



Comunicazione nei Sistemi Distribuiti

Parte 1

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2019/20

Valeria Cardellini

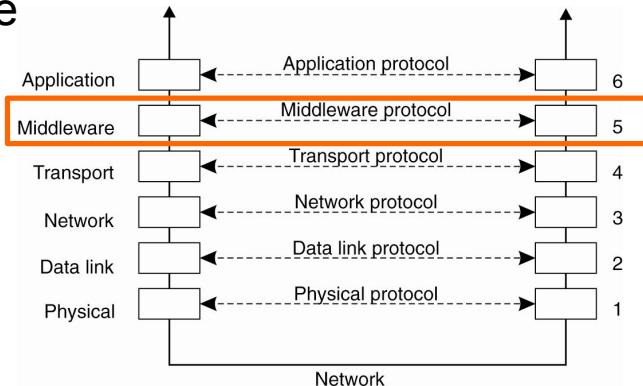
Laurea Magistrale in Ingegneria Informatica

Comunicazione nei SD

- Basata sullo scambio di messaggi
 - Invio e ricezione di messaggi (a basso livello)
- Per permettere lo scambio di messaggi, le parti devono accordarsi su diversi aspetti
 - Quanti volt per segnalare un bit 0 e quanti per un bit 1?
 - Quanti bit per un intero?
 - Come fa il destinatario a sapere quale è l'ultimo bit del messaggio?
 - Come può capire se un messaggio è stato danneggiato e cosa fare quando se ne accorge?
 - ...

Protocolli a livelli

- Soluzione (già nota): suddividere il problema in livelli
 - Ciascun livello in un sistema comunica con lo stesso livello nell'altro sistema
 - Modello di riferimento ISO/OSI
- Adattamento del modello di riferimento tradizionale per le comunicazioni di rete
 - Protocolli di livello più basso
 - Protocolli di trasporto
 - Protocolli middleware
 - Protocolli applicativi



Protocolli middleware

- Livello **middleware**: fornisce servizi comuni e protocolli general-purpose
 - di alto livello
 - indipendenti dalle specifiche applicazioni
 - che possono essere usati da altre applicazioni
- Alcuni esempi:
 - Protocolli di **comunicazione**: per invocare procedure remote o metodi remoti, accodare messaggi, supportare streaming e multicasting
 - Protocolli di **naming**: per permettere la condivisione di risorse tra applicazioni
 - Protocolli di **sicurezza**: per consentire alle applicazioni di comunicare in modo sicuro
 - Protocolli di **consenso distribuito**, tra cui **commit distribuito** (per stabilire che una transazione sia portata a termine da tutte le parti coinvolte o non abbia alcun effetto)
 - Protocolli di **locking distribuiti**
 - Protocolli di **consistenza dei dati**

Obiettivo di queste lezioni

Tipi di comunicazione

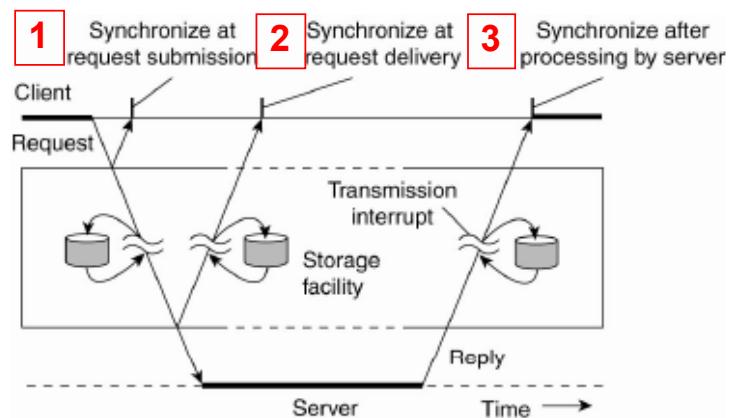
- Distinguiamo la comunicazione in base a:
 - **Persistenza**
 - Comunicazione persistente o transiente
 - **Sincronizzazione**
 - Comunicazione sincrona o asincrona
 - **Dipendenza dal tempo**
 - Comunicazione discreta o streaming

