Luca Cabibbo

Architettura dei Sistemi Software

Connettori distribuiti e comunicazione client-server

dispensa asw815

marzo 2021

When writing programs that communicate across a computer network, one must first invent a protocol, an agreement on how those programs will communicate.

W.R. Stevens, B. Fenner, A.M. Rudoff

- Riferimenti

- Coulouris, G, Dollimore, J., Kindberg, T., and Blair, G. **Distributed Systems**: Concepts and Design, fifth edition. Pearson, 2012.
  - Chapter 4, Interprocess Communication
- Liu, M.L. **Distributed Computing**: Principles and Applications. Pearson, 2004.
  - Chapter 4, The Socket API
  - Chapter 5, The Client-Server Paradigm
- Tanenbaum, A.S. and Bos, H. **Modern Operating Systems**, fourth edition. Pearson, 2015.
  - Chapter 2, Processes and Threads

- Obiettivi e argomenti

- Obiettivi
  - discutere alcuni problemi e soluzioni comuni nella realizzazione dei connettori nei sistemi distribuiti
  - esemplificare l'applicazione dei socket nella realizzazione di semplici connettori distribuiti per applicazioni di tipo client-server
- Argomenti
  - introduzione ai connettori distribuiti e alla comunicazione client-server
  - un'applicazione client-server UDP
  - un'applicazione client-server TCP
  - messaggi e protocolli
  - discussione

* Introduzione ai connettori distribuiti e alla comunicazione client-server

- Lo sviluppo dei sistemi software distribuiti richiede la realizzazione di connettori in grado di far interagire processi diversi, in esecuzione su nodi differenti o anche su uno stesso nodo
  - questa dispensa ha lo scopo di discutere alcuni problemi e soluzioni comuni nella realizzazione dei connettori nei sistemi distribuiti
    - viene mostrata la realizzazione di alcuni connettori basati sui meccanismi di base di comunicazione distribuita offerti dai sistemi operativi
  - in pratica, i connettori distribuiti vengono normalmente realizzati utilizzando il middleware
    - in questo caso, però, è comunque utile avere delle intuizioni sui problemi e le soluzioni che è stato necessario affrontare nella realizzazione del middleware sottostante

Comunicazione interprocesso e socket

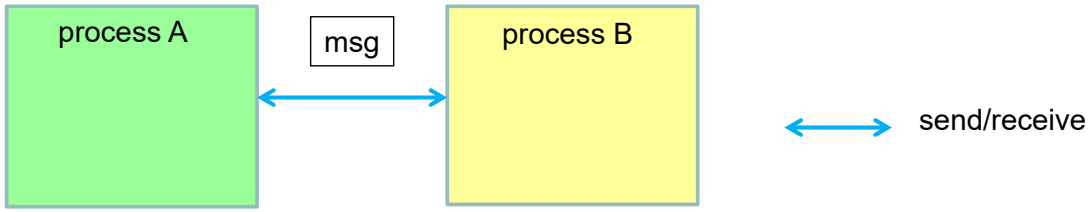
- I sistemi operativi offrono uno o più servizi di base per la *comunicazione interprocesso* (*IPC*, *interprocess communication*) – che sono necessari (direttamente o indirettamente) nella realizzazione dei connettori nei sistemi software distribuiti
  - ad es., pipe e socket
  - alcuni servizi di IPC sono basati su protocolli di Internet – ad es., socket TCP e UDP
  - questa dispensa esemplifica i socket come servizio di IPC per la realizzazione di semplici connettori nella realizzazione di applicazioni distribuite di tipo client-server

Dai socket al middleware

- Questi stessi servizi di IPC di base sono utilizzati nella realizzazione dei servizi di middleware – che offrono astrazioni di comunicazione più sofisticate e più efficaci nella realizzazione dei sistemi distribuiti
  - lo scopo ultimo di questa dispensa non è di discutere come utilizzare i socket per realizzare dei connettori per sistemi distribuiti
  - piuttosto, è di discutere alcuni problemi affrontati dagli strumenti di middleware e fornire delle intuizioni sulle soluzioni che essi offrono

Comunicazione interprocesso di base

- In questa dispensa consideriamo solo la comunicazione interprocesso di tipo unicast (ma ignoriamo quella di tipo multicast)



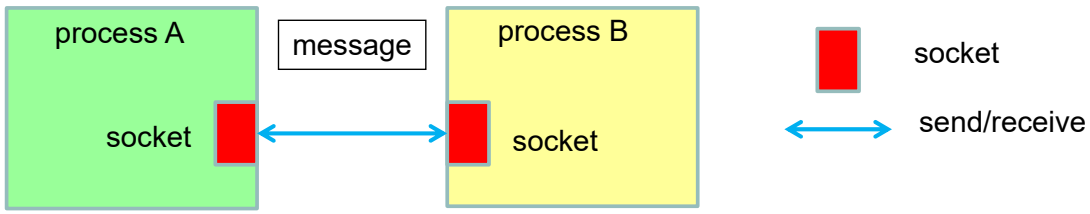
  - una coppia di processi indipendenti – che possono essere in esecuzione su uno stesso computer o su computer diversi, collegati da una rete di interconnessione – si scambiano dati (*messaggi*) usando le primitive **send** e **receive**
  - in pratica, la comunicazione va basata su un qualche *protocollo* (un protocollo applicativo) – formato da messaggi e da regole, anche di sincronizzazione, e accettato da entrambe le parti

Primitive send e receive

- 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 gruppo di dati o *messaggio* da trasmettere – a questo livello di astrazione, per messaggio si intende semplicemente una qualche sequenza di dati, binaria o testuale

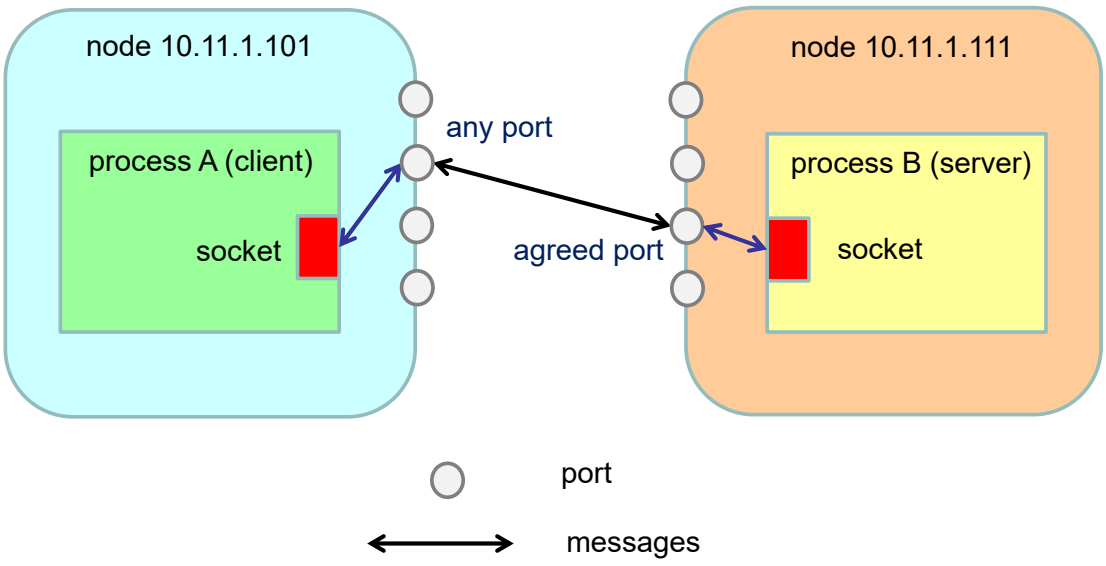
Socket

- I **socket** sono un'astrazione di programmazione (ovvero, un'API standard) per lo scambio (trasmissione e ricezione) di messaggi attraverso una rete oppure come meccanismo di IPC
  - 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 viene legato (a runtime) 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 Internet (a livello di trasporto) UDP e TCP
- **UDP** è un protocollo per il trasporto di datagrammi
  - di tipo “best effort”, senza ack e né ritrasmissione – dunque non offre garanzie di consegna – i datagrammi potrebbero venire persi (a causa di errori di rete o scartati se la rete è congestionata) o consegnati fuori ordine
  - connectionless – non è richiesto setup (l’overhead è basso)
- **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 – è richiesto il setup di un canale di comunicazione bidirezionale (l’overhead è più alto)

