INGEGNERIA DEL SOFTWARE

MIDDLEWARE

Avvertenza: gli appunti si basano sul corso di Ingegneria del Software tenuto dal prof. Picco della facoltà di Ingegneria del Politecnico di Milano (che ringrazio per aver acconsentito alla pubblicazione). Essendo stati integrati da me con appunti presi a lezione, il suddetto docente non ha alcuna responsabilità su eventuali errori, che vi sarei grato mi segnalaste in modo da poterli correggere.

e-mail: webmaster@morpheusweb.it
web: http://www.morpheusweb.it

IL PARADIGMA CLIENT SERVER

Con il termine **client/server** ci riferiamo ad uno stile architetturale per applicazioni distribuite.

Client e server sono processi distinti aventi una ben precisa interfaccia (API)

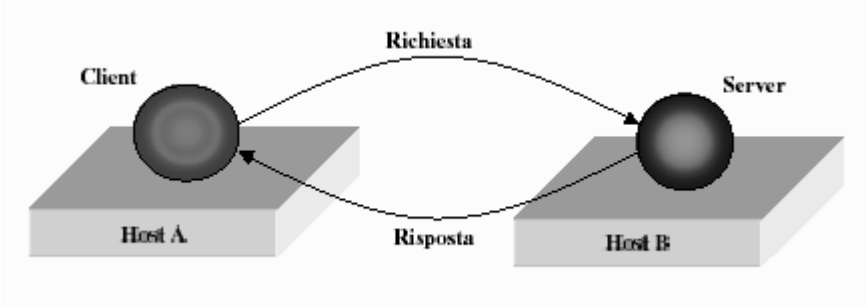
- Sono accessibili solo attraverso tale API
- Possono essere formati da un insieme di moduli software e/o hardware

I client sono i componenti attivi ed invocano i servizi dei server che hanno un ruolo passivo.

Client e server risiedono su macchine diverse connesse da una rete di comunicazione.

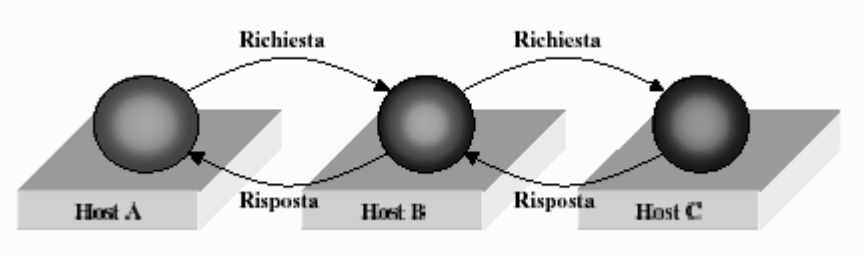
La comunicazione avviene attraverso scambio di messaggi o chiamate remote di procedura.

SCHEMA DI FUNZIONAMENTO



Il client richiede un servizio al server, il server elabora le informazioni e da una risposta.

A sua volta un server potrebbe essere il client di un altro server.



Un esempio è il world wide web, in cui richiediamo una pagina tramite il nostro browser, e questa ci viene restituita in risposta dal server web con cui siamo collegati.

SCOMPOSIZIONE LOGICA DI UN'APPLICAZIONE

Le architetture possono differire a seconda di cosa metto sul client e cosa sul server.

Possiamo scomporre le **funzionalità** in tre insiemi:

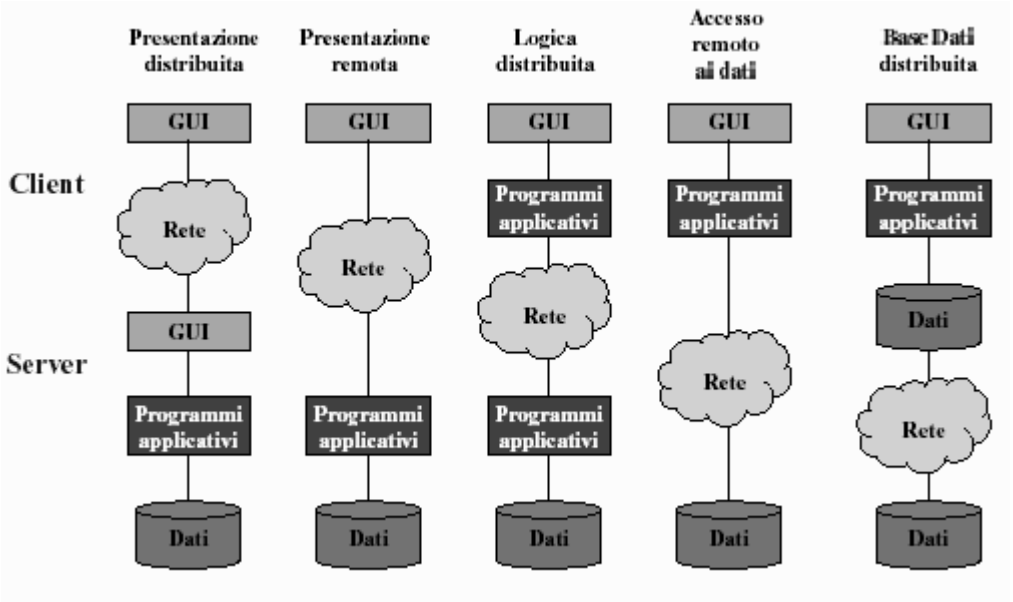
- interfaccia utente
- elaborazione
- gestione persistente dei dati

A seconda di come sono allocate queste funzionalità distinguiamo tra:

- architetture a due livelli (two-tiered architectures)
- architetture a tre livelli (three-tiered architectures)

ARCHITETTURE A DUE LIVELLI

Abbiamo diverse tipologie a seconda di dove viene fatto il taglio.



Presentazione distribuita:
Tutta la logica è nel server, ed il client si occupa solo di interagire col server che gli affida compiti specifici di presentazione.
Ad esempio sistemi alla x-windows, il client ha solo una parte di gestione dell'interfaccia grafica, oppure una forn html pura (senza javascript)

Presentazione remota:
La logica applicativa ed i dati stanno sul server, il client fa da pannello di controllo remoto (sono così la maggior parte delle applicazioni web).

Logica distribuita:
Il client contiene anche una parte della logica distribuita. Cerco di alleggerire il server dal punto di vista computazionale.
Il client si appesantisce, ma si rende più scalabile il server.

Accesso remoto ai dati:
Il server fornisce solo l'accesso ai dati grezzi, il client è un front end per la gestione dei dati.

Base di dati distribuita:
Il client contiene anche una parte dei dati.