Comunicazione persistente o transiente

- Comunicazione **persistente**
 - Il messaggio immesso viene memorizzato dal middleware di comunicazione per tutto il tempo necessario alla consegna
 - Non occorre che il mittente continui l'esecuzione dopo l'invio del messaggio
 - Non è necessario che il destinatario sia in esecuzione quando il messaggio è inviato
- Comunicazione **transiente**
 - Il messaggio è memorizzato dal middleware solo finché mittente e destinatario sono in esecuzione
 - Se la consegna non è possibile, il messaggio viene cancellato
 - Caso tipico della comunicazione di rete a livello di trasporto: i router memorizzano ed inoltrano, ma cancellano se non è possibile inoltrare

Comunicazione sincrona

- Comunicazione sincrona
 - Una volta sottoposto il messaggio, il mittente si blocca finché l'operazione non è completata
 - L'**invio** e la **ricezione** sono operazioni **bloccanti**
 - Fino a quando si blocca il mittente? Varie alternative:
 1. finché il middleware non prende il controllo della trasmissione
 2. finché il messaggio non viene ricevuto dal destinatario
 3. finché il messaggio non viene elaborato dal destinatario



Comunicazione asincrona

- Comunicazione asincrona
 - Una volta sottomesso il messaggio, il mittente continua l'elaborazione: il messaggio viene memorizzato temporaneamente dal middleware fino ad avvenuta trasmissione
 - L'**invio** è un'operazione **non bloccante**
 - La **ricezione** può essere **bloccante** o **non bloccante**

Comunicazione discreta o streaming

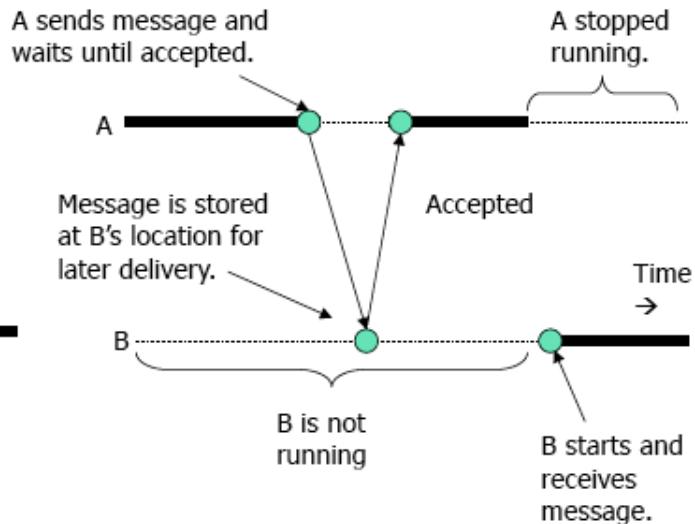
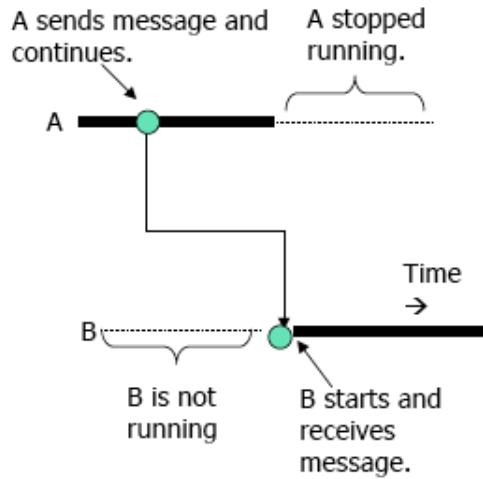
- Comunicazione **discreta**
 - Ogni messaggio costituisce un'unità di informazione completa
- Comunicazione **a stream (streaming)**
 - Comporta l'invio di molti messaggi, in relazione temporale tra loro o in base all'ordine di invio, che servono a ricostruire un'informazione completa

Combinazione dei tipi di comunicazione

- Combinazioni tra persistenza e sincronizzazione
 - a) Comunicazione **persistente e asincrona**
 - E.g., email
 - b) Comunicazione **persistente e sincrona**
 - Mittente bloccato fino alla copia (garantita) del messaggio presso il destinatario
 - c) Comunicazione **transiente e asincrona**
 - Mittente non attende ma messaggio perso se destinatario non raggiungibile (e.g., UDP)
 - Comunicazione **transiente e sincrona**
 - d) mittente bloccato fino alla copia (non garantita) del messaggio nello spazio del destinatario (e.g., RPC asincrona)
 - e) mittente bloccato fino alla consegna del messaggio al destinatario
 - f) mittente bloccato fino alla ricezione di un messaggio di risposta dal destinatario (e.g., RPC sincrona e RMI)