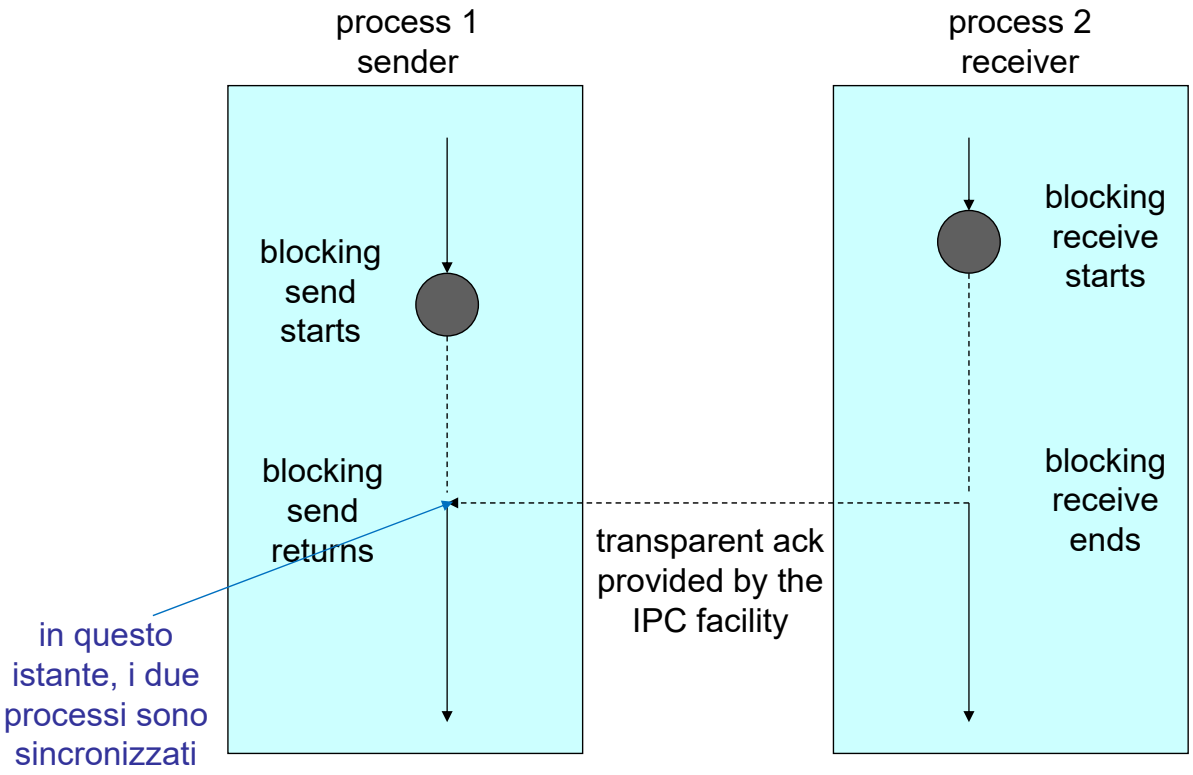
Primitive connect e disconnect

- Due ulteriori primitive per la comunicazione connection-oriented
  - **connect** – per stabilire una connessione logica tra due processi
    - *request-to-connect* + *accept-connection*
    - alloca le risorse per gestire quella connessione
  - **disconnect** – per terminare una connessione logica su entrambi i lati della comunicazione
    - dealloca le risorse per la gestione di quella 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

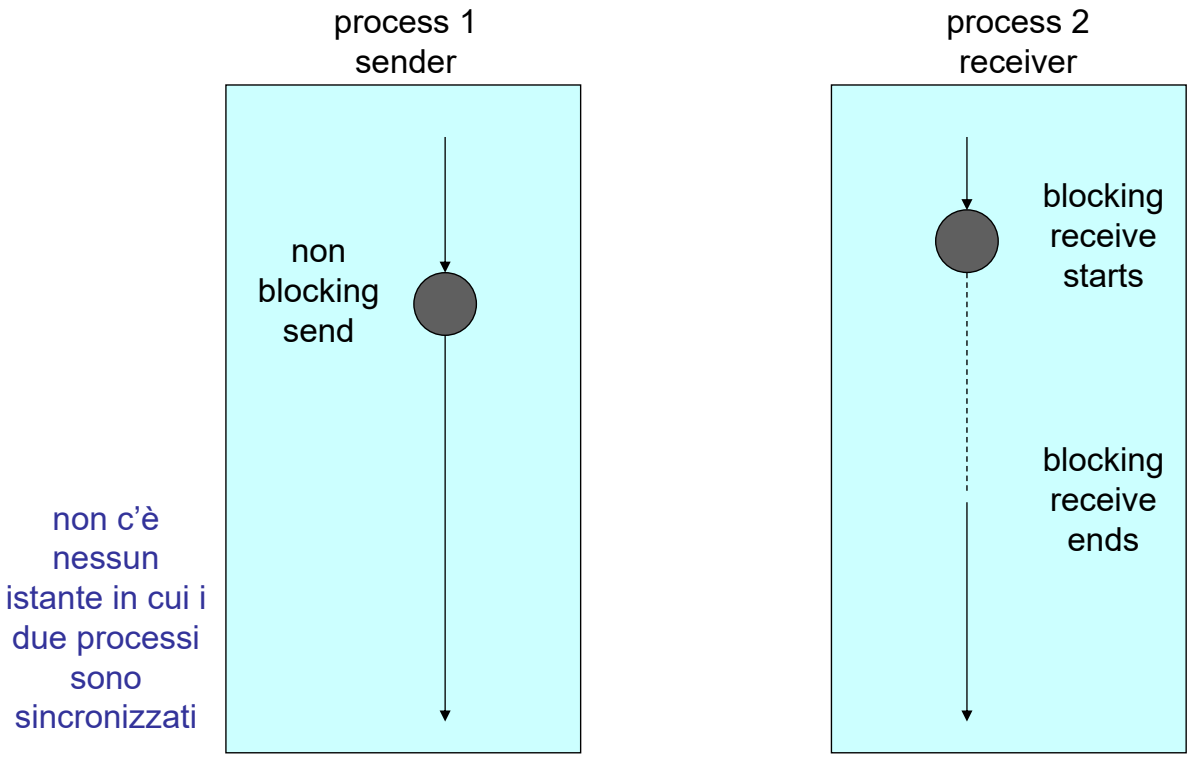
- Sincronizzazione

- 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**
    - 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 abbastanza diverso da questo

Comunicazione sincrona



Comunicazione asincrona



Sincronizzazione

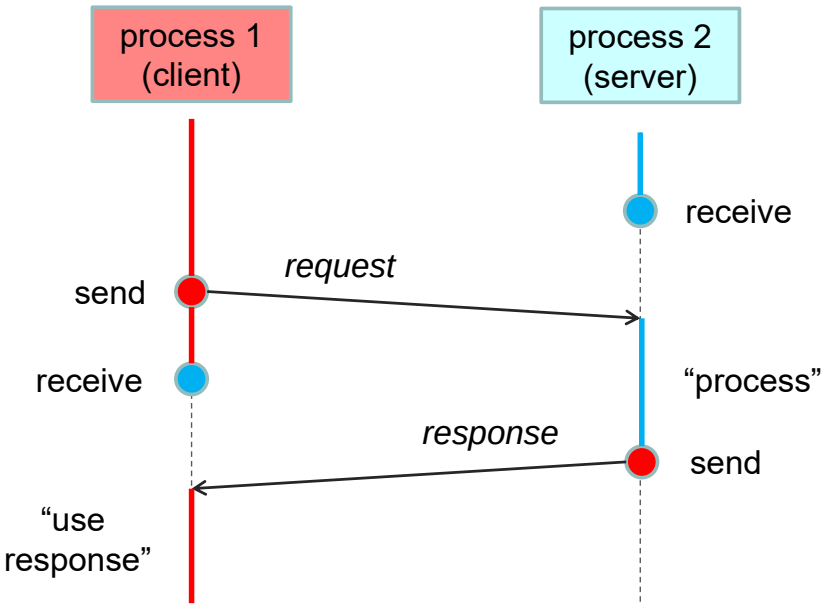
- Processi che operano in modo concorrente e che comunicano devono spesso provvedere a una loro opportuna *sincronizzazione*
  - ad es., un processo client che fa una richiesta a un processo server vuole normalmente sapere se la sua richiesta è stata elaborata o meno
    - che fare se la richiesta viene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

- Comunicazione client-server

- Una modalità di comunicazione comune nei sistemi distribuiti è quella di tipo client-server – basata su un protocollo richiesta-risposta
  - nello stile *client-server*, la comunicazione è iniziata da un processo *client* che interagisce con un processo *server* per chiedergli l'esecuzione di un'operazione o *servizio*
  - in un protocollo *richiesta-risposta*
    - il client fa una *richiesta* al server
    - il server elabora la richiesta e fornisce una *risposta* al client

Protocollo richiesta-risposta

- Comunicazione client-server basata su un protocollo richiesta-risposta
  - viene anche realizzata una sincronizzazione *parziale*



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

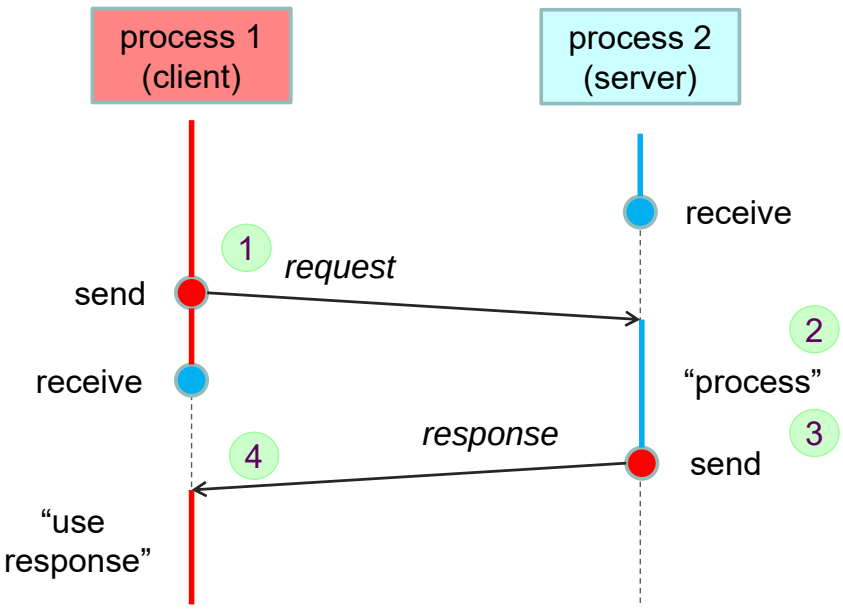
la sincronizzazione è parziale: il server non sa se il client ha ricevuto la sua risposta