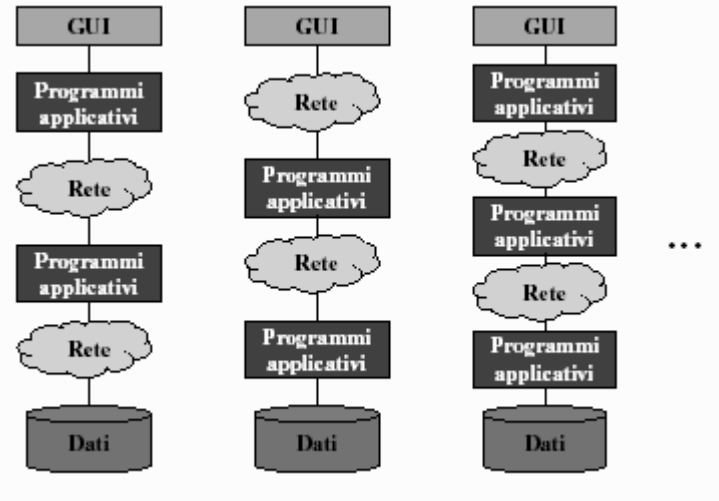
ARCHITETTURE A TRE LIVELLI

Prima avevamo un unico taglio, in questo caso scomponiamo l’applicazione supponendo di avere un altro nodo che fa da server.

Lo schema tipico è il seguente:

Che consente di avere un'unica base di dati, server diversi specializzati per funzionalità o domini applicativi diversi che insistono su un’unica base di dati.

Abbiamo poi diversi schemi alternativi.



FAT CLIENT E FAT SERVER

Mi dice quanto client o server sono sofisticati, quanta logica contengono.

Un client è **fat** se contiene almeno una parte della logica dell'applicazione.

Vantaggi dei fat client:

- il sistema risponde agli input in modo più rapido (parte dei servizi sono in locale)
- il sistema è in grado di fornire un riscontro all'utente con un livello di granularità più fine
- il sistema è più scalabile
- è più facile implementare nei client interfacce grafiche per le esigenze degli utenti

Svantaggi dei fat client

- l'interazione con il server può diventare troppo frequente, occupando eccessivamente i canali di comunicazione.
- si perde l'incapsulamento (il client deve avere una conoscenza approfondita dell'organizzazione dei dati nel server)
- la gestione e l'aggiornamento del sistema diventa più oneroso poiché quando cambio un servizio devo aggiornare tutti i client e non solo il server

MIDDLEWARE

Il **middleware** è uno strato di software posto a metà fra il sistema operativo e le applicazioni.
E' tipicamente impiegato per facilitare lo sviluppo di applicazioni in un sistema distribuito.

Invece di usare i socket, posso usare il middleware basato su invocazione di procedure remote.

Cerco di mascherare gli aspetti relativi alla distribuzione.

E' tipicamente usato per sistemi distribuiti (una collezione di calcolatori indipendenti che appare agli utenti come un unico calcolatore)

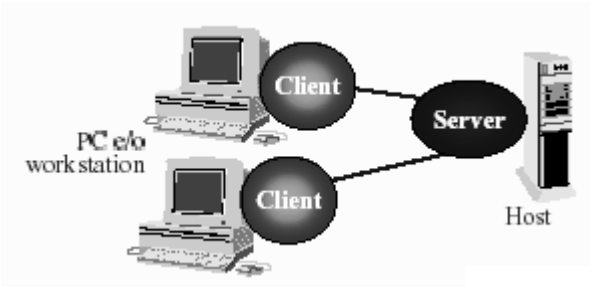
SISTEMI CENTRALIZZATI

Le applicazioni girano su un singolo processo, o comunque su un solo host, che costituisce l'unico componente autonomo del sistema.
Tale componente è condiviso da vari utenti. Tutte le risorse del componente sono sempre accessibili.
Il componente costituisce il "*single point of controlI*" ed il "*single point of failure*" del sistema.

E' l'architettura hardware ad essere centralizzata, tutti i terminali non hanno la cpu.

SISTEMI DISTRIBUITI

Le applicazioni sono costituite da più processi, fortemente cooperanti, eseguiti in parallelo su un insieme di unità di elaborazione (cpu) autonome. Senza condividere memoria, comunicando tramite messaggi con un ritardo di trasmissione non trascurabile.



Nella figura, l'architettura è distribuita a livello hardware, ma centralizzata a livello software in cui c'è la logica applicativa. (ad esempio le architetture peer-to-peer sono distribuite anche a livello software)

VANTAGGI DEI SISTEMI DISTRIBUITI

I sistemi che concentrano grandi capacità di computazione sono molto costosi, mentre il costo delle cpu e dell'infrastruttura di comunicazione si è ridotto notevolmente: un sistema distribuito può avere più potenza computazionale di un mainframe.

E' più facile gestire la crescita del sistema (nodi o numero di utenti) in modo incrementale.

E' possibile ripartire il carico di lavoro sulle macchine in maniera più efficiente

E' possibile condividere dati e risorse computazionali

Alcune applicazioni sono inerentemente distribuite, (es. servizi di prenotazione); nuove applicazioni sono possibili (es. e-mail)

I sistemi distribuiti sono maggiormente **fault-tollerant**: il sistema può continuare a lavorare anche se alcune sue parti sono fuori uso.

L'enorme diffusione di Internet ha portato i sistemi distribuiti a ricoprire un ruolo fondamentale ("*the network is the computer*").

SVANTAGGI DEI SISTEMI DISTRIBUITI

Latenza: la differenza tra l'invocazione locale e remota di un servizio è di 4-5 ordini di grandezza.

Accesso alle risorse: ad esempio, i puntatori in uno spazio di indirizzamento locale non sono più validi in remoto.

Partial failure e concorrenza: la computazione locale *può* essere concorrente, quella distribuita lo è di certo. Una computazione locale fallisce soltanto in maniera completa, ma una distribuita può farlo anche in maniera parziale. Come si mantiene la consistenza? In una computazione locale esiste un solo punto di controllo per l'allocazione delle risorse, la sincronizzazione, la gestione dei malfunzionamenti, ciò non è vero in un sistema distribuito.

Sicurezza: la facilità di accesso vale anche per i malintenzionati.

PROPRIETA' DESIDERABILI

Access transparency: l'utente non percepisce la differenza tra accessi locali e remoti.

Location trasparency: dati e servizi sono acceduti senza conoscere la loro locazione fisica

Concurrency transparency: processi diversi possono condividere risorse senza interferenza

Replication transparency: è possibile usare copie degli oggetti nel sistema distribuito per migliorare le prestazioni.

Failure transparency: i malfunzionamenti sono mascherati

Migration transparency: dati e servizi possono essere rilocati

Performance transparency: il sistema può essere riconfigurato dinamicamente per adattarsi al carico

Scaling transparency: il sistema può cambiare scala senza cambiamenti nella sua struttura o negli algoritmi applicativi

PROGRAMMAZIONE DI RETE IN JAVA: I SOCKET

IL SOCKET

Il socket è stato introdotto in Unix BSD come generalizzazione del concetto di pipe Unix, per consentire lo scambio di sequenze di caratteri o byte fra processi dislocati su macchine remote.

Ormai disponibili per tutte le piattaforme e linguaggi

L'astrazione è quella di un tubo attraverso cui scorrono stream di dati.