Persistent communication

- a) Persistent asynchronous communication
- b) Persistent synchronous communication

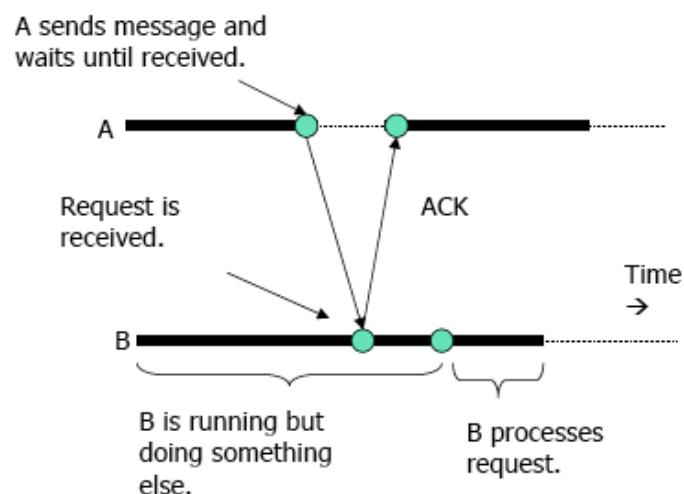
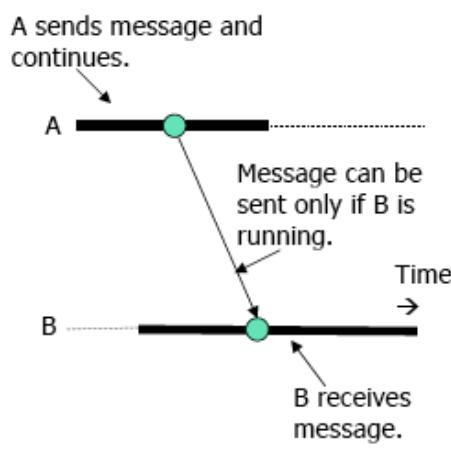


(a) Persistent asynchronous communication

(b) Persistent synchronous communication

Transient communication

- c) Transient asynchronous communication
- d) Receipt-based synchronous communication

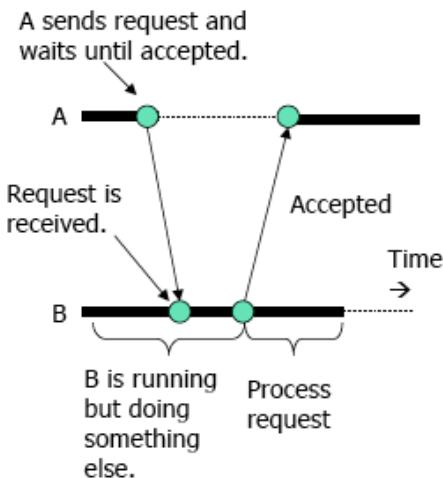


(c) Asynchronous communication

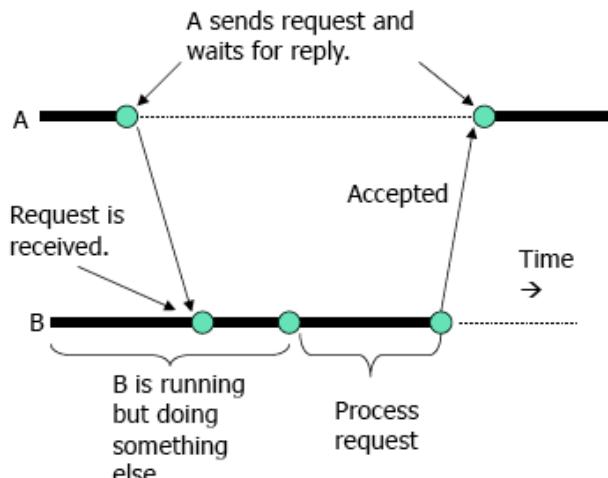
(d) Receipt-based synchronous communication

Transient communication

- e) Delivery-based synchronous communication
- f) Response-based synchronous communication



