getLength() );
/* la richiesta ha la forma "operazione$parametro" */
int sep = request.indexOf("$");
String op = request.substring(0,sep);
String arg = request.substring(sep+1);
```

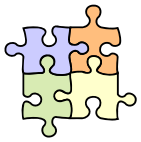
Il proxy lato server (5)

```
/* chiedi l'erogazione del servizio, ottieni il risultato, calcola la risposta */
String reply = null;
try {
    String result = this.executeOperation(op, arg);
    /* se siamo ancora qui, operazione completata, rispondi "#risultato" */
    reply = "#" + result;
} catch (ServiceException e) {
    /* se siamo qui, operazione NON completata, rispondi "@messaggio" */
    reply = "@" + e.getMessage();
} catch (RemoteException e) {
    /* no, il servente non dovrebbe mai sollevare RemoteException */
    reply = "@" + e.getMessage();
}
```



Il proxy lato server (6)

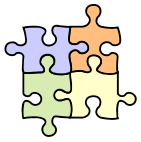
```
/* crea il datagramma di risposta */
byte[] replyMessage = reply.getBytes();
DatagramPacket replyPacket =
    new DatagramPacket( replyMessage, replyMessage.length,
                        requestPacket.getAddress(),
                        requestPacket.getPort() );
/* invia il datagramma di risposta */
socket.send(replyPacket);    // non bloccante
```

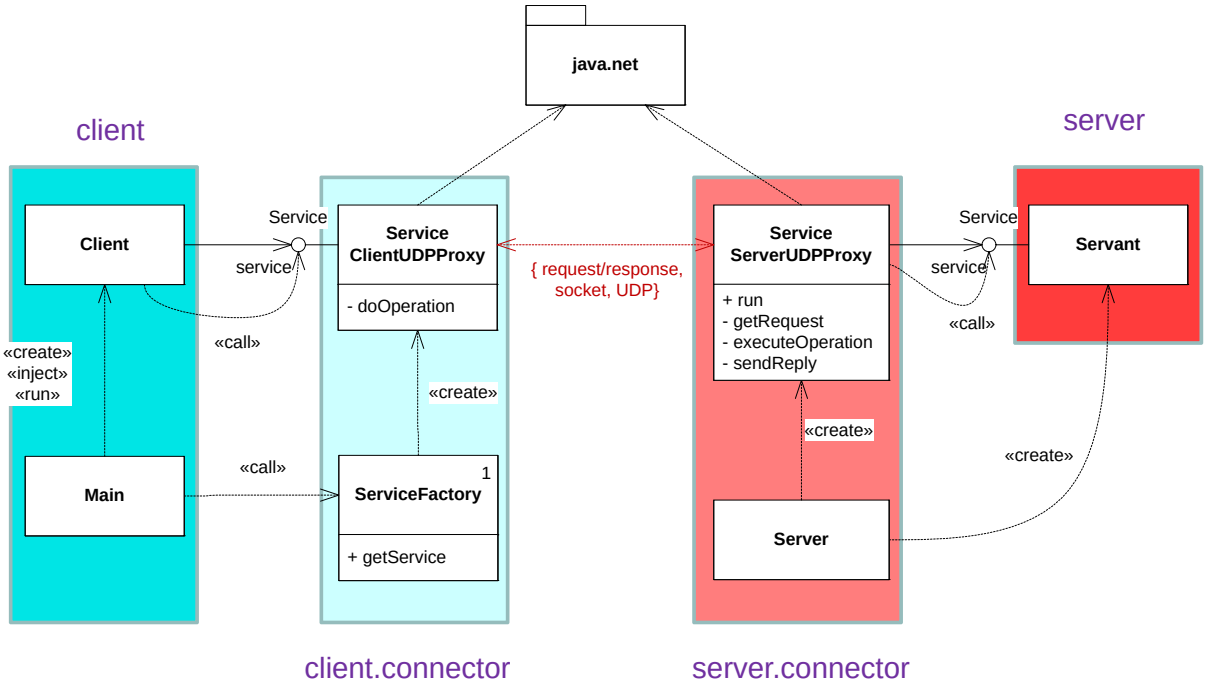
Il proxy lato server (7)

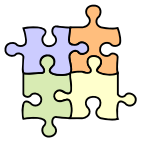
- Il metodo ***executeOperation*** del “remote proxy” lato server
  - ipotesi (semplificativa): il servizio è definito in termini di più operazioni – ma, per semplicità, tutte queste operazioni hanno un solo parametro (una stringa) e restituiscono un solo valore (una stringa)