Caratteristiche della comunicazione:

- Tipicamente punto-punto (unicast), ma esistono estensioni che permettono comunicazione multi-punto (multicast)
- La sorgente della comunicazione deve conoscere l'identità (indirizzo IP e numero di porta) del destinatario, la comunicazione è esplicita.
- La serializzazione e il marshalling sono a carico delle applicazioni
    - **Serializzazione**: trasformazione fra dati strutturati e sequenze di byte, consente di appiattire i dati in una sequenza lineare per farli passare nel canale.
    - **Marshalling**: conversione fra rappresentazioni di dati diverse (per sopperire alle diversità di piattaforma)

SOCKET IN JAVA

Forniscono le funzionalità base dei socket, più

- Socket su **UDP** e **TCP**
- Supporto per socket multicast (su UDP)
- Trasformazione di dati strutturati in byte stream e viceversa (serializzazione)
- Possibilità di utilizzare implementazioni diverse dello stack di rete impiegando le medesime classi

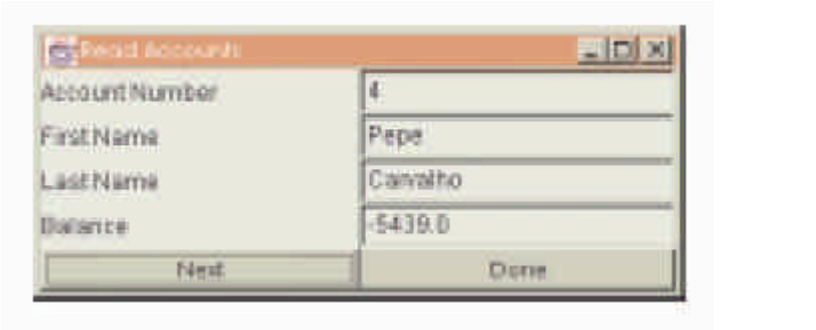
Implementati dalle classi nel package **java.net**

Su UDP: ha meno overhead, è connection-less; non garantisce ordinamento e consegna affidabile.

Su TCP: ha più overhead, è connection-oriented; è affidabile ma con più overhead

ESEMPIO: ACCESSO REMOTO AI CONTI CORRENTI

Il server fornisce accesso al contenuto del file **accounts.dat**, presente sul suo file system
I client remoti, dopo aver inizializzato la connessione con il server specificando indirizzo IP e porta, sono in grado di sfogliare i dati relativi ai conti correnti gestiti dal server



ESEMPIO: IL CLIENT

```
…
InetAddress host = …
s = new Socket(host, port);
input = new DataInputStream(s.getInputStream());
output = new DataOutputStream(s.getOutputStream());
…
public void readRecord() {
      try {
            output.writeShort(NEXT);
            accountNumber = input.readInt();
            first = input.readUTF();
            last = input.readUTF();
            balance = input.readDouble();
            } catch (EOFException e) { closeConnection();
            } catch (IOException e) { System.exit(1); }
      }
```

`InetAddress host = …` → E' la classe per rappresentare gli host, contiene l'ip.

`s = new Socket(host, port);` → devo creare un oggetto socket, creare una connessione di rete tra client e server

Tramite s, posso ottenere gli stream di input ed output che sono usati per leggere dal socket e scrivervi.

`output = new DataOutputStream(s.getOutputStream());` → Utilizzo il decorator pattern per leggere la sorgente. DAtaOutputStream mi consente di leggere in termini di dati e non di byte.

`public void readRecord() {` → legge un record in remoto dal server.

`output.writeShort(NEXT);` → esportato da DataOutputStream, scrive un valore il cui tipo è uno short.

Dopo il client si sospende su un operazione di lettura finchè non c'è un dato sullo stream di input.

`accountNumber = input.readInt();` → legge un intero dallo stream. Quando il dato arriva, viene copiato nella variabile accountNumber.

Poi leggo gli altri dati:

`readUTF();` → legge una stringa da un input stream.

Il tutto messo in un blocco try/catch per gestire la fine dei dati o la disconnessione.

ESEMPIO: IL SERVER

Il client apre una connessione, ma sul server deve esserci un processo che ascolta su una certa porta, accetta la richiesta ed apre il canale di comunicazione.

Sul server deve esserci la porta che ascolta in attesa di connessione.

C'è una classe ServerSocket che viene usata per ascoltare sulla porta.

```
try {
      server = new ServerSocket(port);
      while(true) {
      client = server.accept();
      is = new DataInputStream(client.getInputStream());
      os = new DataOutputStream(client.getOutputStream());
      DataInputStream dataStream =
            new DataInputStream(new FileInputStream(DATAFILE));
      int command = -1;
      while(command != DONE) {
            command = is.readShort();
            switch(command) {
                  case NEXT:
                  os.writeInt(dataStream.readInt());
                  os.writeUTF(dataStream.readUTF());
                  os.writeUTF(dataStream.readUTF());
                  os.writeDouble(dataStream.readDouble());
                  break;
                  case DONE:
                  dataStream.close();
                  is.close();os.flush();os.close();client.close();
                  break;
                  }
            }
} catch(IOException ioe) { ioe.printStackTrace(); }
```

`server = new ServerSocket(port);` → creo una oggetto di tipo ServerSocket

`client = server.accept();` → da qui il server comincia ad ascoltare. Il comando accept() è bloccante finché non è accettata la connessione. Quando avviene, ritorna una variabile di tipo Socket.

Poi creo in locale due stream di input ed output associati al client.

In più il server gestisce i dati leggendo il file.

Creo un DataInputStream.

Il server va in ciclo finché il comando dato è diverso da DONE.

Con DONE: sono chiusi gli stream e il socket.
`os.flush();` → termino la scrittura
`os.close();` → chiudo lo stream (va fatto dopo aver terminato la scrittura)

Con NEXT: leggo i dati con lo stesso dataStream e li metto sull'output stream *os* associato al socket.

SERVER E CONCORRENZA

Il server non può accettare una nuova connessione finché non incontra nuovamente una **accept();** nell'esempio precedente ciò avviene soltanto quando il client decide di terminare la connessione.

accept() è bloccante, il server si blocca finché non c'è una connessione del client. Se mentre sto servendo un client, arriva una seconda richiesta da parte di un altro client, la connessione non è accettata ed è sospesa finché il server non si libera e fa un'altra accept().

Vogliamo che il server possa servire un client e contemporaneamente forzare a fare un accept() per ascoltare altri client.

Un server multithreaded consente di servire più client contemporaneamente, nonché di accettare nuove connessioni.

Tipicamente viene impiegato un thread per client

SERVER MULTITHREADED: ON DEMAND VS. THREAD POOL

I server multithreaded vengono tipicamente realizzati seguendo uno di questi schemi:

- **on demand**:il thread viene creato all'atto della richiesta di connessione, e terminato alla chiusura della medesima

- **thread pool**:un numero prefissato di thread viene creato ed attivato; alla chiusura della connessione, il thread relativo non non viene terminato, bensì viene sospeso e "riciclato" per una nuova connessione.

Il primo schema è più semplice ma più costoso, in quanto il tempo necessario ad attivare è dominato dalla creazione del thread

Il secondo schema consente di evitare questo overhead, ma richiede un dimensionamento del pool

È possibile anche impiegare uno schema misto (posso risolvere il problema del dimensionamento)

ESEMPIO: CREAZIONE ON DEMAND

```
public class AccountServer {
…
      try{
            server = new ServerSocket(port);
            while (true){
            Socket client = server.accept();
            new Thread(new ServerThread(client)).start();
            }
      } …
}
class ServerThread implements Runnable {
      Socket s = null;
      ServerThread(Socket s) { this.s = s; }
      public void run() {
            DataInputStream is =
                  new DataInputStream(s.getInputStream());
            DataOutputStream os =
                  new DataOutputStream(s.getOutputStream());
                  …
            while(command != DONE) {
                  command = is.readShort();
                        switch(command) {
                        case NEXT:
                        … server body come prima …
            }
      }
}
```

`server = new ServerSocket(port);` → viene creata una classe ServerTread, il cui metodo run() è quello che prima c'era nel corpo del thread.