(e) Delivery-based synchronous communication



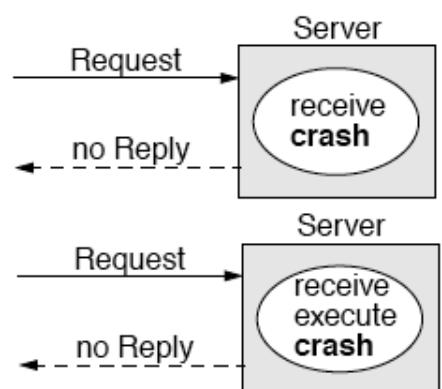
(f) Response-based synchronous communication

Semantica della comunicazione ed errori

- Diverse tipologie di failure nella comunicazione tra client e server
- 1. Il messaggio di richiesta e/o risposta può essere perso o ritardato, oppure la connessione resettata
 - La rete è affidabile (“The Eight Fallacies of Distributed Computing”)

1. Il server può subire un crash
 - a. prima di eseguire il servizio richiesto
 - b. dopo aver eseguito il servizio richiesto

Il client non può distinguere tra a e b
1. Il client può subire un crash



Semantica della comunicazione ed errori

- In un sistema distribuito quale è la semantica della comunicazione *in presenza di errori*?
 - Semantica **may-be**
 - Semantica **at-least-once**
 - Semantica **at-most-once**
 - Semantica **exactly-once**
- Si applica sia al processamento di servizi (es. RPC), sia al delivery di messaggi (es. MOM)
 - Consideriamo il processamento di un servizio

Meccanismi di base

- La semantica della comunicazione dipende dalla combinazione di tre meccanismi di base
1. Lato client: riprova (**Request Retry – RR1**)
 - Il client continua a provare finché ottiene risposta oppure è certo del guasto del server dopo un certo numero di retry senza risposta
 2. Lato server: filtra i duplicati (**Duplicate Filtering – DF**)
 - Il server scarta gli eventuali duplicati di richieste provenienti dallo stesso client
 3. Lato server: ritrasmetti le risposte (**Result Retransmit – RR2**)
 - Il server conserva le risposte per poterle ritrasmettere senza ricalcolarle nel caso riceva una richiesta duplicata
 - Meccanismo necessario se l'operazione eseguita dal server non è *idempotente*
 - Operazione **idempotente** (i.e., priva di effetti collaterali): molteplici esecuzioni dell'operazione producono gli stessi effetti/risultati di una sola esecuzione dell'operazione, ad es. operazione read-only oppure $x=1$

Semantica may-be

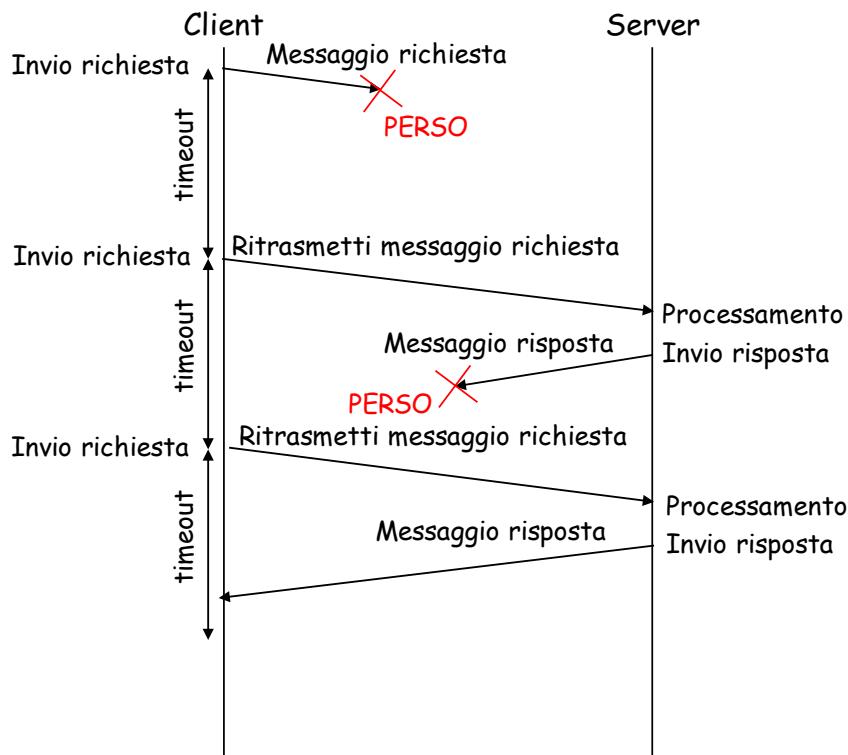
- **Semantica may-be (forse)**
 - Il servizio sul server può essere stato eseguito o meno
 - Non si attuano azioni per garantire l'affidabilità della comunicazione: nessun meccanismo (RR1, DF, RR2) in uso
 - Ad es. best-effort in UDP