- Affidabilità

- Un problema comune nei sistemi distribuiti è che si possono verificare dei *guasti* nella comunicazione
  - i guasti possono essere
    - guasti nei canali di comunicazione – guasti hardware
    - guasti (crash) nei processi che comunicano – guasti software
  - 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 – per esempio
  - 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'operazione 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 – operazione 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 non sempre sono richieste tutte 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 nemmeno TCP si rivela completamente affidabile [Coulouris]
  - se la perdita di pacchetti è eccessiva, oppure la rete è partizionata oppure è molto congestionata, allora TCP potrebbe dare un ack quando un pacchetto è stato consegnato all'OS destinatario – ma il pacchetto potrebbe non venire mai ricevuto dall'applicazione destinataria
  - i processi comunicanti non riescono sempre a sapere se i messaggi che hanno inviato sono stati consegnati o meno – e non sempre riescono a distinguere tra guasto della rete e guasto dell'altro processo remoto

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 ven
- Tuttavia, compl
  - se par potrebbe da nato all'OS destinata nire mai ricevuto dall'applicazione destin
  - i processi comunicanti non riescono sempre a sapere se i messaggi che hanno inviato sono stati consegnati o meno – e non sempre riescono a distinguere tra guasto della rete e guasto dell'altro processo remoto

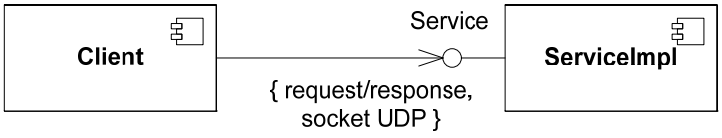
Dunque, valuta con attenzione quanto viene detto – e poniti dei dubbi sull'apparenza delle cose e su quanto non viene detto…

Affidabilità

- Un servizio di comunicazione – in particolare, un servizio di middleware
  - può tollerare e mascherare alcuni possibili guasti – e realizzare un canale di comunicazione che garantisce un certo livello di affidabilità
  - ma probabilmente non può mai realizzare 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 e discussa una semplice applicazione di tipo client-server, basata su un protocollo richiesta/risposta, realizzata con socket UDP
  - la struttura dell'applicazione è simile (ma un po' più complessa) a quella mostrata nel contesto dell'"Introduzione ai connettori"
    - un servizio **Service**
    - un elemento **ServiceImpl** in grado di erogare il servizio **Service**
    - un elemento **Client** che richiede l'erogazione del servizio **Service**

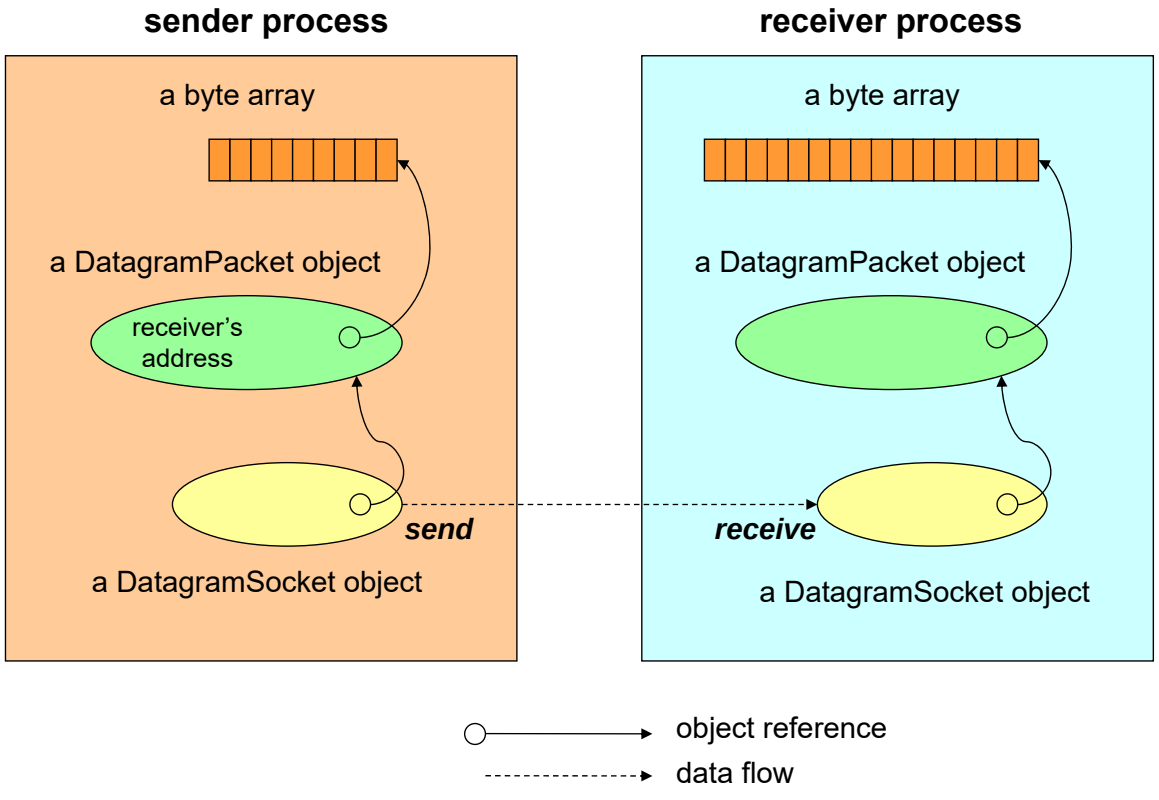


  - ci occupiamo della gestione dei guasti in modo semplificato

- 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

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

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

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 al socket specificato (address rappresenta un indirizzo IP)
- DatagramSocket()
  - crea un DatagramSocket legato a 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 servizio

```
package asw.socket.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 – il servizio lo segnala sollevando un'eccezione
  - 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 per il servizio Service

```
package asw.socket.service;

/** ServiceException rappresenta un'eccezione "funzionale"
 *  legata al servizio. */
public class ServiceException extends Exception {

   public ServiceException(String message) {
      super(message);
   }

}
```

```
package asw.socket.service;

/** RemoteException indica un problema nell'accesso remoto al servizio. */
public class RemoteException extends Exception {

   public RemoteException(String message) {
      super(message);
   }

}
```


- L'implementazione del servizio

- Ecco la definizione di **ServiceImpl**, che completa gli aspetti funzionali del servizio

```
package asw.socket.service.impl;

import asw.socket.service.*;

/* Implementazione del servizio Service. */
public class ServiceImpl implements Service {

   public String alpha(String arg) throws ServiceException {
      ... fa qualcosa ...
   }

   public String beta(String arg) throws ServiceException {
      ... fa qualcosa ...
   }

}
```

  - si noti che l'implementazione del servizio non solleva l'eccezione remota **RemoteException**
  - questa eccezione potrebbe essere invece sollevata da un remote proxy, se rileva un problema nella comunicazione

- Protocollo

- È prima necessario definire il protocollo applicativo che intendiamo utilizzare per gestire la comunicazione distribuita
  - utilizziamo un *protocollo richiesta-risposta*
  - per i messaggi di *richiesta* utilizziamo un formato di questo tipo
    - `operazione$argomento` – ad es., `alpha$abc`
  - per i messaggi di *risposta* utilizziamo i seguenti formati
    - `#risultato` se l'operazione termina normalmente
      - ad es., `#ABC`
    - `@messaggio` se l'operazione termina sollevando un'eccezione
      - ad es., `@Argument is null`
    - `!messaggio` se si è verificato un problema di comunicazione
      - ad es., `!Operation not supported`
  - inoltre, assumiamo che il server sia su **10.11.1.111** e ascolti sulla porta **6789**

- Un client del servizio

```
package asw.socket.client;

import asw.socket.service.*;

/* client del servizio */
public class Client {

   private Service service;

   public Client() { }

   public void setService(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.socket.main;

import asw.socket.client.Client;
import asw.socket.context.ApplicationContext;

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

   public static void main(String[] args) {
      Client client = ApplicationContext.getInstance().getClient();
      client.run();
   }

}
```


Application context lato client (1)

```
package asw.socket.client.connector;

import ...

public class ApplicationContext {

    ... parti per singleton ...

   /* Factory method per il client Client. */
   public Client getClient() {
      Client client = new Client();
      client.setService( this.getService() );
      return client;
   }

   ... segue ...
```

Application context lato client (2)

```
   ... segue ...

   /* Factory method per il servizio Service. */
   public Service getService() {
      Service proxy = null;
      try {
         InetAddress address = InetAddress.getByName("10.11.1.111");
         int port = 6789;
         proxy = new ServiceClientUDPProxy(address, port);
      } catch(Exception e) { e.printStackTrace(); }
      return proxy;
   }

}
```


Il proxy lato client (1)

```
package asw.socket.client.connector;

import asw.client.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) ... vedi dopo ...

   public String beta(String arg) ... vedi dopo ...