```
private String executeOperation(String op, String arg)
        throws ServiceException, RemoteException {
    String result = null;

    if ( op.equals("alpha") ) {
        result = service.alpha(arg);
    } else if ( op.equals("beta") ) {
        result = service.beta(arg);
    } else {
        throw new RemoteException("Operation " + op + " is not supported");
    }

    return result;
}
```



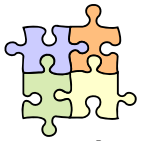
In sintesi





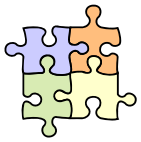
UDP – Discussione

- Concentriamoci solo su una caratteristica di questa soluzione, legata all'uso di UDP
  - poiché UDP è senza ack né ritrasmissione, possono verificarsi problemi di
    - integrità – messaggi ricevuti diversi da quelli trasmessi
    - omissione – si possono perdere messaggi
    - ordine – i messaggi possono arrivare fuori ordine
    - duplicazione – possono arrivare messaggi duplicati
    - ...
- È possibile usare UDP per definire un proprio protocollo di comunicazione affidabile?
  - ovvero, è possibile gestire i fallimenti di cui sopra? come si possono gestire? chi li deve gestire?



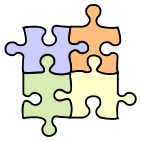
UDP – Discussione

- È possibile usare UDP 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 (con cautela!)
  - 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
  - ...
- Tutto ciò, nel connettore



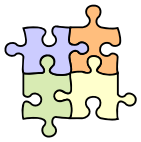
UDP – Discussione

- Ulteriori aspetti interessanti
  - *server concorrente*
    - con UDP è anche possibile realizzare un server concorrente – multithreaded – in cui i diversi thread condividono il socket che utilizzano per comunicare
    - vedremo un esempio di server concorrente con riferimento a TCP
  - servizi stateless e stateful
    - un servizio è *stateless* se non deve gestire lo stato della conversazione con i suoi client
    - un servizio è *stateful* se gestisce lo stato della conversazione con i suoi client – l'effetto dell'esecuzione di un'operazione può dipendere dalla storia della conversazione con il particolare client
    - discuteremo di servizi stateful con riferimento a TCP



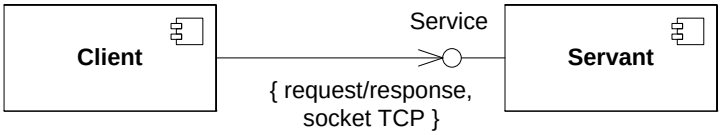
UDP – Discussione

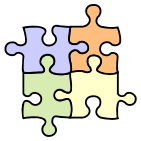
- In conclusione, possibili usi di UDP
  - per ridurre l'overhead della comunicazione
  - se è accettabile una comunicazione non affidabile
  - se posso permettermi di gestire esplicitamente l'affidabilità
  - soprattutto per server stateless



* Un'applicazione client-server TCP

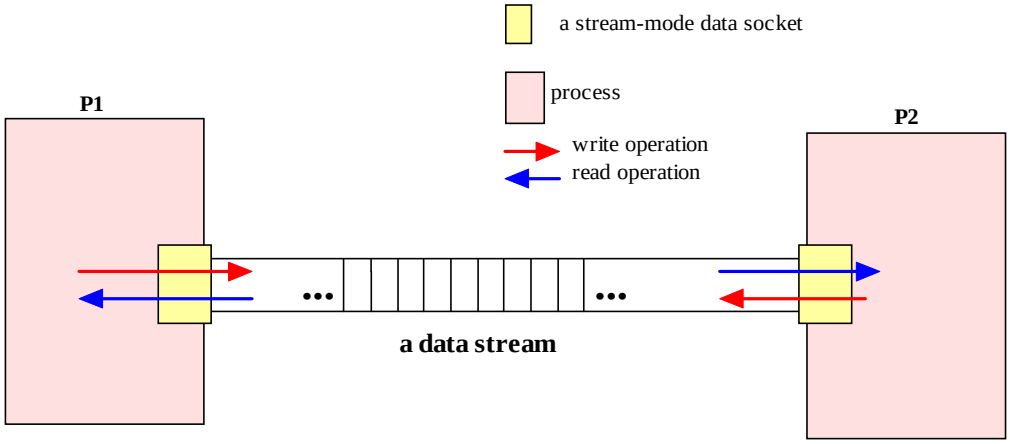
- Viene ora mostrata la realizzazione di un'applicazione client-server – sempre basata su un protocollo-richiesta-risposta
  - basata su TCP – i messaggi vengono scambiati su un canale di comunicazione bidirezionale
  - realizzata come server concorrente – multithreaded
    - intuitivamente, ogni volta che viene ricevuta una richiesta, il proxy lato server crea un nuovo thread per gestire la richiesta
  - scopo è notare come i diversi interventi siano localizzati nel codice del "connettore"

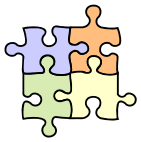




- API di Java per socket TCP

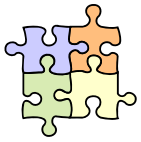
- TCP consente la trasmissione di flussi di dati (bidirezionali) tra due processi





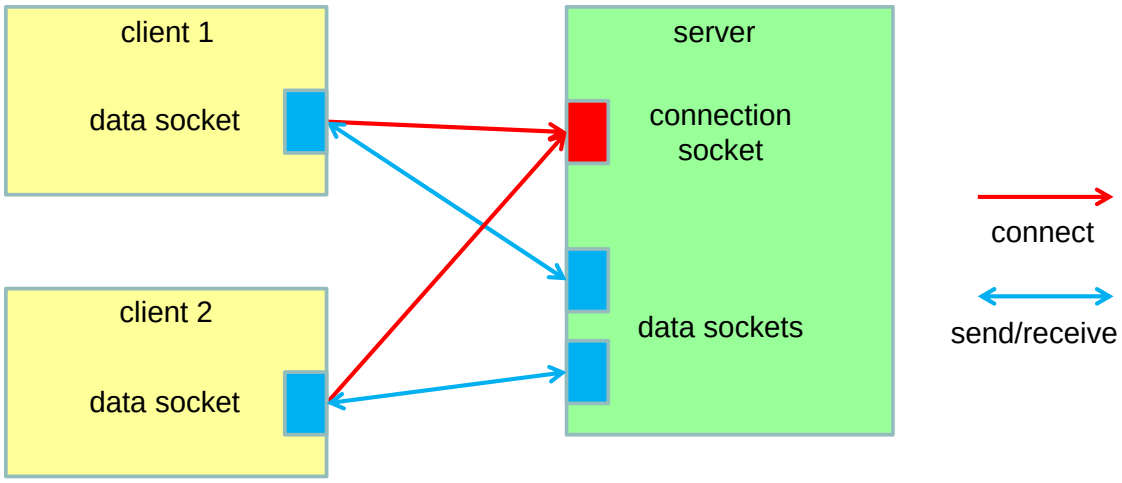
API di Java per socket TCP

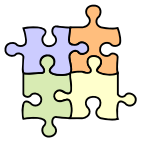
- Le API assumono che, nel momento in cui una coppia di processi devono stabilire una connessione
  - uno abbia il ruolo di *client* – fa una richiesta di ***connect***
  - l'altro quello di *server* – a fronte di una richiesta di connect risponde con una ***accept*** – bloccante
- Da quel momento in 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



API di Java per socket TCP

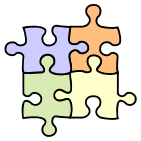
- Per il server, due tipi di socket
  - un tipo per accettare connessioni (condiviso)
  - un altro tipo per le operazioni send e receive (non condivisi)





API: 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



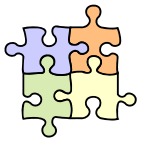
- Il servizio

- Le definizioni del ***Service*** (con le relative eccezioni) e del ***Servant*** sono inalterate