Il server è più semplice: crea il socket, fa l'accept() e poi inizializza il thread; quindi cicla e torna all'accept().

SERIALIZZAZIONE IN JAVA

Nell'esempio precedente, le informazioni relative a un conto (numero, nome e cognome, saldo) erano memorizzate nel file e trasmesse sul socket separatamente

Tuttavia, in generale è più naturale gestire tali informazioni con un oggetto:

```
public class Account implements java.io.Serializable {
      private int account;
      private String first;
      private String last;
      private double balance;
      public Account (int account, String first, String last, double
      balance) {
            this.account = account;
            this.first = first;
            this.last = last;
            this.balance = balance;
      }
public int getAccount() {return account;}
public void setAccount(int v) {this.account = v;}
… …
}
```

Affinché un oggetto possa essere serializzato, esso deve implementare l'interfaccia **java.io.Serializable**

E'una tagging interface, non fa niente, ma Java sa che la classe può essere appiattita.

Le classi ObjectInputStream e ObjectOutputStream consentono la lettura e scrittura di oggetti serializzati

La serializzazione esegue una **deep copy** dell'oggetto (object closure): l'immagine serializzata contiene tutti gli oggetti contenuti nell'oggetto iniziale, gli oggetti contenuti in tali oggetti, e così via ricorsivamente

Gli attributi dichiarati come **transient** non vengono serializzati

Un account è rappresentato da una classe Account.

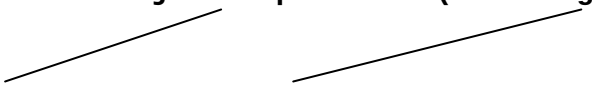
Questa ha due metodi: getAccount e setAccount per leggere e scrivere i valori

IL SERVER

```
try {
      server = new ServerSocket(port);
      while(true) {
      client = server.accept();
      is = new ObjectInputStream(client.getInputStream());
      os = new ObjectOutputStream(client.getOutputStream());
      ObjectInputStream dataStream =
            new ObjectInputStream(new FileInputStream(DATAFILE));
      int command = -1;
      while(command != DONE) {
            command = is.readShort();
            switch(command) {
                  case NEXT:
                  os.writeObject(dataStream.readObject());
                  break;
```

```
                case DONE:
                dataStream.close();
                is.close();os.flush();os.close();client.close();
                break;
                }
            }
      } catch(IOException ioe) { ioe.printStackTrace();
}

os = new ObjectOutputStream(client.getOutputStream());
```



scrivo l’oggetto che ho letto

SOCKET SU UDP

Le classi **Socket** e **ServerSocket** implementano socket connection-oriented su TCP

Forniscono l’illusione di un canale di comunicazione dedicato e affidabile: ritrasmissione e ordinamento dei pacchetti sono gestiti dal livello di trasporto

La classe **DatagramSocket** implementa socket connectionless su UDP

Fornisce solo la possibilità di inviare e ricevere un pacchetto: ritrasmissione e ordinamento sono a carico del programmatore

La classe **MulticastSocket** implementa socket multicast su UDP

Estende DatagramSocket con la possibilità di inviare lo stesso pacchetto a un gruppo di riceventi anziché a uno solo

MIDDLEWARE TRADIZIONALI

I sistemi distribuiti rivestono importanza sempre crescente.

La realizzazione di un sistema distribuito è assai più complessa di quella di un sistema tradizionale.

Nonostante ciò, le applicazioni distribuite sono spesso sviluppate usando le astrazioni fornite dal sistema operativo (di basso livello, pongono gran parte del peso dello sviluppo sulle spalle del programmatore)

Esiste la necessità di uno strato intermedio fra sistema operativo e applicazioni, che innalzi il livello di astrazione della programmazione distribuita

- Fornendo astrazioni appropriate, che nascondano la complessità degli strati sottostanti
- Liberando il programmatore da compiti ripetitivi e automatizzabili
- Migliorando la qualità del software mediante il riuso di soluzioni consolidate, corrette, ed efficienti Il **middleware** fornisce precisamente tale strato intermedio

DEFINIZIONE

Insieme di servizi trasversali che permettono ai componenti di un applicazione distribuita di interagire sfruttando una rete di comunicazione.

REQUISITI

Comunicazione

- Come servizio di base, un middleware deve fornire a componenti distribuiti la possibilità di comunicare
- Marshalling e serializzazione
- I livelli di sessione e presentazione sono gestiti dal middleware, non dal programmatore

Coordinazione

- Primitive di sincronizzazione
- Primitive di comunicazione sincrone e asincrone
- Astrazioni di alto livello per la comunicazione multi-punto (es., eventi)

Affidabilità

- Garanzie circa la ricezione di un messaggio (best effort, at most once, at least once, exactly once), l'ordinamento dei messaggi, atomicità delle transazioni, …
- Gestita dal middleware, non dal programmatore

Scalabilità

- Tipicamente ottenuta attraverso la "trasparenza" … (… degli accessi, della locazione, della migrazione, della replicazione)

Eterogeneità

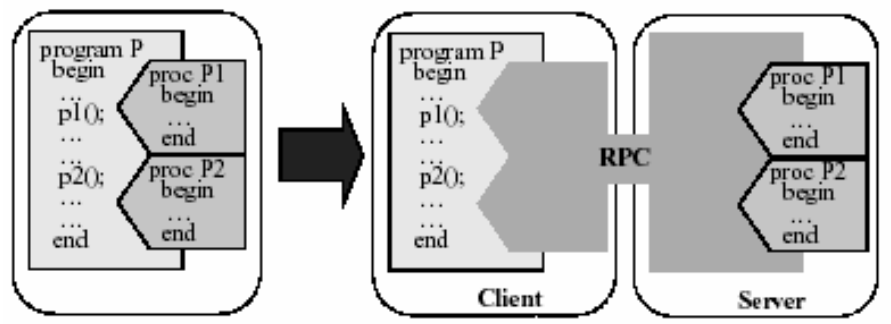
- I sistemi distribuiti sono costituiti da piattaforme hardware e software eterogenee, ciascuna con i suoi linguaggi, protocolli, formati, convenzioni (es., littleendian vs. bigendian)
- Interoperabilità fra middleware