}
```

Il proxy lato client (2)

- I metodi **alpha** e **beta** del “remote proxy” lato client
  - sono i metodi di servizio invocati dal client – che però pensa di parlare direttamente con il servizio
  - ipotesi (semplificativa): l’interfaccia del servizio definisce più operazioni – ma, per semplicità, tutte queste operazioni hanno un solo parametro (una stringa) e restituiscono un solo valore (una stringa)

```
/* il metodo alpha del proxy */
public String alpha(String arg)
      throws ServiceException, RemoteException {
   return doOperation("alpha", arg);
}

/* il metodo beta del proxy */
public String beta(String arg)
      throws ServiceException, RemoteException {
   return doOperation("beta", arg);
}
```


Il proxy lato client (3)

- Il metodo **doOperation**

```
/* metodo di supporto per la comunicazione remota */
private String doOperation(String op, String arg)
      throws throws ServiceException, RemoteException {
   String result = null;
   try {
      ... crea il socket ...
      ... crea un datagramma di richiesta che codifica l’invocazione
          del servizio e i relativi parametri ...
      ... invia il datagramma di richiesta ...
      ... ricevi il datagramma di risposta ...
      ... estrai il risultato dal datagramma di risposta ...
      ... chiudi il socket ...
   } 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();
/* imposta un timeout (1 sec) */
socket.setSoTimeout(1000);
/* crea il datagramma di richiesta che codifica l'invocazione
 * del 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 (con timeout)
/* estrai la risposta dal datagramma di risposta */
String reply = new String( replyPacket.getData(),
                           replyPacket.getOffset(),
                           replyPacket.getLength() );
```

Si noti che, se si verifica un timeout nell'operazione receive (segnalato da un'eccezione), allora doOperation solleva un'eccezione remota

Il proxy lato client (6)

- Una possibile implementazione di **doOperation**

```
/* elabora la risposta, che può avere
 * le seguenti forme (vedi proxy lato server):
 * "#risultato" "@eccezione di servizio" "!eccezione remota" */
if ( reply.startsWith("#") ) {  // è un risultato
   result = reply.substring(1);
} else if ( reply.startsWith("@") ) {  // ServiceException
   String message = reply.substring(1);
   throw new ServiceException(message);
} else if ( reply.startsWith("!") ) {  // RemoteException
   String message = reply.substring(1);
   throw new RemoteException(message);
} else {  // risposta malformata
   throw new RemoteException("Malformed reply: " + reply);
}
/* chiudi il socket */
socket.close();
return result;
```


- Il server

- L'oggetto **Server** – eseguito su **10.11.1.111** e ascolta su **6789** – è responsabile di
  - creare un'istanza di **ServiceImpl**
  - creare e avviare il proxy lato server

```
package asw.socket.server.connector;

import asw.socket.service.Service;
import asw.socket.service.impl.ServiceImpl;

/* server per il servizio */
public class Server {

   public static void main(String[] args) {
      Service service = new ServiceImpl();
      int port = 6789;
      ServiceServerUDPProxy server =
         new ServiceServerUDPProxy(service, port);
      server.run();
   }

}
```

Il proxy lato server (1)

- Struttura del “remote proxy” lato server

```
package asw.socket.server.connector;

import asw.socket.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() {  ... vedi dopo ...  }

}
```


Il proxy lato server (2)

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

```
public void run() {
   try {
      /* crea il socket su cui ricevere le richieste */
      DatagramSocket socket = new DatagramSocket(this.port);
      /* sul server, disabilita il timeout */
      socket.setSoTimeout(0);
      byte[] buffer = new byte[1000];
      /* gestisci le richieste dei client */
      while (true) {
         getRequestAndSendReply(socket, buffer);
      }
      /* chiudi il socket */
      socket.close();
   } 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 ...
      ... invoca il 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 (senza timeout)

/* 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)

```
/* invoca il 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) {
   /* il servente non solleva MAI RemoteException,
    * ma si può arrivare qui se la richiesta è malformata */
   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
```

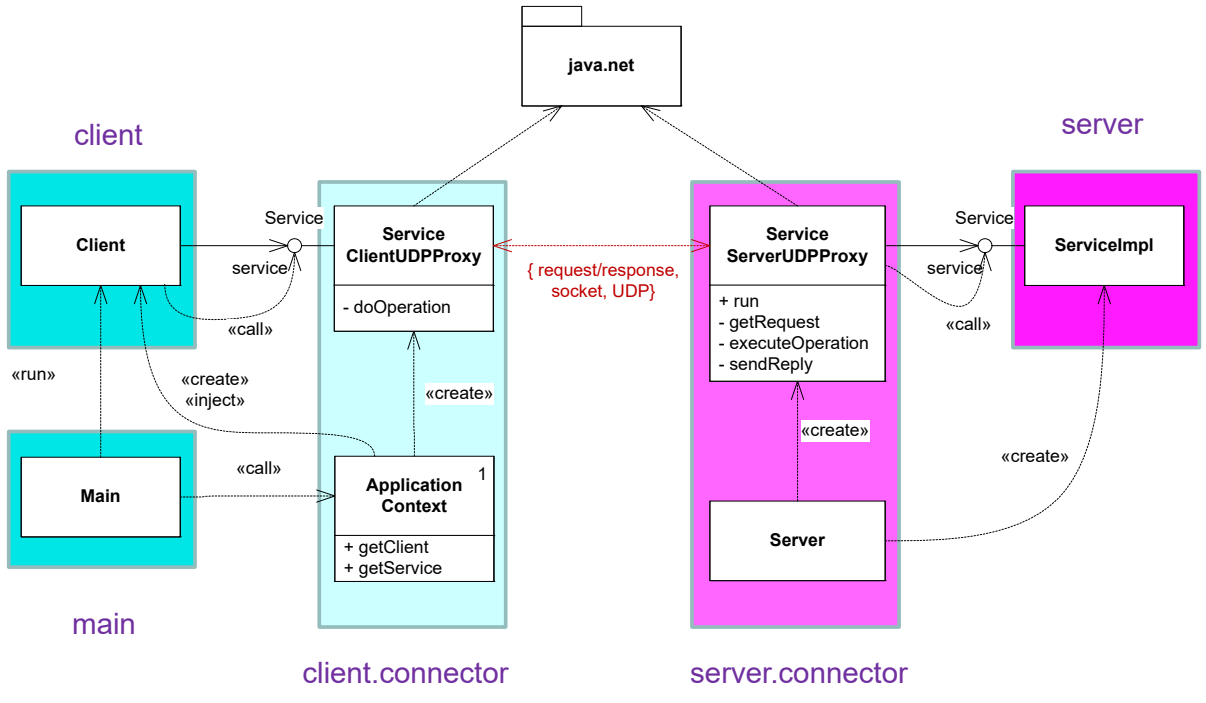
Il proxy lato server (7)