```
package asw.asw820.service;

/* Interfaccia del servizio Service. */
public interface Service {
    public String alpha(String arg) throws ServiceException, RemoteException;
    public String beta(String arg) throws ServiceException, RemoteException;
}
```

```
package asw.asw820.service;

/** ServiceException rappresenta un'eccezione "funzionale" legata al servizio. */
public class ServiceException extends Exception {
        public ServiceException(String message) {
                super(message);
        }
}
```

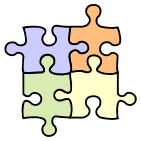
```
package asw.asw820.service;

/** RemoteException indica un problema nell'accesso remoto al servizio Service. */
public class RemoteException extends Exception {
        public RemoteException(String message) {
                super(message);
        }
}
```

Il servente

- Le definizioni del ***Service*** (con le relative eccezioni) e del ***Servant*** sono inalterate

```
package asw.asw820.server;

import asw.asw820.service.*;

/* Implementazione del servizio Service. */
public class Servant implements Service {
    public String alpha(String arg) throws ServiceException { ... fa qualcosa ... }
    public String beta(String arg) throws ServiceException { ... fa qualcosa ... }
}
```

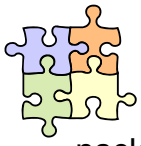


Un client del servizio

- Anche la definizione del ***Client*** può rimanere inalterata

```
package asw.asw820.client;

import asw.asw820.service.*;

/* client del servizio */
public class Client {

    private Service service;

    public Client(Service service) {  this.service = service;  }

    public void run(...) {
        try {
            ... service.alpha(...) ...
        } catch (ServiceException e) { ... gestisci e ... }
        } catch (RemoteException e) { ... gestisci e ... }
    }

}
```

```
package asw.asw820.client;

import asw.asw820.service.*;
import asw.asw820.client.connector.*;

/* Applicazione client: ottiene e avvia il client. */
public class Main {

        /* Crea e avvia un oggetto Client. */
        public static void main(String[] args) {
                Service service = ServiceFactory.getInstance().getService();
                /* crea il client e gli inietta il servizio da cui dipende */
                Client client = new Client(service);
                client.run();
        }

}
```

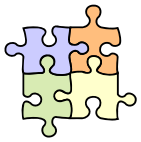



Factory per il proxy lato client

```
package asw.asw820.client.connector;

import asw.asw820.service.*;
import java.net.*;

/* Factory per il servizio Service (lato client). */
public class ServiceFactory {
    ... variabile e metodo per singleton ...