Semantica at-least-once

- **Semantica at-least-once (almeno una volta)**
 - Il servizio, se eseguito, è stata eseguito *almeno* una volta
 - Anche più volte, a causa della duplicazione dovuta alle ritrasmissioni
 - Il client usa RR1, ma il server non usa né DF né RR2
 - Il server non si accorge della ritrasmissione del messaggio di richiesta e conseguente duplicazione
 - Adatta per **servizi idempotenti** (server stateless)
 - All'arrivo di una risposta, il client non sa quante volte sia stata processata dal server (*almeno una*): non conosce lo stato del server
 - Il server può aver eseguito il servizio richiesto ma subire un crash prima di inviare la risposta: allo scadere del timeout il client invia la stessa richiesta ed il server esegue nuovamente il servizio ed invia la risposta al client

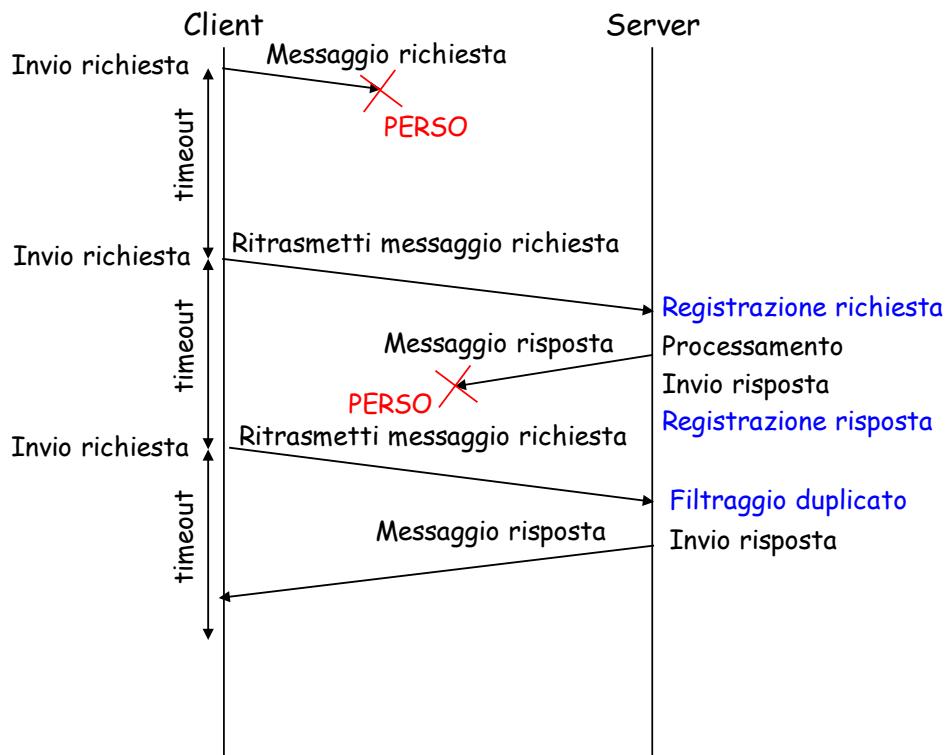
Semantica at-least-once



Semantica at-most-once

- **Semantica at-most-once (al più una volta)**
 - Il servizio, se eseguito, è stata eseguito *al più* una volta
 - Il client sa che, se riceve la risposta, essa è stata calcolata dal server una sola volta
 - In caso di insuccesso, nessuna informazione (*at-most-once*: la risposta è stata calcolata al più una volta, ma possibilmente anche nessuna)
 - Tutti e 3 i meccanismi in uso (RR1, DF, RR2)
 - Il client effettua ritrasmissioni
 - Il server mantiene uno **stato** per riconoscere messaggi di richiesta già ricevuti e non eseguire il servizio più di una volta
 - Adatta per qualunque tipo di servizio
 - Anche **non idempotente**
 - Semantica che non mette vincoli sulle azioni conseguenti
 - No coordinamento tra client e server: in caso di errore, il client non sa se il server ha eseguito il servizio, mentre il server ignora se il client sa che ha eseguito il servizio
 - Possibile inconsistenza sull'accordo tra client e server