- 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 not supported");
   }
   return result;
}
```


- In sintesi

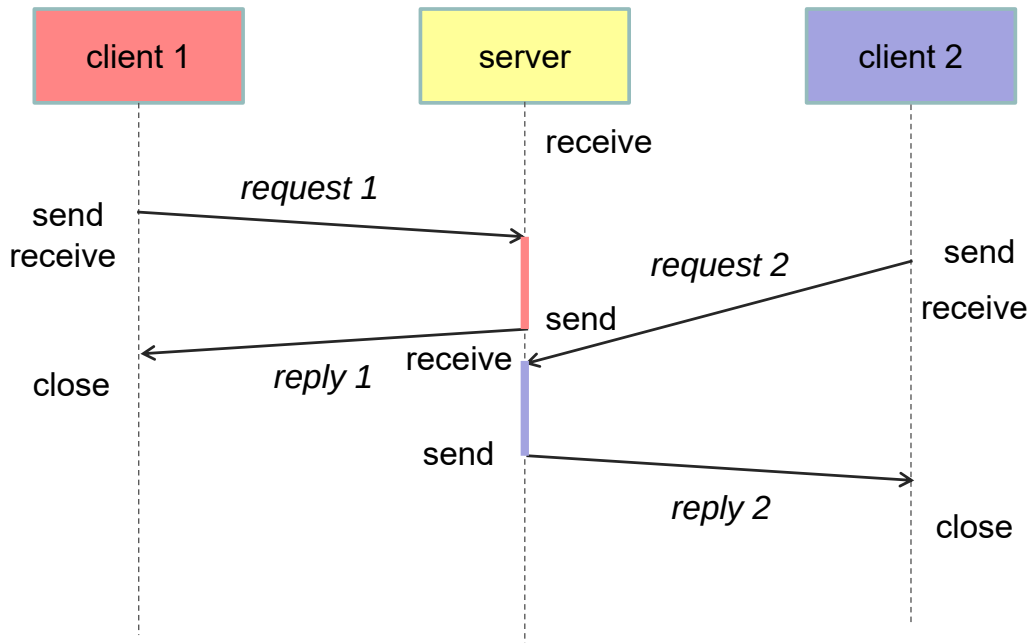


In sintesi

- Il client e il server interagiscono per *invocare un'operazione* che produce un *risultato* mediante un protocollo *richiesta-risposta*
  - il client vuole *invocare un'operazione* del server
    - per questo, il client invia al server un messaggio di *richiesta* che codifica l'*invocazione dell'operazione*
    - il client è sender del messaggio di *richiesta*
  - il server esegue l'operazione richiesta dal client
    - il server è receiver del messaggio di *richiesta*
    - il server decodifica il messaggio di *richiesta* per capire qual è l'*operazione invocata* dal client
    - il server esegue l'operazione invocata e calcola il *risultato*
    - il server codifica il *risultato* in un messaggio di *risposta* e lo invia al client
    - il server è sender del messaggio di *risposta*
  - il client riceve il risultato dell'operazione richiesta
    - il client è receiver del messaggio di *risposta*
    - il client estrae il *risultato* dal messaggio di *risposta*

In sintesi

- Esempio – due client (vengono gestito in modo sequenziale)



- 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 sequenziale e concorrente
    - nell'esempio, il server gestisce tutte le richieste nell'ambito di un solo thread, in modo *sequenziale* – il server potrà gestire una richiesta successiva solo dopo aver gestito la richiesta corrente
    - tuttavia, un server può anche essere *concorrente* (multithreaded) – in cui ciascuna richiesta viene gestita da un thread diverso – con UDP, tutti questi thread condividono il socket che utilizzano per comunicare
    - vedremo un esempio di server concorrente con riferimento a TCP

UDP – Discussione

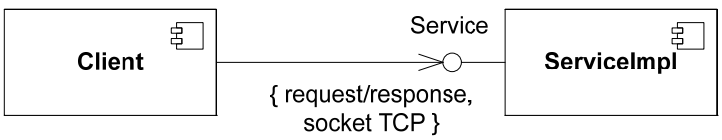
- Ulteriori aspetti interessanti
  - servizi stateless e stateful
    - un servizio è *stateless* se non deve gestire lo stato della conversazione con i suoi client
    - un servizio è *stateful* se gestisce lo stato della conversazione con i suoi client – l'effetto dell'esecuzione di un'operazione può dipendere dalla storia della conversazione con il particolare client
    - discuteremo di servizi stateful con riferimento a TCP

UDP – Discussione

- Per concludere, 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

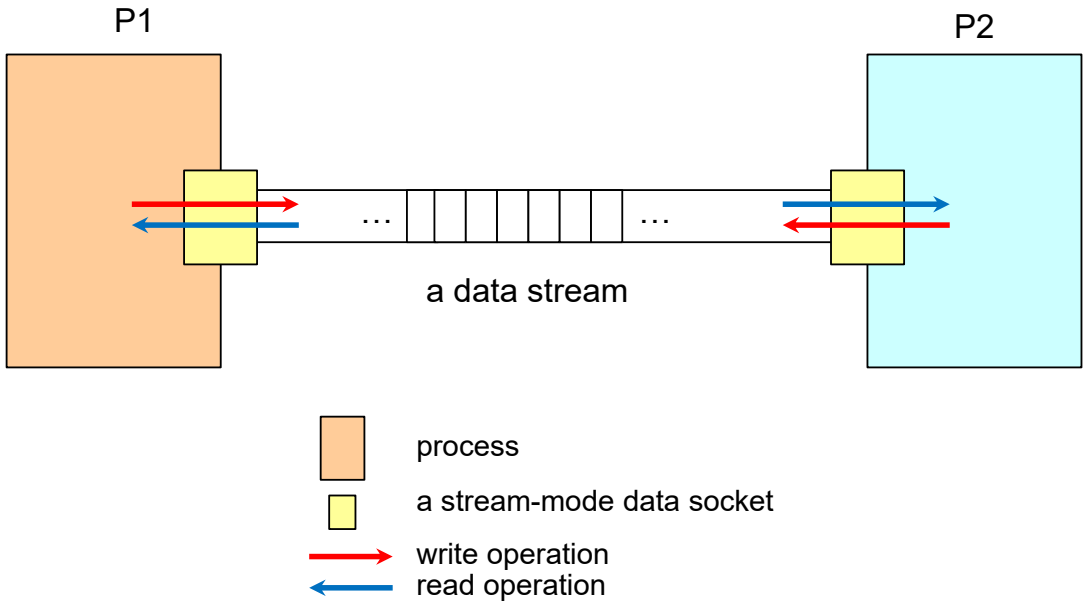
- Viene ora esemplificata una semplice applicazione di tipo client-server, con un protocollo richiesta/risposta, realizzata con socket TCP
  - con TCP, i messaggi vengono scambiati su un canale di comunicazione bidirezionale
  - il server è concorrente – multithreaded
    - intuitivamente, ogni volta che viene ricevuta una richiesta, il proxy lato server crea un nuovo thread per gestire la richiesta



  - è interessante notare che le differenze rispetto all'applicazione client-server UDP mostrata in precedenza sono localizzate tutte nel codice del "connettore" – ma non dei componenti

- API di Java per socket TCP

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

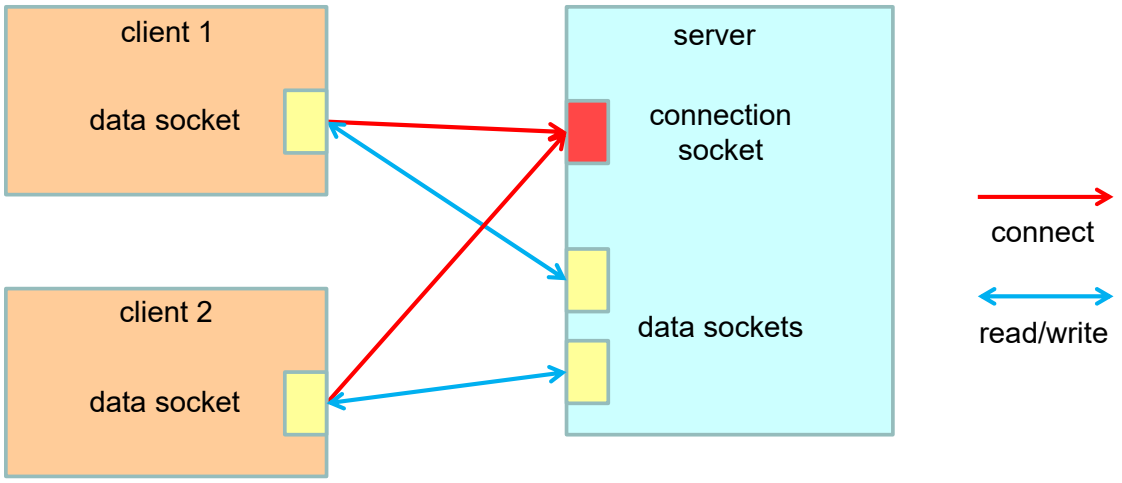


API di Java per socket TCP

- 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, i due processi possono agire come pari (*peer*), con operazioni
  - **read** – bloccanti
  - **write** – non bloccanti
- Sono definiti 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 tra tutti i client)
  - un altro tipo per le operazioni read e write (non condivisi)



API: ServerSocket e Socket

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

- Il servizio

- Le definizioni del **Service** (con le relative eccezioni) e di **ServiceImpl** sono come prima

```
package asw.socket.service;

/* Interfaccia del servizio Service. */
public interface Service {

   public String alpha(String arg)
         throws ServiceException, RemoteException;

   public String beta(String arg)
         throws ServiceException, RemoteException;

}
```


Il servizio

- Le definizioni del **Service** (con le relative eccezioni) e di **ServiceImpl** sono come prima

```
package asw.socket.service;

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

```
package asw.socket.service;

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

Il servizio

- Le definizioni del **Service** (con le relative eccezioni) e di **ServiceImpl** sono come prima

```
package asw.socket.service.impl;

import asw.socket.service.*;

/* Implementazione del servizio Service. */
public class ServiceImpl 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.socket.client;

import asw.socket.service.*;

/* client del servizio */
public class Client {

   private Service service;

   public Client() { }

   public void setService(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

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

```
package asw.socket.main;

import asw.socket.client.Client;
import asw.socket.context.ApplicationContext;

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

   public static void main(String[] args) {
      Client client = ApplicationContext.getInstance().getClient();
      client.run();
   }

}
```


Application context lato client

- Nell'application context, cambia solo il metodo **getService**
  - ipotizziamo che il server sia su **10.11.1.111** e ascolti sulla porta **7896**

```
/* Factory method per il servizio Service. */
public Service getService() {
   Service proxy = null;
   try {
      InetAddress address = InetAddress.getByName("10.11.1.111");
      int port = 7896;
      proxy = new ServiceClientTCPProxy(address, port);
   } catch(Exception e) { e.printStackTrace(); }
   return proxy;
}
```

Il proxy lato client (1)

```
package asw.socket.client.connector;

import ...

/* 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 ServiceClientTCPProxy(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 diversi dettagli