    /* Factory method per il servizio Service. */
    public Service getService() {
      Service service = null;
      try {
            InetAddress address = InetAddress.getByName("192.168.56.99");
            int port = 7896;
            service = new ServiceClientTCPProxy(address, port);
      } catch(Exception e) { e.printStackTrace(); }
      return service;
    }
}
```

Il proxy lato client (1)

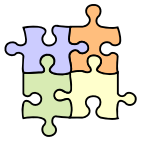
```
package asw.asw820.client.connector;

import asw.asw820.service.*;
import java.net.*;
import java.io.*;

/* Remote proxy lato client per il servizio Service. */
public class ServiceClientTCPProxy implements Service {
    private InetAddress address;    // indirizzo del server
    private int port;                // porta per il servizio

    public ServiceClientProxy(InetAddress address, int port) {
        this.address = address;    this.port = port;
    }
    public String alpha(String arg) throws ServiceException, RemoteException {
        return doOperation("alpha", arg);
    }
    public String beta(String arg) throws ServiceException, RemoteException {
        return doOperation("beta", arg);
    }
}
```

Il proxy lato client (2)

- Il metodo ***doOperation*** cambia in modo significativo

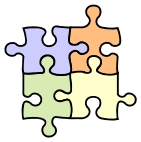
```
/* metodo di supporto che gestisce la richiesta remota di un'operazione */
public String doOperation(String op, String arg)
        throws ServiceException, RemoteException {
    String result = null;
    Socket socket = null;
    try {
        ... chiede una connessione al server ...
        ... prepara la richiesta ...
        ... invia la richiesta di servizio e i relativi parametri ...
        ... riceve la risposta ...
        ... estrae il risultato ...
    } catch(Exception e) {
        prova a gestire l'eccezione, oppure, più semplicemente:
        throw new RemoteException("Client Proxy: " + e.getMessage());
    }
    return result;
}
```



Il proxy lato client (3)

- Il metodo ***doOperation*** cambia in modo significativo

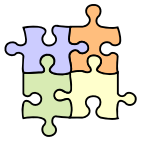
```
/* chiede una connessione al server */
socket = new Socket(address, port);    // bloccante
/* i due canali di comunicazione con il server */
DataInputStream in =
    new DataInputStream(socket.getInputStream());
DataOutputStream out =
    new DataOutputStream(socket.getOutputStream());

/* codifica la richiesta di servizio e i relativi parametri */
 /* la richiesta ha la forma "operazione$parametro" */
String request = op + "$" + arg;
/* invia la richiesta */
out.writeUTF(request);    // non bloccante

/* riceve la risposta */
String reply = in.readUTF();    // bloccante
```

Il proxy lato client (4)

- Il metodo ***doOperation*** cambia in modo significativo

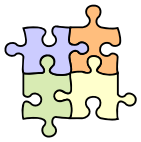
```
/* la risposta può avere le seguenti forme:
 * "#risultato"
 * "@messaggio per eccezione" */
if ( reply.startsWith("#") ) {
        /* è un risultato */
        result = reply.substring(1);
} else if ( reply.startsWith("@") ) {
        /* si è verificata una ServiceException */
        String message = reply.substring(1);
        throw new ServiceException(message);
} else {
        /* risposta malformata, solleva una RemoteException */
        throw new RemoteException("Malformed reply: " + reply);
}

return result;
```



Il server

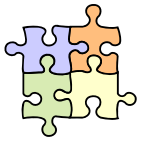
- L'oggetto ***Server*** – in esecuzione su 192.168.56.99 – è responsabile di
  - creare il ***Servant***
  - creare e avviare il proxy lato server

```
package asw.asw820.server.connector;

import asw.asw820.service.Service;
import asw.asw820.server.Servant;

/* server per il servizio */
public class Server {
    public static void main(String[] args) {
        Service service = new Servant();
        int port = 7896;
        ServiceServerTCPProxy server =
            new ServiceServerTCPProxy(service, port);
        server.run();
    }
}
```

Il proxy lato server (1)

```
package asw.asw820.server.connector;

import asw.asw820.service.Service;

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

/* Remote proxy lato server per il servizio Service. */
public class ServiceServerTCPProxy {
    private Service service;          // il vero servizio
    private int port;                 // porta per il servizio

    public ServiceServerTCPProxy(Service service, int port) {
        this.service = service;    this.port = port;
    }

    public void run() {  ... segue ...  }
}
```



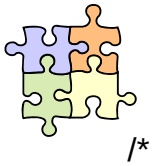
Il proxy lato server (2)

- Il metodo ***run*** del "remote proxy" lato server
  - questa volta un server concorrente