REMOTE PROCEDURE CALL

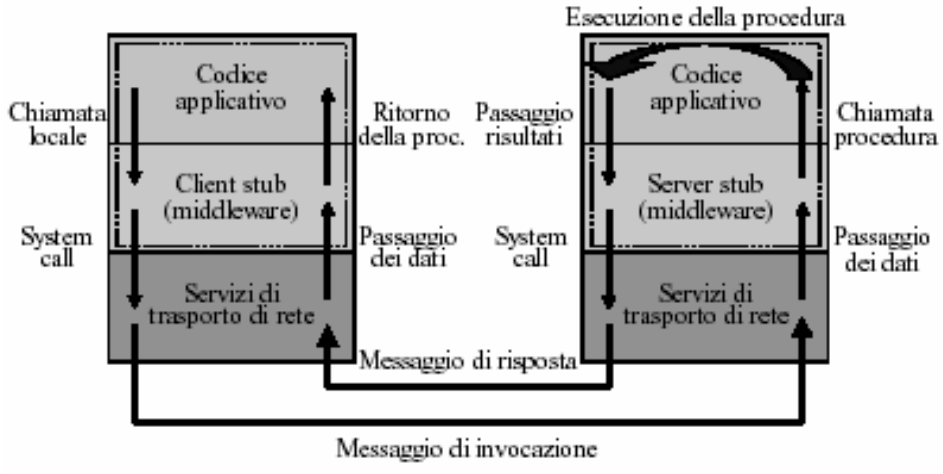
Fu inventata da Sun Microsystems nei primi anni ’80

Problema: la gestione dell’interazione tra client e server avviene primitive I/ O del SO, il livello di astrazione è troppo basso.

Idea: fornire al programmatore un meccanismo per accedere a risorse remote attraverso la comune chiamata a procedura, nascondendo tutti i dettagli relativi al setup della connessione, al marshalling dei parametri, ed alla eterogeneità delle piattaforme.



FUNZIONAMENTO



Quando il client fa una chiamata locale, l’esecuzione di questa è legata al client stub. Questo ha il codice per prelevare i parametri, appiattirli, aprire i socket… tutto in automatico.

Viene generato un messaggio con i dati per l’invocazione.

Viene preso dal server stub che ha le operazioni inverse e lo passa al codice applicativo.

Esegue il codice, e i risultati fanno il giro contrario

Dal punto di vista concettuale è come se l’invocazione avvenisse in locale.

MARSHALLING

Il marshalling risolve due problemi:

- Conversione dei parametri in stream di byte.
  I parametri delle procedure devono essere impacchettati secondo un formato che sia compreso da client e server e che sia gestibile direttamente dallo strato di trasporto. Quindi, strutture dati complesse (e.g., struct) devono essere ridotte a sequenze di byte. Spesso chiamata serializzazione o pickling.

- Conversione fra dati appartenenti a piattaforme eterogenee.
  Macchine diverse hanno rappresentazioni diverse dei dati (interi in complemento a 1 o a 2, rappresentazione little endian e big endian)

Il codice che esegue marshalling e unmarshalling viene generato automaticamente a partire dalla specifica delle interfacce delle procedure, descritta in un Interface Definition Language (IDL), e fa parte degli stub

BINDING DINAMICO DEL SERVIZIO

Problema: compilare staticamente il nome del server host nel codice sarebbe altamente indesiderabile

Soluzione: introdurre un processo (demone) addizionale sul server, il binder (solitamente chiamato portmap) che si occupa di legare le chiamate alle procedure:

- Il server registra le proprie procedure presso il demone

- I client possono contattare un dato demone e verificare che un servizio sia supportato, o richiedere la lista dei servizi

- I client possono anche effettuare una invocazione in broadcast su più demoni: solo il server che possiede il servizio si incaricherà della risposta

Il binding dinamico permette di ottenere la trasparenza della locazione

ATTIVAZIONE DINAMICA DEL SERVIZIO

Problema: l'esistenza di server perennemente attivi può determinare uno spreco di risorse nel caso in cui i servizi forniti siano usati solo saltuariamente

Soluzione: introdurre un processo (demone) addizionale sul server, solitamente chiamato inetd, che si occupa di rispondere alle richieste dei client attivando il server corrispondente e passandogli la richiesta

Ovviamente, la prima richiesta verrà servita in maniera meno efficiente

SEMANTICA DI INVOCAZIONE IN PRESENZA DI MALFUNZIONAMENTI

1. Il client non riesce a trovare un server.

Tipicamente viene ritornato un errore o eccezione.

2. Il messaggio di richiesta è andato perso.

Tipicamente gestito mediante timer.

3. Il messaggio di risposta è andato perso.

Il problema è determinare se la risposta non arriva perché è andata persa o la richiesta non è mai arrivata. Si usano timers e sequence numbers. Semplice da gestire se l'operazione è idempotente.

4. Crash del server.

Esistono quattro semantiche possibili: exactly once, at least once, at most once, best effort. Exactly once in generale è impossibile da garantire (con best effort, non do garanzie).

5. Crash del client.

Viene creata una computazione orfana sul server. Sistemi diversi adottano politiche diverse (es. extermination, reincarnation, …)
Vale il principio dell'im possibilità del consenso distribuito.

ALTRE CONSIDERAZIONI

Client e server stub

Possono essere generati automaticamente, oppure scritti dal programmatore sfruttando una serie di servizi di base. Possono essere generati staticamente o dinamicamente

Passaggio parametri

Può essere per valore (semplice), oppure per riferimento (complesso, occorre mantenere riferimenti di rete)

Semantica di chiamata

Tipicamente sincrona, raramente asincrona

Protocollo di trasporto

TCP, UDP, altri protocolli

Sicurezza
Esistono implementazioni "sicure" di RPC

SUN RPC

Sun Microsystems fornisce una implementazione di RPC che si è imposta come standard su Internet

Usa XDR (eXternal Data Representation) per rappresentare le strutture dati in maniera indipendente dalla piattaforma

Può usare TCP o UDP come meccanismo di trasporto

I parametri possono essere passati solo per copia, ed è possibile specificare un solo argomento e un solo valore di ritorno

È obbligatorio l'uso del binding dinamico

È possibile specificare tre livelli di sicurezza: nessuna sicurezza, diritti di accesso alla Unix, crittografia DES

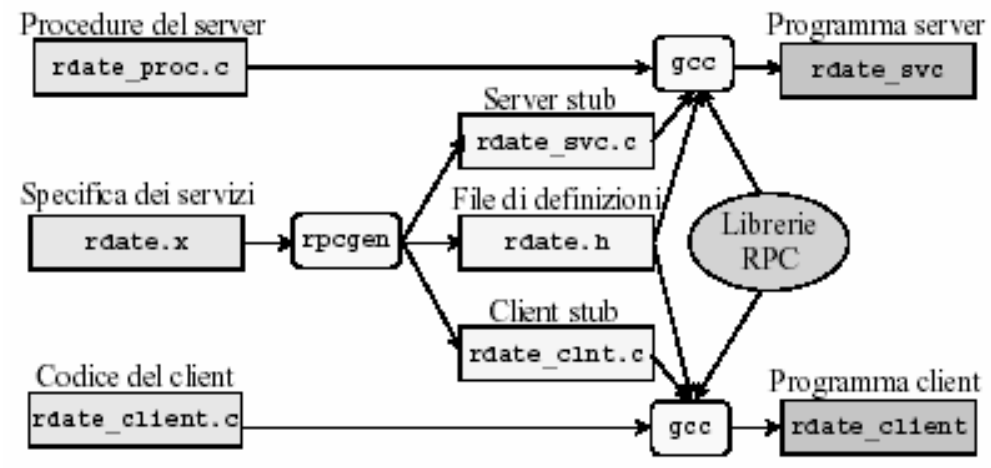
SUN RPC: CICLO DI SVILUPPO

Lo sviluppatore del server:

- Scrive le procedure del server
- Scrive un file che definisce il prototipo delle funzioni esportate (numero di programma e versione)
- Usa il compilatore rpcgenche a partire dai file di cui sopra genera gli stub del client e del server e un file di definizioni

Lo sviluppatore del client:

- Scrive il codice del client includendo il file di definizioni
- Compila il codice insieme al codice dello stub corrispondente



MIDDLEWARE AD OGGETTI

L'idea è quella di RPC: supportare gli aspetti di comunicazione sollevando il programmatore dalla maggior parte della gestione della complessità della distribuzione e fornendogli primitive il più possibile simili a quelle usate in locale.