```
/* metodo di supporto per la comunicazione remota */
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 ...
      ... chiudi il socket ...
   } 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 diversi dettagli

```
/* chiede una connessione al server */
socket = new Socket(address, port);    // bloccante
/* imposta un timeout (1 sec) */
socket.setSoTimeout(1000);
/* i due flussi di comunicazione con il server */
DataInputStream in =
   new DataInputStream(socket.getInputStream());
DataOutputStream out =
   new DataOutputStream(socket.getOutputStream());

/* prepara la richiesta, di 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 (6)

- Una possibile implementazione di **doOperation**

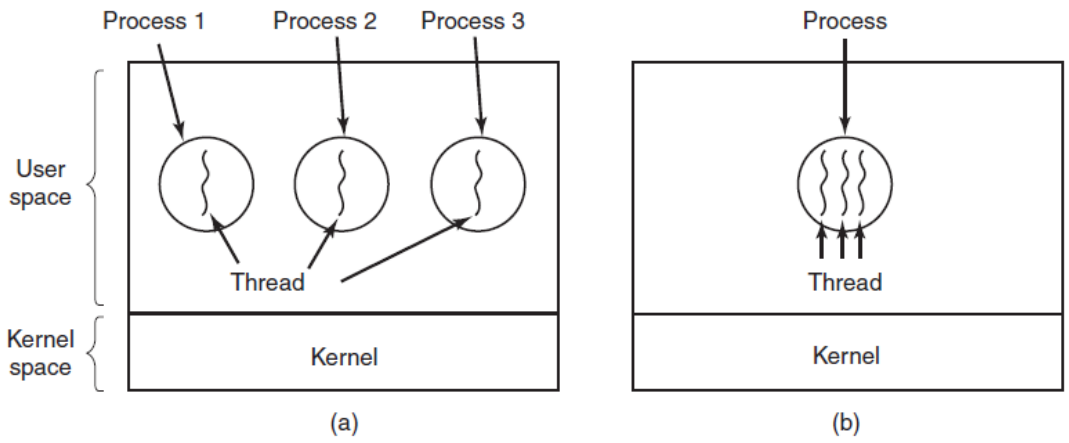
```
/* elabora la risposta, che può avere
 * le seguenti forme (vedi proxy lato server):
 * "#risultato" "@eccezione di servizio" "!eccezione remota" */
if ( reply.startsWith("#") ) {  // è un risultato
   result = reply.substring(1);
} else if ( reply.startsWith("@") ) {  // ServiceException
   String message = reply.substring(1);
   throw new ServiceException(message);
} else if ( reply.startsWith("!") ) {  // RemoteException
   String message = reply.substring(1);
   throw new RemoteException(message);
} else {  // risposta malformata
   throw new RemoteException("Malformed reply: " + reply);
}
/* chiudi il socket */
socket.close();
return result;
```

- Processi e thread

- Lato server, il server è in esecuzione in un suo processo – ogni volta che il server riceve un messaggio di richiesta da un client, crea un nuovo thread per gestire questa richiesta
- Nei sistemi operativi
  - un **processo** è un'istanza di un programma in esecuzione
    - l'OS assegna a un processo le risorse necessarie per eseguire il programma – come uno spazio degli indirizzi (privato) e altre risorse (ad es., delle connessioni)
    - ogni processo ha almeno un *thread* ("filo") di esecuzione – ma può averne più di uno
  - un **thread** è un'entità che può essere assegnata a un processore ed eseguita dal processore
    - una sequenza di istruzioni – che rappresenta un'"attività" o un "compito" di elaborazione nel processo

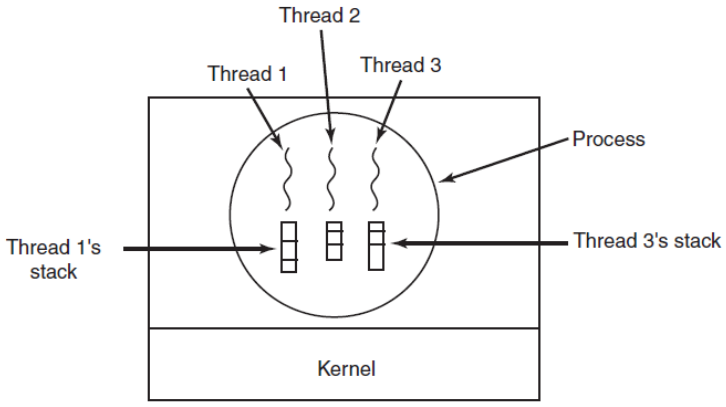
Processi e thread

- Esempi
  - (a) tre processi con un thread ciascuno
  - (b) un processo con tre thread



Processi e thread

- I thread di un processo
  - sono eseguiti in modo concorrente tra loro
  - condividono con il processo le sue risorse – certamente lo spazio degli indirizzi
    - lo heap di un processo è condiviso da tutti i suoi thread
    - ogni thread ha un proprio stack

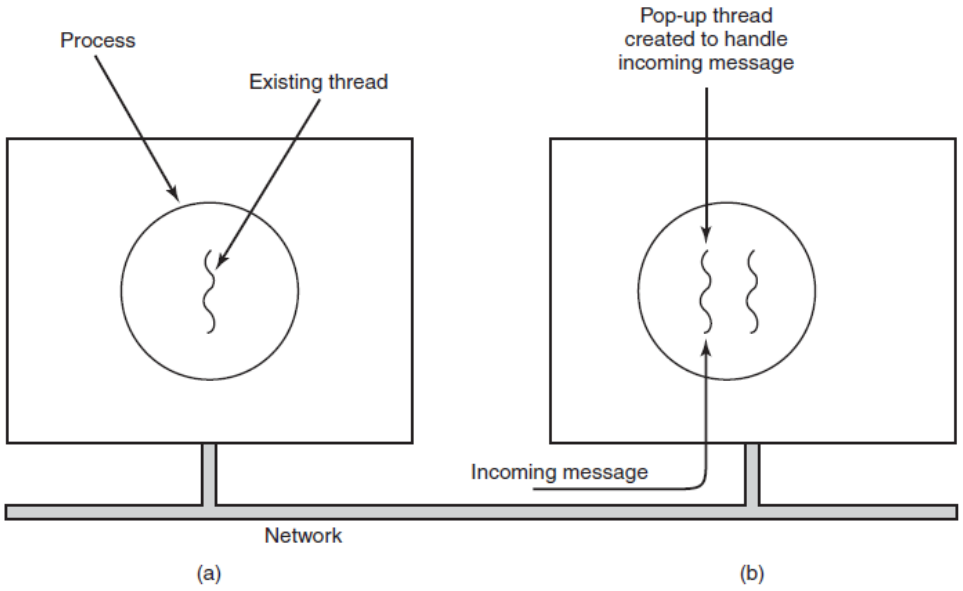


API di Java per i thread

- In Java, i thread sono rappresentati dalla classe **Thread**
  - **Thread** implementa l'interfaccia **Runnable**
    - l'interfaccia **Runnable** definisce un metodo **run()** – che rappresenta un'attività che deve essere eseguita in un proprio thread
  - di solito, il programmatore definisce una classe che estende **Thread** e, in particolare, implementa **run()**
    - attenzione, il metodo **run()** non va chiamato direttamente – altrimenti verrebbe eseguito nel thread/stack del chiamante
    - piuttosto, il metodo **run()** va avviato chiamando il metodo **start()** di **Thread** – che crea il nuovo thread, avvia l'esecuzione del metodo **run()** nel nuovo thread, e poi termina immediatamente
  - in alternativa, il metodo **run()** di un **Runnable r** può essere mandato in esecuzione con l'istruzione **new Thread(r).start();**

- Il server

- Il server è in esecuzione in un suo processo – ogni volta che il server riceve un messaggio di richiesta da un client, crea un nuovo thread (pop-up thread) per gestire questa richiesta



Il server

- L'oggetto **Server** – eseguito su **10.11.1.111** e ascolta su **7896** – è responsabile di
  - creare un'istanza di **ServiceImpl**
  - creare e avviare il proxy lato server

```
package asw.socket.server.connector;

import asw.socket.service.Service;
import asw.socket.service.impl.ServiceImpl;

/* server per il servizio */
public class Server {

   public static void main(String[] args) {
      Service service = new ServiceImpl();
      int port = 7896;
      ServiceServerTCPProxy server =
         new ServiceServerTCPProxy(service, port);
      server.run();
   }

}
```

Il proxy lato server (1)

```
package asw.socket.server.connector;

import asw.socket.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

```
public void run() {
   try {
      /* crea il server socket
       * su cui ascoltare/ricevere richieste */
      ServerSocket listenSocket = new ServerSocket(port);
      /* per il server, disabilita il timeout */
      listenSocket.setSoTimeout(0);
      /* ciclo di gestione delle richieste */
      while (true) {
         ... segue ...
      }
      listenSocket.close();
   } catch (Exception e) { ... gestisci eccezione ... }
}
```

Il proxy lato server (3)

- Il metodo **run** del “remote proxy” lato server
  - questa volta il ciclo di gestione delle richieste implementa un server concorrente

```
/* ciclo di gestione delle richieste */
while (true) {
   /* aspetta/accetta una richiesta e,
    * quando arriva, crea il relativo socket */
   Socket clientSocket = listenSocket.accept();  // bloccante
   /* la richiesta sarà gestita
    * in un nuovo pop-up thread, separato */
   ServerThread thread =
      new ServerThread(clientSocket, service);
   thread.start();    // avvia il nuovo thread
   /* poi torna immediatamente ad aspettare
    * la richiesta successiva */
}
```


Gestione di una richiesta (1)

```
package asw.socket.server.connector;

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

public class ServerThread extends Thread {

   private Service service;
   private Socket clientSocket;
   private DataInputStream in;
   private DataOutputStream out;

   public ServerThread(Socket clientSocket, Service service) {
      try {
         this.clientSocket = clientSocket; this.service = service;
         in = new DataInputStream(clientSocket.getInputStream());
         out = new DataOutputStream(clientSocket.getOutputStream());
      } catch (IOException e) { ... gestisci eccezione ... }
   }
   ... segue ...
}
```

Gestione di una 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 ...;

      ... invoca il servizio, ottieni il risultato,
          calcola la risposta reply ...