```
public void run() {
   try {
      /* crea il server socket su cui ascoltare/ricevere richieste */
      ServerSocket listenSocket = new ServerSocket(port);
      while (true) {
         /* aspetta/accetta una richiesta
          * quando arriva una richiesta, crea il relativo socket */
         Socket clientSocket = listenSocket.accept();    // bloccante
         /* la richiesta sarà gestita in un nuovo thread, separato */
         ServantThread thread = new ServantThread(clientSocket, service);
      }
   } catch (Exception e) { ... gestisci eccezione ... }
}
```

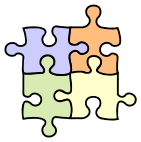

Gestione della richiesta (1)

```
package asw.asw820.server.connector;

import asw.asw820.service.*;
import java.net.*;    // per le socket
import java.io.*;     // per i flussi di I/O

public class ServantThread extends Thread {
    private Service service;
    private Socket clientSocket;
    private DataInputStream in;
    private DataOutputStream out;

    public ServantThread(Socket clientSocket, Service service) {
        try {
            this.clientSocket = clientSocket; this.service = service;
            in = new DataInputStream(clientSocket.getInputStream());
            out = new DataOutputStream(clientSocket.getOutputStream());
            this.start();    // esegue run() in un nuovo thread
        } catch (IOException e) { ... gestisci eccezione ... }
    }
    ... segue ...
}
```



Gestione della richiesta (2)

```
/* run eseguito in un nuovo thread */
public void run() {
    try {
        /* riceve una richiesta */
        String request = in.readUTF();  // bloccante

        /* estrae operazione e parametro */
        String op = ... come prima ...;
        String param = ... come prima ...;

        ... chiedi l'erogazione del servizio,
            ottieni il risultato, genera la risposta reply ...

        /* invia la risposta */
        out.writeUTF(reply);    // non bloccante
    } catch (Exception e) { ... gestisci eccezione ... }
}
```

Gestione della richiesta (3)

- Questa parte della gestione della richiesta è come in precedenza

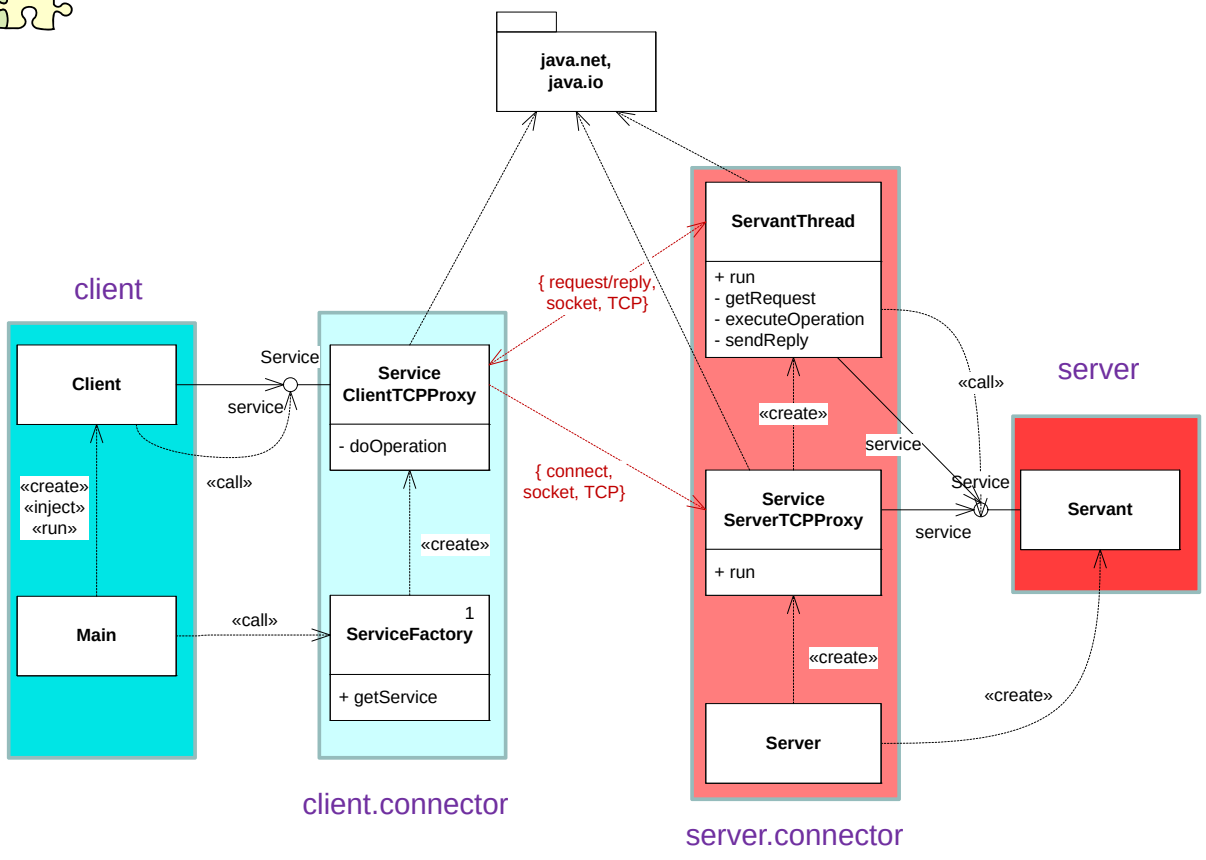
```
/* chiedi l'erogazione del servizio, ottieni il risultato,
 * genera la risposta reply */
/* la risposta può avere le seguenti forme:
 * "#risultato" oppure "@messaggio per eccezione" */
String reply = null;
try {
    String result = this.executeOperation(op, arg);
    /* operazione completata, rispondi "#risultato" */
    reply = "#" + result;
} catch (ServiceException e) {
    /* operazione NON completata, rispondi "@messaggio" */
    reply = "@" + e.getMessage();
} catch (RemoteException e) {
    /* no, il servente non dovrebbe mai sollevare RemoteException */
    reply = "@" + e.getMessage();
}
```

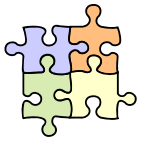
Gestione della richiesta (4)

- Questa parte della gestione della richiesta è come in precedenza