In questo caso, le primitive forniscono le astrazioni del modello OO, in modo da mantenere i suoi vantaggi anche nel distribuito

Le applicazioni sono costituite da oggetti che risiedono su macchine diverse e comunicano mediante invocazione remota di metodi, una specie di RPC fra oggetti

Alcuni dei meccanismi sono simili a quelli di RPC: IDL, stub, binding dinamico

Spesso, vengono addirittura riusate implementazioni di RPC per l'implementazione dei meccanismi di invocazione OO

Tecnologie: OMG CORBA, Microsoft COM+, Sun Java/RMI

INTERFACE DEFINITION LANGUAGE

Per il middleware ad oggetti l'IDL è ancora più importante, vediamo alcuni motivi:

- Un fatto: In ambito distribuito i principi dell'information hiding e della compilazione separata diventano essenziali
- La sua spiegazione: non è ipotizzabile che sia nota l'implementazione dei servizi da invocare
- La soluzione: ogni oggetto deve conoscere solo l'interfaccia degli oggetti con cui comunica, cioè l'elenco dei metodi che è possibile invocare. Tali interfacce sono specificate mediante un interface definition language

Nel middleware OO l'IDL assume un ruolo ancora maggiore

- La separazione tra interfaccia e implementazione è uno dei cardini dell'object-orientation
- La definizione dell'IDL di fatto vincola il modello a oggetti che viene implementato dal middleware (es., ereditarietà singola o multipla)

Gli IDL per middleware OO sono più espressivi

- In RPC, a una procedura non è permesso ritornare un'altra procedura, mentre un oggetto server può ritornarne un altro
- Gestione delle eccezioni
- Ereditarietà

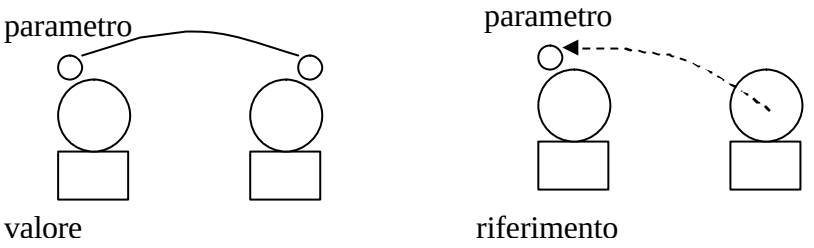
JAVA/RMI

Java RMI è in pratica una trasposizione in ambito object-oriented di RPC, viene infatti fornita la **possibilità di invocare un metodo su un oggetto remoto** come se fosse locale.

Gli aspetti innovativi sono:

- La possibilità di avere passaggio parametri sia per valore che per riferimento

  In ambito distribuito, il passaggio per copia implica la spedizione sulla rete



  In genere conviene passare gli oggetti per riferimento e i tipi per copia.

- Il download automatico delle classi necessarie a un'invocazione remota

INTERFACCE

In RMI manca completamente la nozione di IDL, o meglio: l'IDL è costituito dal linguaggio Java

In Java esiste già una forte distinzione tra classi e interfacce: l'interfaccia "alla CORBA" di un oggetto distribuito è specificata con una normale interfaccia Java.

Tuttavia, esistono dei **vincoli**:

- Le interfacce di oggetti distribuiti devono ereditare tutte da **java.rmi.Remote**
- I metodi definiti in tali interfacce devono dichiarare tutti l'eccezione **java.rmi.RemoteException**

Un oggetto che implementa l'interfaccia *Remote* viene chiamato **oggetto remoto** e le sue operazioni vengono accedute remotamente.

Grazie all'ereditarietà multipla fra interfacce, metodi definiti in un'interfaccia MyInterface che non era stata pensata remota, possono essere forniti in maniera remota creando una nuova interfaccia che eredita da MyInterface e Remote

IMPLEMENTAZIONE DI OGGETTI DISTRIBUITI

I vincoli sulle interfacce si ripercuotono sull'implementazione degli **oggetti remoti**, che devono:

- implementare **java.rmi.Remote** o un suo sottotipo
- estendere la classe **java.rmi.RemoteObject** o un suo sottotipo (tipicamente **UnicastRemoteObject** che ha la ridefinizione dei metodi necessari all'invocazione remota)
- dichiarare l'eccezione **java.rmi.RemoteException** per tutti i costruttori (per poter riportare al client le eccezioni sollevate)

Per gli oggetti che non sono remoti, ma che vengono comunque **passati come parametro** (attuale o di ritorno) in un'invocazione remota, l'unico vincolo è quello di essere serializzabili, cioè di implementare **java.io.Serializable**

OTTENERE UN RIFERIMENTO AL SERVER

In locale devo avere una variabile riferimento all'oggetto.

Il client può ottenere un riferimento all'oggetto remoto server

- come parametro passato o ritornato da altri oggetti
- ottenuto accedendo allo **rmiregistry** (fa le veci del binder per rpc, risiede sul server e associa nomi simbolici ad oggetti)

Il client può invocare tutti i metodi elencati nell'interfaccia remota implementata dal server.
Per il client non occorre alcun processo di compilazione speciale.

PASSAGGIO PARAMETRI IN RMI

La semantica di invocazione nel caso remoto è diversa dal caso locale

Data un'invocazione di metodo **m(obj)**:

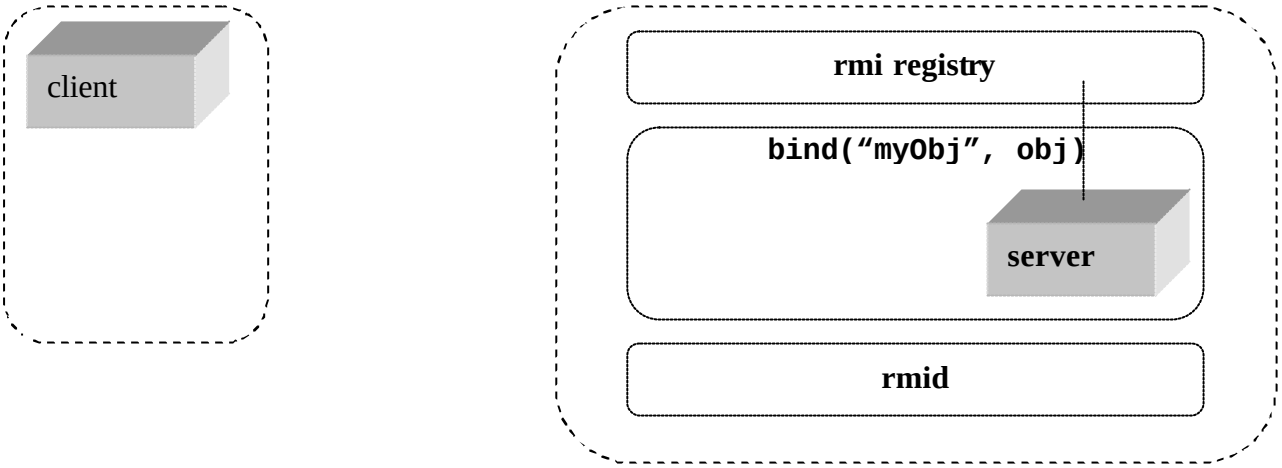
- Se *obj* è un oggetto remoto, la normale semantica di invocazione viene mantenuta; eventuali modifiche effettuate in m allo stato di *obj* avverranno sfruttando riferimenti di rete
- Se *obj* non è un oggetto remoto, *m* opera in realtà su una copia serializzata dell'oggetto originale *obj*; eventuali modifiche effettuate in *m* allo stato di *obj* avranno effetto solo sulla copia locale e non sull'originale remoto

In pratica, il passaggio parametri di oggetti non-remoti è per valore, mentre quello di oggetti remoti è per riferimento.

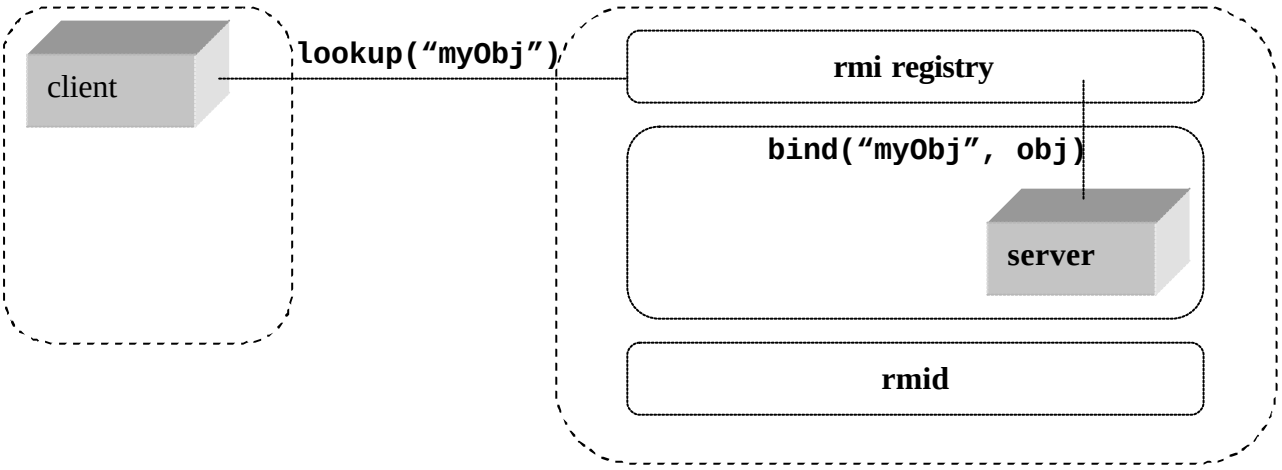
Lo stesso vale per i parametri di ritorno.

ARCHITETTURA DI RMI

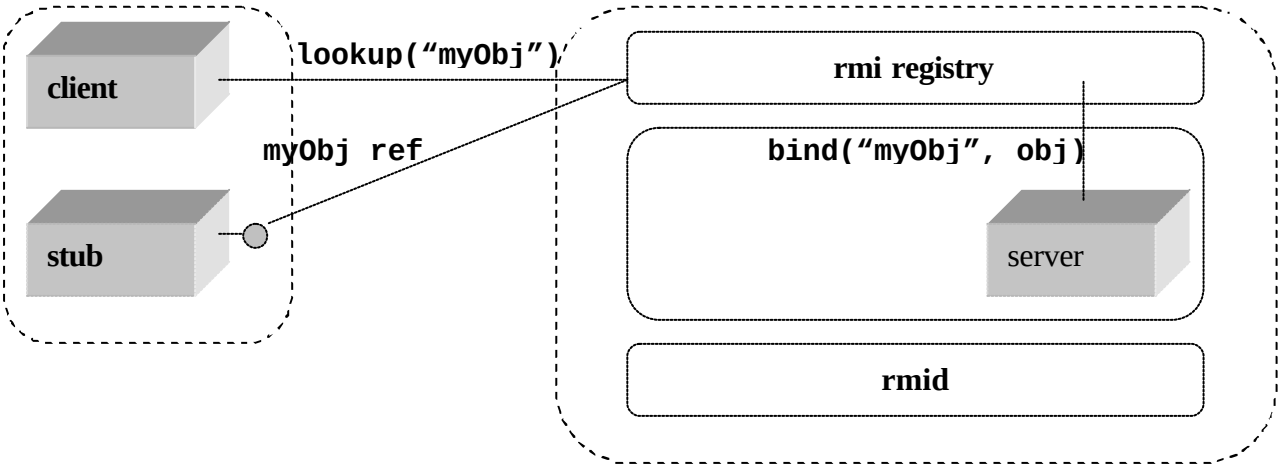
Il server registra l'oggetto applicativo che deve essere esportato mediante *bind*



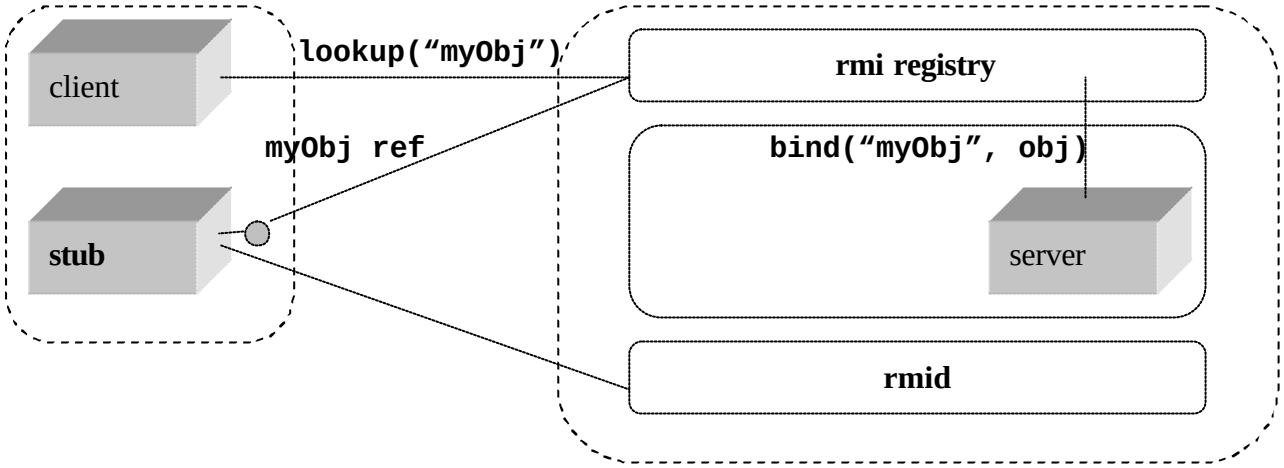
Il client deve fare il *lookup* passando il nome simbolico