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


Gestione di una richiesta (3)

- Questa parte della gestione della richiesta è come prima

```
/* invoca il servizio, ottieni il risultato,
 * calcola la risposta reply */
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) {
   /* il servente non solleva MAI RemoteException,
    * ma si può arrivare qui se la richiesta è malformata */
   reply = "!" + e.getMessage();
}
```

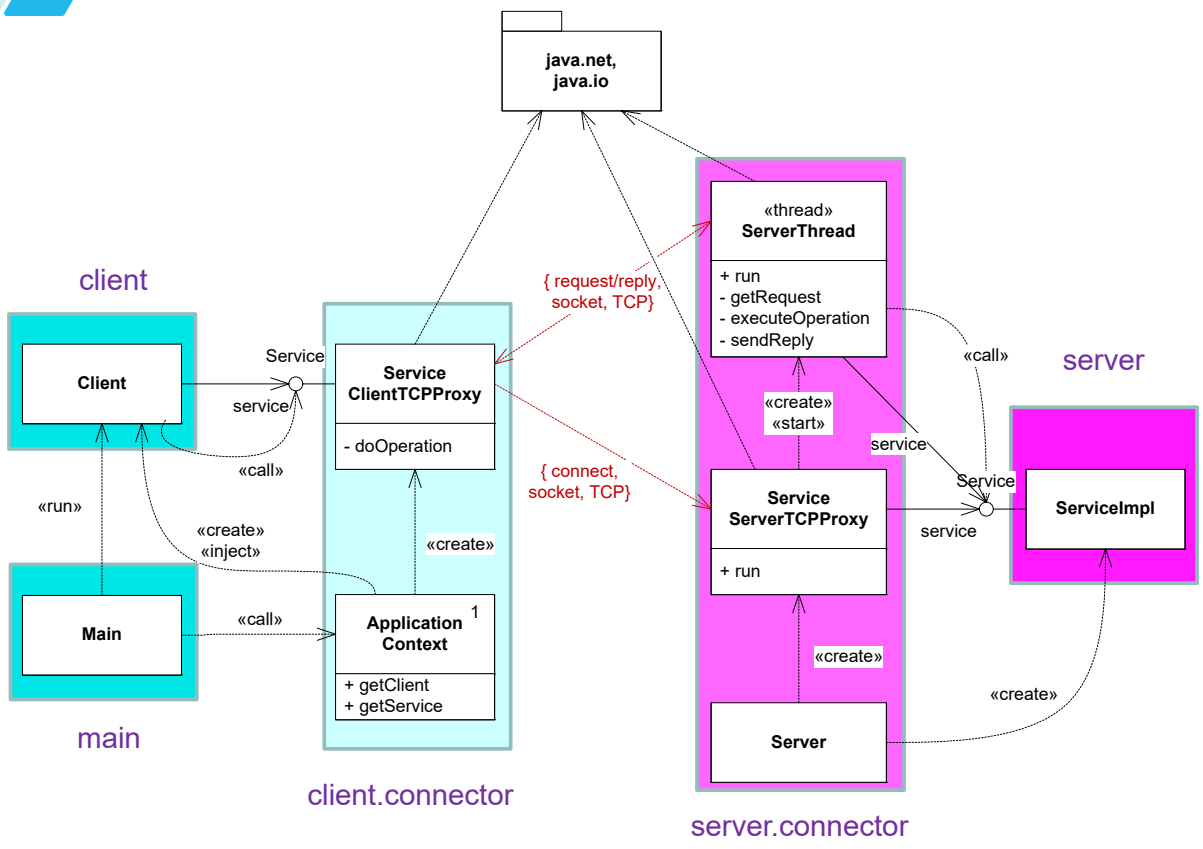
Gestione di una richiesta (4)

- Questa parte della gestione della richiesta è come prima

```
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 not supported");
   }
   return result;
}
```


- In sintesi

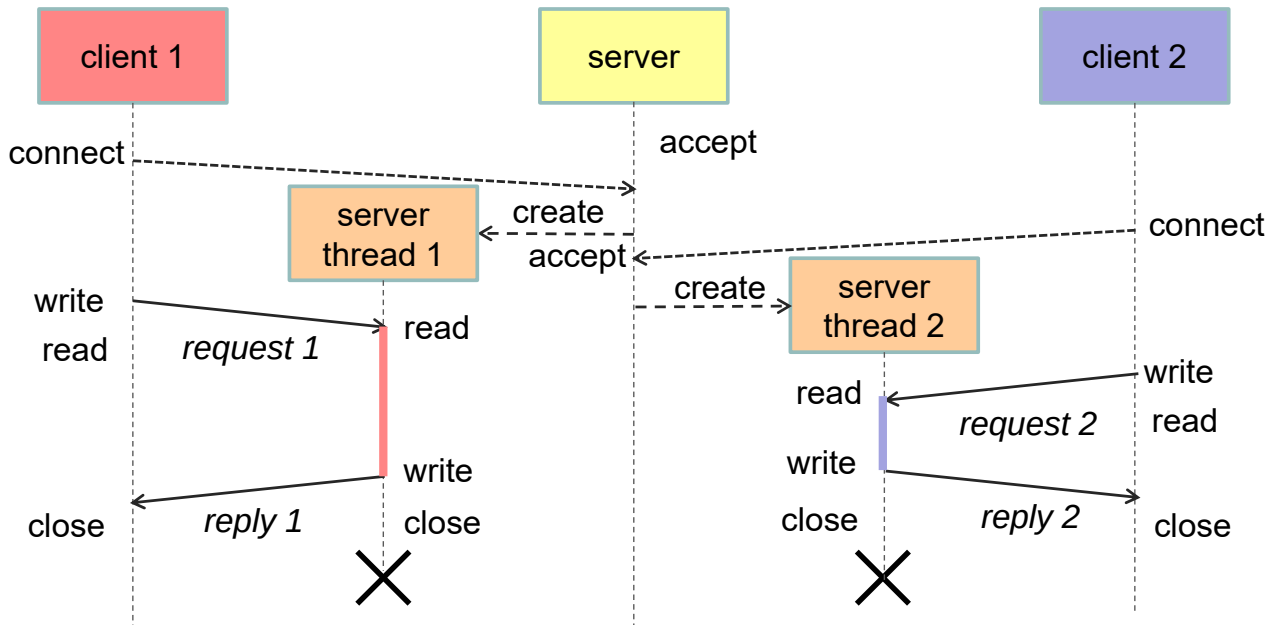


In sintesi

- Interazione tra client e server
  - il client vuole interagire con il server
    - il client richiede una connessione al server
  - il server accetta la richiesta di connessione
    - il server crea un nuovo thread per gestire l'interazione con quel client, poi è immediatamente pronto ad accettare altre richieste
  - il client vuole chiedere al server di eseguire un'operazione
    - il client invia al server una richiesta che codifica l'invocazione dell'operazione
  - il server esegue l'operazione richiesta dal client
    - il thread del server dedicato a quel client legge la richiesta
    - poi il thread del server decodifica la richiesta, esegue l'operazione richiesta e calcola il risultato
    - il thread del server codifica il risultato in una risposta, che poi invia al client
  - il client riceve il risultato dell'operazione richiesta
    - il client riceve la risposta e poi estrae il risultato dalla risposta

In sintesi

- Esempio – con due client (vengono gestito in modo concorrente)



- 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 bisogna 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 protocollo di comunicazione implementato da ciascun connettore
  - nel caso dei servizi di middleware è utile comprendere, oltre al paradigma di comunicazione che implementa, anche la sua struttura e i suoi principi di funzionamento

TCP – Discussione

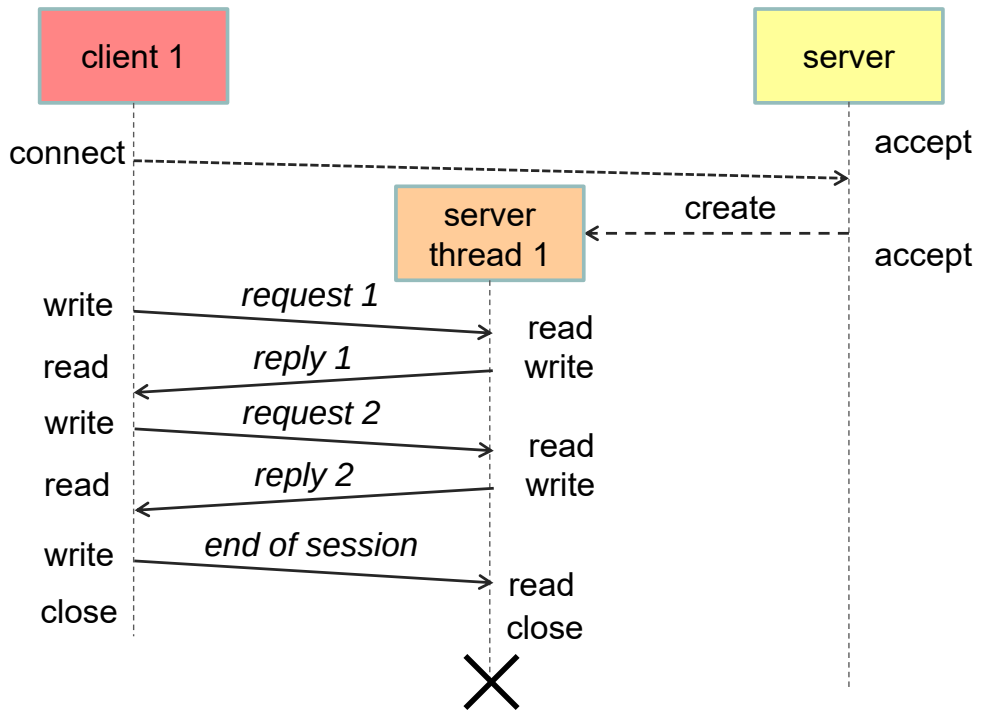
- Gestione delle connessioni
  - nella versione mostrata
    - il client ottiene e richiede una connessione al server per ciascuna singola richiesta di servizio
    - in corrispondenza, il server crea e alloca un thread per gestire ciascuna singola richiesta
    - vengono dunque usate connessioni "non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

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 esempio
    - 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 connessione "persistente" o conversazione o sessione tra client e server
      - in pratica, il metodo **run()** del **ServerThread** potrebbe gestire una sequenza di richieste di uno stesso cliente (mediante un'istruzione ripetitiva)
    - oppure, può essere usato un pool di connessioni
    - queste scelte hanno impatto su prestazioni e scalabilità