```
private String executeOperation(String op, String arg)
        throws ServiceException, RemoteException {
    String result = null;

    if ( op.equals("alpha") ) {
        result = service.alpha(arg);
    } else if ( op.equals("beta") ) {
        result = service.beta(arg);
    } else {
        throw new RemoteException("Operation " + op + " is not supported");
    }

    return result;
}
```

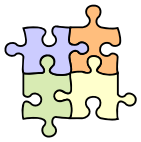

In sintesi





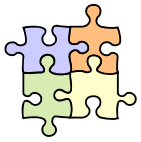
TCP – Discussione

- TCP offre garanzie di affidabilità migliori rispetto a UDP
  - ma, come discusso in precedenza, sono comunque garanzie di affidabilità limitate
- In generale, non ipotizzare mai che la comunicazione distribuita abbia la stessa semantica della comunicazione locale – ovvero di quella che avviene all'interno di un singolo processo
  - piuttosto, "è necessaria una buona comprensione del paradigma di comunicazione implementato da ciascun middleware, della sua struttura e dei suoi principi di funzionamento"



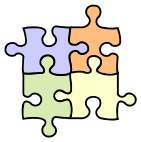
TCP – Discussione

- Gestione delle connessioni
  - nella versione mostrata
    - il client ottiene e richiede una connessione al server per ciascuna singola richiesta di servizio
    - in corrispondenza, il server crea e alloca un thread per gestire ciascuna singola richiesta



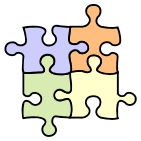
TCP – Discussione

- Gestione delle connessioni
  - poiché il costo (temporale) dell'instaurazione di una connessione e della creazione di un thread può essere significativo, in pratica è bene usare altri approcci
    - ad es., una connessione (e un corrispondente thread) può essere usata per gestire un gruppo di richieste (temporalmente contigue) da parte di un client – nell'ambito di una conversazione o sessione tra client e server
      - in pratica, il metodo ***run()*** del ***ServantThread*** può gestire una sequenza di richieste mediante un'istruzione ripetitiva
    - ad es., può essere usato un pool di connessioni
    - queste scelte hanno impatto su prestazioni e scalabilità



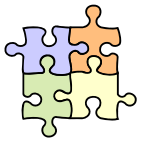
TCP – Discussione

- Con TCP è relativamente semplice realizzare anche servizi stateful – ovvero, che gestiscono lo stato delle conversazioni (*sessioni*) con i loro client – in particolare
  - ciascun servant thread (ovvero, ciascuna istanza di servant thread) può essere dedicato alla gestione di tutta la conversazione con un particolare client
    - inoltre, a ciascun servant thread può essere assegnata anche la responsabilità di gestire lo stato della conversazione (sessione) con quel particolare client