Semantica at-most-once



At-most-once semantics: implementation

- Server detects duplicate requests and returns previous reply instead of re-running operation handler()
- How to detect a duplicate request?
 - Client includes a unique ID (xid) with each request and uses same xid when re-sending
- Some at-most-once complexities
 - How to ensure xid is unique?
 - Server must eventually discard info about old requests: when is discard safe?
 - Can use sliding windows and sequence numbers
 - Can discard information older than maximal message lifetime
 - How to handle duplicate requests while original one is still executing?

```
Server:  
if seen[xid]  
    r = old[xid]  
else  
    r = handler()  
    old[xid] = r  
    seen[xid] = true
```

Semantica exactly-once

- **Semantica exactly-once (esattamente una volta)**
 - Garanzie migliori ma più difficili da attuare in un SD, soprattutto a larga scala
 - Richiede accordo completo sull'interazione
 - Il servizio è eseguito una sola volta oppure non è eseguito: **semantica tutto o niente**
 - Se va tutto bene: il servizio viene eseguito una sola volta, riconoscendo i duplicati
 - Se qualcosa va male: client o server sanno se il servizio è eseguito (una sola volta - *tutto*) o se non è stato eseguito (nessuna volta - *niente*)
 - Semantica con conoscenza concorde dello stato dell'altro e senza ipotesi sulla durata massima del protocollo di interazione tra client e server
 - Mancanza di vincoli sulla durata massima: poco praticabile in un sistema reale!

Exactly-once semantics: mechanisms

- Server-side basic mechanisms (RR1, DF, RR2) are not enough
- Additional mechanisms are required to tolerate server-side faults
 - **Transparent server replication**
 - **Write-ahead logging (WAL)**
 - Changes must be written only after they have been logged that is, after log records describing the changes have been flushed to permanent storage
 - **Recovery**
 - Mechanisms to recover from whatever state the failed server left behind and begin processing from a safe point
 - We will study distributed snapshot and state checkpointing

Summing up the failure semantics

- At-least once and at-most once semantics are feasible and widely used in distributed systems
- We often choose the lesser of two evils, which is at-least-once semantics in most cases
 - At-least once semantics is also easier to scale

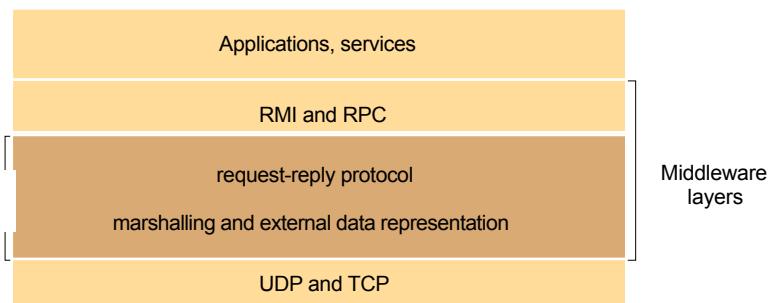
Distributed systems are all about trade-offs!

Programmazione di applicazioni di rete

- Programmazione di rete *esplicita*
 - Chiamata diretta all'API socket e scambio esplicito di messaggi
 - Usata nella maggior parte delle applicazioni di rete (ad es. Web browser, Web server)
 - La distribuzione non è trasparente
 - Gran parte del peso dello sviluppo sulle spalle del programmatore
- Come innalzare il livello di astrazione della programmazione distribuita? Fornendo uno strato intermedio (middleware) fra SO ed applicazioni che:
 - Nasconde la complessità degli strati sottostanti
 - Liberi il programmatore da compiti automatizzabili
 - Migliori la qualità del software mediante il riuso di soluzioni consolidate, corrette ed efficienti