TCP – Discussione

- Esempio – gestione di una conversazione con un client



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 thread del server (ovvero, ciascuna istanza di server thread) può essere dedicato alla gestione di tutta la conversazione con un particolare client
    - per questo, a ciascun server thread può essere assegnata anche la responsabilità di gestire lo stato della conversazione (sessione) con quel particolare client
  - il server proxy è condiviso da tutti i client, e viene usato per iniziare nuove conversazioni (è una factory di server thread)
    - 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
  - le astrazioni di comunicazione distribuita che implementano sono di basso livello – inoltre il programmatore deve prendersi carico direttamente di sopperire alle limitazioni dei socket
  - in alternativa, il programmatore può decidere di usare un servizio specifico di middleware, in grado di
    - superare le limitazioni riscontrate
    - offrire un paradigma di programmazione di livello più alto e più semplice da utilizzare – e che nasconde meglio la complessità della comunicazione
    - a costo, probabilmente, di un qualche overhead

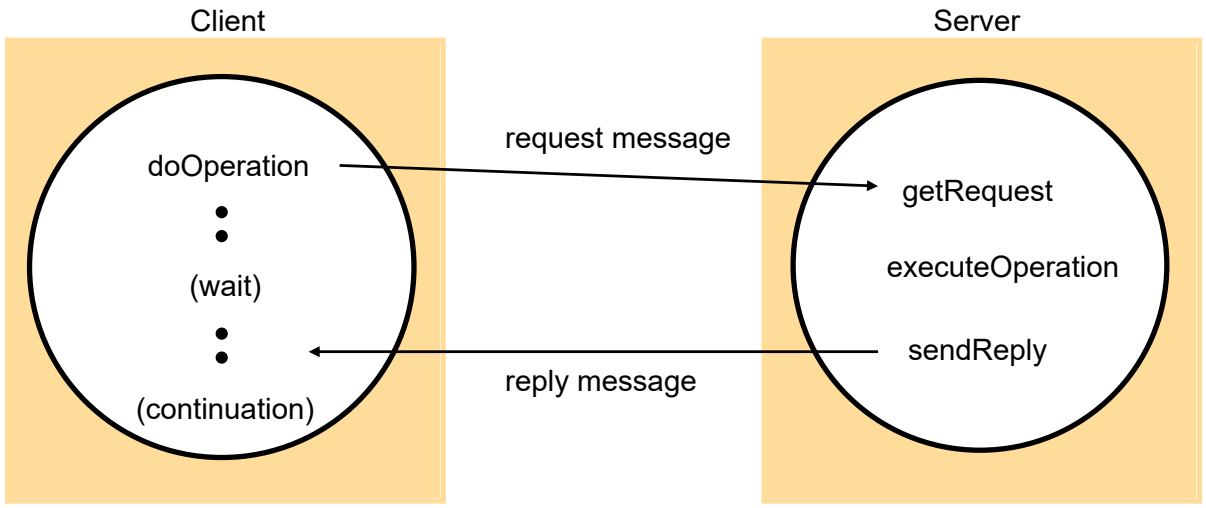
* Messaggi e protocolli

- 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?

Protocollo richiesta-risposta

- Finora abbiamo esemplificato un protocollo richiesta-risposta – lo abbiamo usato per realizzare una "chiamata di operazioni remote"
  - un server espone, mediante un opportuno protocollo, alcune operazioni che possono essere invocate remotamente
    - un protocollo definisce, per ciascuna operazione
      - il formato del messaggio di richiesta, per invocare l'operazione
      - il formato del messaggio di risposta, per il risultato
    - un protocollo può definire anche l'ordine con cui è possibile richiedere le varie operazioni
  - i client possono chiedere al server l'esecuzione di operazioni remote adeguandosi a questo protocollo e a questi formati
  - si tratta di una modalità di comunicazione comune (ma non è l'unica possibile) – che è implementata anche dal middleware di tipo RPC/RMI

Protocollo richiesta-risposta



Messaggi per chiamate remote

- Tipi di messaggi coinvolti nella chiamata di operazioni remote
  - *richiesta*
    - codifica l'invocazione dell'operazione richiesta, inclusi i suoi parametri attuali
  - *risposta*
    - codifica i risultati restituiti
    - può codificare anche un'eccezione sollevata durante l'esecuzione dell'operazione richiesta
- Nell'esempio precedente – relativo a un caso "giocattolo"
  - le richieste sono stringhe nella forma *operazione*$*argomento*
  - le risposte relative a risultati sono stringhe nella forma #*risultato*
  - le risposte relative alle eccezioni sono stringhe nella forma @*eccezionediservizio* oppure !*eccezioneremota*

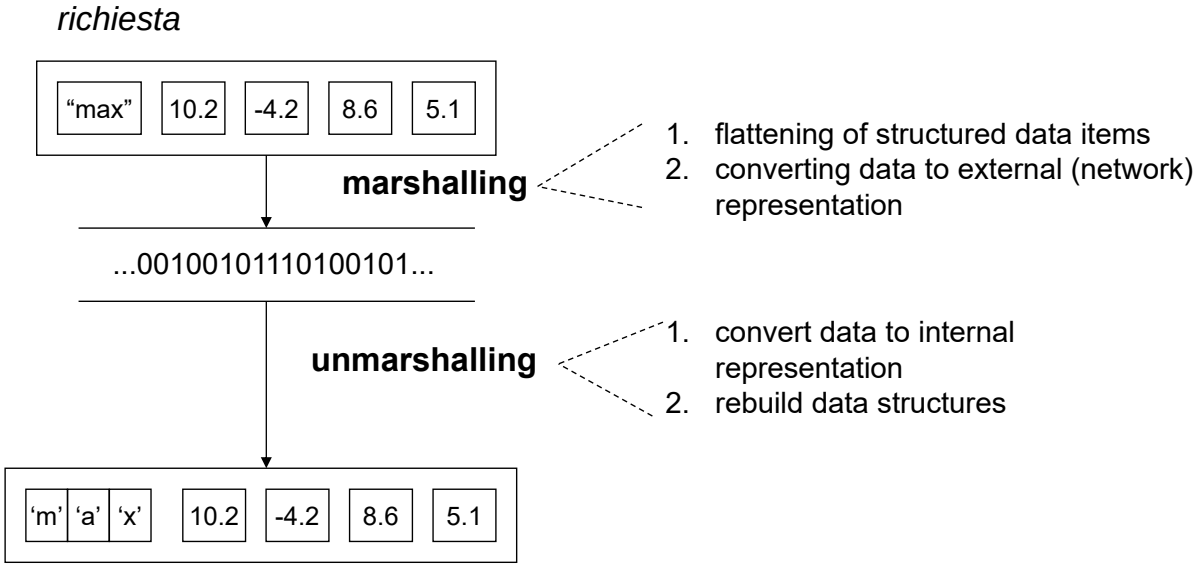
Formato dei messaggi

- Nella pratica (e nei servizi di middleware RPC/RMI) vengono invece utilizzate altre codifiche – ecco alcune possibilità riguardo al formato per i messaggi
  - una prima distinzione è tra messaggi in formato testuale e messaggi in formato binario
  - una distinzione più importante riguarda l'uso di formati proprietari oppure di formati di interscambio standardizzati
    - esempi di formati standard testuali e autodescrittivi sono XML e JSON
    - esempi di formati standard binari sono Protocol Buffer (Google) e Avro (Apache)
    - un esempio di formato proprietario è la serializzazione di Java (usata da Java RMI)
  - queste codifiche vengono di solito generate da opportuni "compilatori di interfacce"

Codifica e decodifica di messaggi

- Le due parti devono essere d'accordo sul formato (sintassi e semantica) dei messaggi scambiati (di richiesta e di risposta)
  - le due parti devono occuparsi delle attività di *codifica* e *decodifica* dei messaggi di richiesta e di risposta
    - ad es., il client deve codificare l'invocazione di un'operazione che vuole richiedere in un messaggio di richiesta
    - il server deve decodificare un messaggio di richiesta per capire qual è l'operazione invocata a quali sono i suoi parametri
  - si parla anche di marshalling e unmarshalling di dati e richieste
    - *marshalling* – codificare/assemblare un gruppo di dati in una forma adatta ad essere trasmessa in un messaggio
    - *unmarshalling* – decodificare/disassemblare il contenuto di un messaggio – per estrarre i dati in esso contenuti e ricostruire eventuali strutture di dati

Codifica e decodifica di messaggi



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 la codifica della richiesta (operazione e dati)
- invio il messaggio di richiesta
- ricevo il messaggio di risposta
- effettuo la decodifica della risposta (risultati)

programma server

sono in attesa che un client mi faccia qualche richiesta:
- ricevo il messaggio di richiesta
- effettuo la decodifica della richiesta (operazione e dati)
- eseguo l'operazione richiesta (calcolo i risultati a partire dai dati)
- effettuo la codifica della risposta (risultati)
- invio il messaggio di risposta

Altri tipi di comunicazione

- Oltre alla comunicazione richiesta-risposta esistono anche altre modalità di comunicazione – ecco alcuni esempi
  - 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 **Producer** invia dei dati a un componente **Consumer**
    - non ha bisogno di una risposta
  - un componente **Publisher** invia 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

- I servizi 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 un servizio 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 (scegliere anche se è più opportuno usare socket UDP o TCP)
  - 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 (scegliere anche se è più opportuno usare socket UDP o TCP)
  - 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 modi per realizzare queste applicazioni!