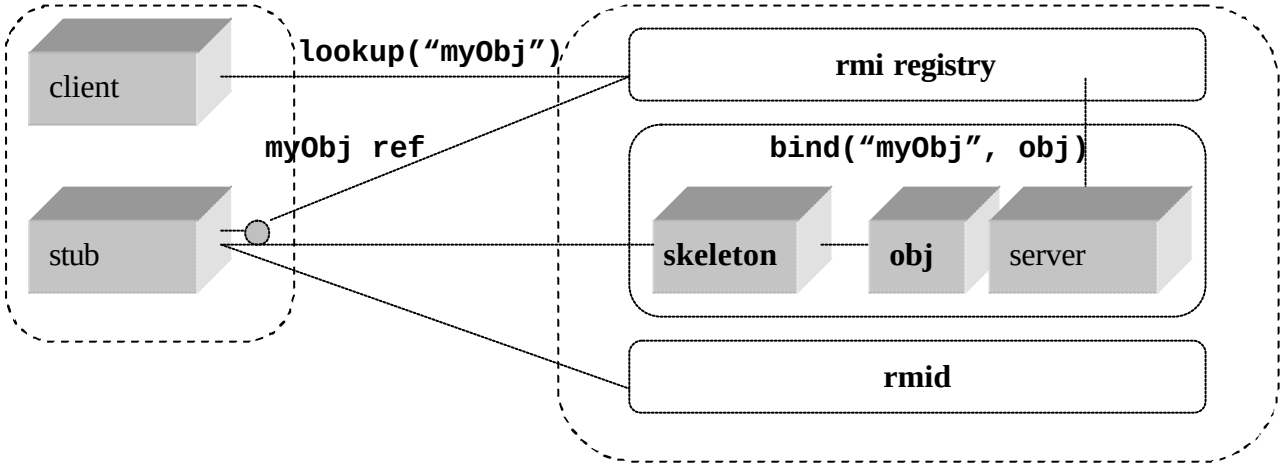
Riceve in ritorno l'equivalente di un riferimento ad un oggetto
Lo stub ha il codice che gestisce la comunicazione



Il client ha un riferimento remoto all'oggetto e può fare un invocazione di metodo passando per lo stub



Con rmid viene creato l'oggetto obj e questo viene masso in comunicazione con lo skeleton che rappresenta il client sul lato server.



NAMING: RMIREGISTRY

rmiregistry fornisce un servizio di directory per RMI

Il servizio non è distribuito; inoltre, per motivi di sicurezza non è possibile registrare un oggetto su un registry in esecuzione su una macchina diversa da quella dove risiede l'oggetto.

Il client deve sapere dov'è il server.

Un oggetto server RMI si registra presso un registry specificando un nome simbolico

Un client RMI può:

- ottenere un riferimento ad un server RMI indicandone il nome simbolico
- chiedere la lista dei nomi simbolici disponibili presso il registry Tali servizi sono realizzati attraverso le classi:
  - **java.rmi.Naming**
  - **java.rmi.registry.LocateRegistry**
  - **java.rmi.registry.Registry**

Sono inoltre fornite classi per invocare il registry da programma, e per creare registry diversi da quello fornito da Sun.

Il client può anche ottenere il riferimento al server in altri modi, es. come parametro di ritorno di qualche chiamata locale

ESEMPIO

SERVER

```
public interface AccountServer extends java.rmi.Remote {
  Account getAccount(int number) throws RemoteException;
}

public class AccountServerImpl
 extends java.rmi.server.UnicastRemoteObject
 implements AccountServer {
 private Hashtable accounts;
 …
 public static void main(String[] args) {
  …
  try {
   Naming.rebind("//localhost:"+ String.valueOf(port) +
                 "/AccountServer", new AccountServerImpl());
 } catch(java.net.MalformedURLException mue) { … }
 } catch(RemoteException re) { … }
 }
 public AccountServerImpl() throws RemoteException { … }

 public Account getAccount(int number)
        throws RemoteException {
    return (Account) accounts.get(new Integer(number));
 }
}
```

L'interfaccia stabilisce il contratto tra client e server, cosa il client può invocare sul chiamato.

Il client sa che il server implementa l'interfaccia AccountServer.

L'interfaccia estende da Remote (è un oggetto remoto)

Sotto c'è l'implementazione dell'interfaccia (nel client ho solo l'interfaccia e non l'implementazione)

AccountServerImpl estende UnicastRemoteObject

Hashtable è un array associativo, ho un accoppiamento tra chiave e un contenuto. Dando la chiave, mi restituisce l'oggetto associato.

```
Naming.rebind("//localhost:"+ String.valueOf(port) +
              "/AccountServer", new AccountServerImpl());
```

passa un nome simbolico all'host locale e l'oggetto associato al nome locale

ho poi l'implementazione del metodo getAccount che ritorna l'oggetto associato all'intero nella hash table accounts.

CLIENT

```
…
try {
 accountServer = (AccountServer) Naming.lookup("//" +
  host + ":" + String.valueOf(port) +
  "/AccountServer");
} catch(NotBoundException nbe) { … }
} catch(RemoteException re) { … }
} catch(java.net.MalformedURLException mue) { … }
}
…
Account a = null;
try {
 a = accountServer.getAccount(current++);
} catch (RemoteException re) { … }
…
```

Devo prendere un riferimento ad un oggetto remoto.

Contatto il registry

```
accountServer = (AccountServer) Naming.lookup("//" host + ":" +
String.valueOf(port) +  "/AccountServer");
```

viene tornato all'indietro un oggetto di tipo object che va "castato" al tipo che mi aspetto nell'applicazione.

da questo punto in avanti lo vedo come un oggetto locale.

La differenza rispetto ad una chiamata di metodo locale è che devo inserire un blocco try/catch

```
try {
   a = accountServer.getAccount(current++);
} catch (RemoteException re) { … }
```

SVILUPPO

javac AccountClient.java

- Si compila il client, senza accortezze particolari

javac AccountServerImpl.java

- Si compila il server; solitamente il server inizializza anche un RMISecurityManager, per il quale vanno specificate politiche di sicurezza opportune

rmic AccountServerImpl

- Si creano le classi stub e skeleton con il compilatore rmic; si noti che rmic compila il bytecode e non il sorgente

Si lancia rmiregistry ed eventualmente rmid

Attenzione al CLASSPATH! Le classi del server devono essere visibili al rmiregistry (in una directory del CLASSPATH o nella directory da cui è stato lanciato rmiregistry)

Si lanciano server e client

DEPLOY

Quando facciamo il deployment:

Client

AccountServer.class (interfaccia)
AccountClient.class
?_stub.class

Server

AccountServer.class
AccountServerImpl.class
?_skel.class

RUN

Lanciare il registry (deve essere in grado di trovare le classi, queste devono essere nel classpath)

Lanciare Il server

Lanciare il client