  - il server proxy è invece condiviso da tutti i client, e viene usato per iniziare nuove conversazioni
    - inoltre, al server proxy può essere assegnata anche la responsabilità di gestire lo stato dell'applicazione, condiviso da tutti i client dell'applicazione
  - nel protocollo, potrebbero essere utili operazioni per iniziare e concludere una sessione



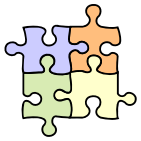
- Socket – discussione

- 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



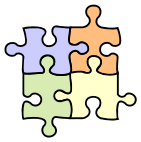
* Messaggi da scambiare

- I 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?



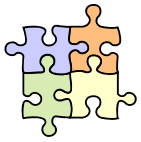
Chiamata di procedute remote

- Quello che abbiamo fatto finora rappresenta un caso comune (ma non è l'unico possibile) – la *chiamata di procedure remote* (*remote procedure call*, o *RPC*) – anche detta *invocazione di operazioni/metodi remoti*
  - un server espone, mediante un opportuno protocollo, un insieme di operazioni/procedure/metodi 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

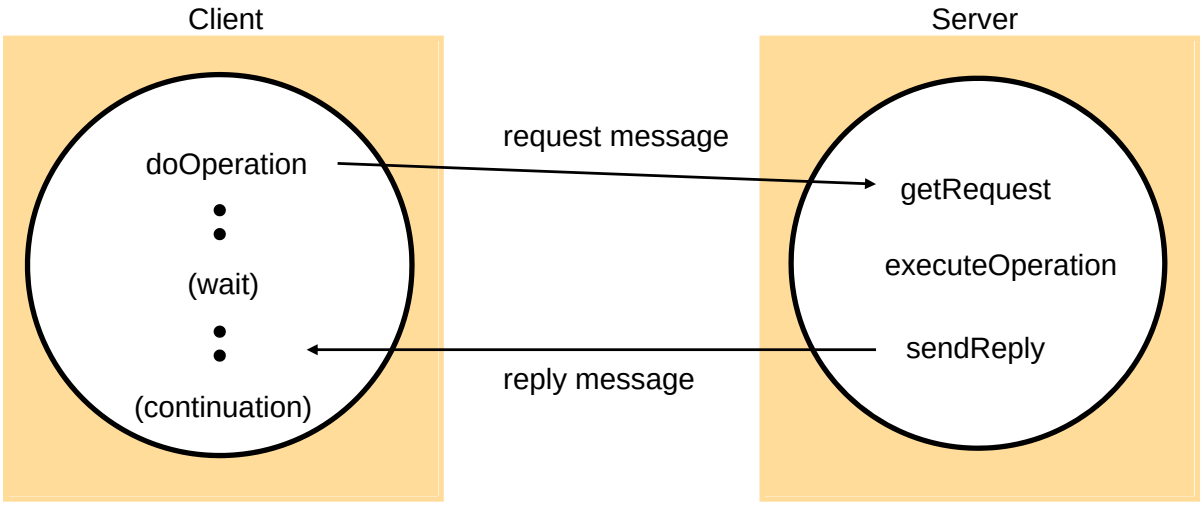


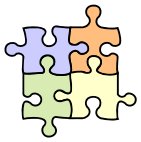
Messaggi per chiamate di procedure remote

- Tipi di messaggi coinvolti nella chiamata di procedure remote
  - *richiesta*
    - codifica l'operazione richiesta, nonché i parametri attuali
  - *risposta*
    - codifica i risultati restituiti – potrebbero essere anche più di uno
    - può codificare anche un'eccezione sollevata durante l'esecuzione dell'operazione richiesta



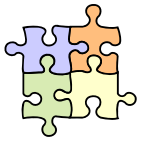
Protocollo richiesta-risposta



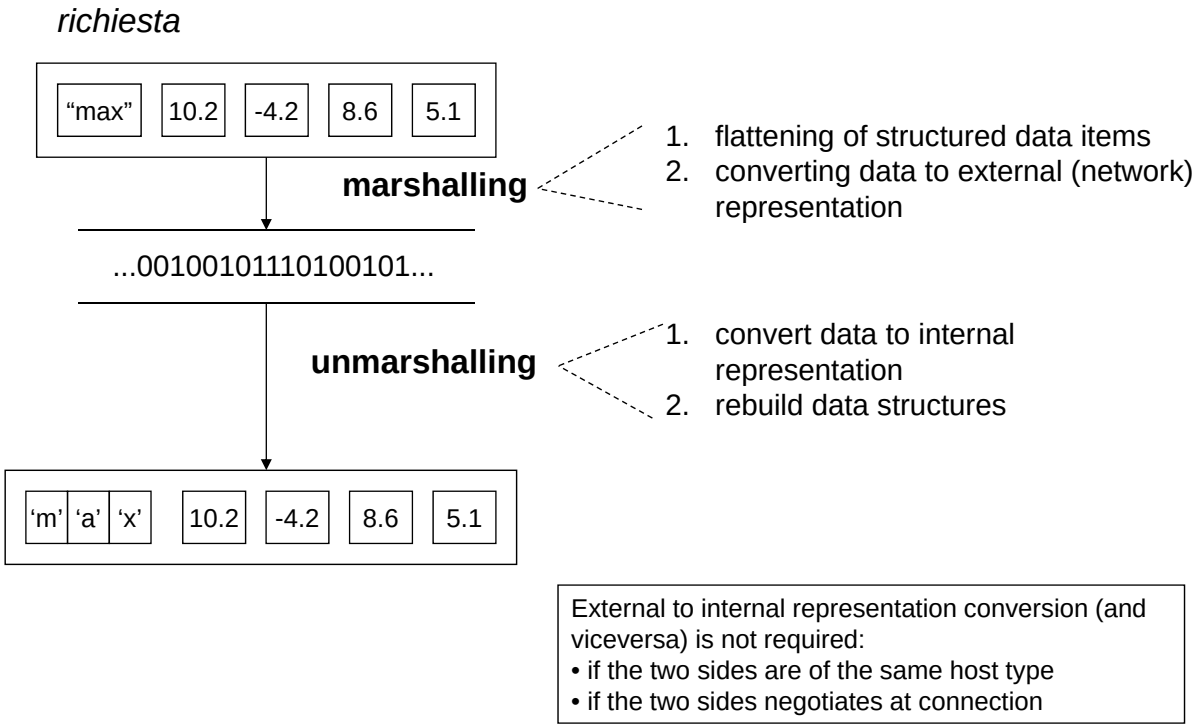


Marshalling e unmarshalling

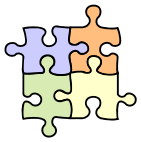
- Le due parti devono essere d'accordo sul formato (sintassi e semantica) dei messaggi scambiati
  - trascuriamo qui dettagli di basso livello (che vanno comunque considerati)
    - ad es., le rappresentazioni "little endian" e "big endian"
- Le due parti devono inoltre svolgere attività di
  - *marshalling*
    - 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



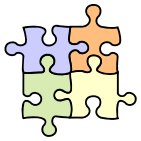
Comunicazione richiesta-risposta

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)

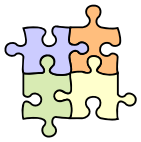
programma server

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



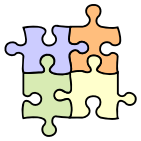
Esempio

- 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
- Nell'esempio precedente – relativo a un caso molto semplice
  - le richieste sono stringhe nella forma operazione$parametro
  - le risposte relative a risultati sono stringhe nella forma #risultato
  - le risposte relative ad eccezioni sono stringhe nella forma @messaggio
- Nella pratica vengono invece preferite delle codifiche binarie – generate da opportuni "compilatori di interfacce"



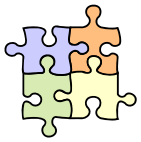
Marshalling di dati e risultati

- 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
  - nella forma di oggetti serializzati
    - linguaggi come Java consentono di serializzare un oggetto/un grafo di oggetti collegati come una sequenza binaria – e di trasmetterli e deserializzarli
    - JSON



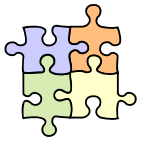
Altri tipi di comunicazione

- Non esiste solo la comunicazione richiesta-risposta – alcuni altri casi di comunicazione
  - un componente ***Client*** invia una richiesta a un componente ***Server***
    - ma non rimane in attesa della risposta
  - un componente ***Server*** invia una risposta – relativa a un richiesta ricevuta in precedenza – a un componente ***Client***
  - un componente ***Provider*** vuole inviare dei dati a un componente ***Consumer***
    - non ha bisogno di una risposta
  - un componente ***Publisher*** vuole inviare delle notifiche di eventi a diversi componenti ***Subscriber***
- Ciascuno di questi casi richiede di stabilire un opportuno protocollo di comunicazione e formato per i messaggi scambiati



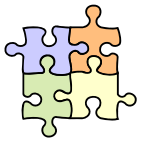
* Discussione

- I socket consentono la comunicazione tra processi
  - ma a un livello di astrazione basso (troppo basso?)
  - bisogna implementare (quasi) tutti gli aspetti della comunicazione
- Molti aspetti non sono stati discussi – talvolta nemmeno accennati
  - come gestire l'erogazione di servizi stateful?
  - come gestire la sicurezza?
  - come gestire/mascherare eventuali fallimenti della comunicazione?
  - la creazione di thread è un'operazione costosa – dal punto di vista temporale – come posso ridurre questo costo nel momento in cui un nuovo client effettua una richiesta?
  - quale semantica per il legame dei parametri?
  - ...



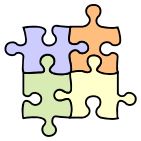
Discussione

- Gli strumenti di middleware sono stati realizzati per semplificare lo sviluppo di applicazioni distribuite
  - per semplificare la comunicazione tra processi
  - per offrire diversi paradigmi di interazione 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
- L'uso corretto di uno strumento di middleware richiede una comprensione della sua particolare "semantica"
  - ad es., come viene gestita l'affidabilità?



- Alcuni 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



Alcuni 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?
- Attenzione, ci sono più modalità di realizzazione per queste applicazioni!