Programmazione di applicazioni di rete

- Programmazione di rete *implicita*
 - **Chiamata di procedura remota (RPC)**
 - L'applicazione distribuita è realizzata usando le chiamate di procedura, ma il processo chiamante (client) ed il processo contenente la procedura chiamata (server) possono essere su macchine diverse
 - **Invocazione di metodo remoto (Java RMI)**
 - L'applicazione distribuita è realizzata invocando i metodi di un oggetto di un'applicazione Java in esecuzione su una macchina remota
 - Trasparenza della distribuzione



Requisiti del middleware di comunicazione

- Il middleware **scambia messaggi** per consentire la chiamata/invocazione di procedura/metodo; occorre:
 - Identificare i messaggi di chiamata/invocazione ed i messaggi di risposta
 - Identificare univocamente procedura/metodo remota/o
- Il middleware **gestisce l'eterogeneità dei dati** scambiati
 - **Marshaling/unmarshaling** dei parametri (dati assemblati/disassemblati in modo opportuno per la trasmissione in un messaggio) usato ad es. da RPC oppure
 - **Serializzazione dei parametri** (dati strutturati in flusso di byte) usato ad es. da Java RMI e .Net
- Il middleware **gestisce alcuni errori** dovuti alla distribuzione
 - Errori nella comunicazione
 - Errori dell'utente

Problemi

- Problemi da risolvere per realizzare la chiamata di procedura/invocazione di metodo in modo trasparente e senza sapere dove si trova la procedura chiamata/il metodo invocato
1. Come gestire l'eterogeneità nella **rappresentazione dei dati**?
 - Inoltre client e server devono anche coordinarsi sul protocollo di trasporto usato per lo scambio di messaggi
 2. In presenza di **errori**, quale è la semantica della chiamata di procedura remota (o dell'invocazione di metodo remoto)?
 - Procedura locale: semantica exactly-once
 - Procedura **remota**: semantica **at-least-once** o **at-most-once**
 3. Come avviene il **binding** al server?

Eterogeneità nella rappresentazione dei dati

- Client e server possono usare una diversa codifica dei dati, ad es.:
 - Codifica dei caratteri
 - Rappresentazione di numeri interi e in virgola mobile
 - Ordinamento dei byte (little endian vs big endian)
 - Non solo per tipi di dato elementari ma anche strutturati
- Quali alternative per gestire l'eterogeneità nella rappresentazione dei dati ed interpretare il messaggio in modo non ambiguo?
 1. La codifica è specificata nel messaggio stesso
 2. Il mittente del messaggio converte i dati nella rappresentazione attesa dal destinatario
 3. Ogni dato è convertito in un formato comune concordato
 - Mittente: da codifica proprietaria a comune
 - Destinatario: da codifica comune a proprietaria
 4. Esiste un intermediario che effettua le conversioni tra codifiche diverse

Eterogeneità nella rappresentazione dei dati

- Confrontiamo le alternative 2 e 3, nell'ipotesi di N componenti che comunicano tra loro
- Alternativa 2: dotare ogni componente di **tutte le funzioni di conversione** possibili per ogni rappresentazione dei dati
 - **Alte prestazioni** nella conversione
 - **Alto numero** di funzioni di conversione, pari a $N^*(N-1)$
- Alternativa 3: concordare un **formato comune** di rappresentazione dei dati ed ogni componente possiede le funzioni di conversione da/per questo formato
 - **Minori prestazioni** nella conversione
 - **Basso numero** di funzioni di conversione, pari a $2*N$

Design patterns to tackle heterogeneity

- **Proxy**
 - Aim: support access and location transparency (see RPC and RMI)
 - Control access to an object using another proxy object
 - The proxy is created in the local address space to represent the remote object and offers the same interface of the remote object
- **Broker**
 - Aim: separate and encapsulate the details of communication infrastructure from its functionality
 - Enables components to interact without handling remote concerns by themselves
 - Locates the server for the client, hides the details of remote communication, ...

Semantics of remote call/method

- Exactly once semantics is difficult to implement: communication middleware generally implements weaker semantics
- **At-least-once semantics**: if the client received a reply from the server, that remote call/method has been **executed at least once** on the server
- **At-most-once semantics**: if the client gets a reply from the server, it means that the remote call/method has been **executed at most once** on the server

Binding del server

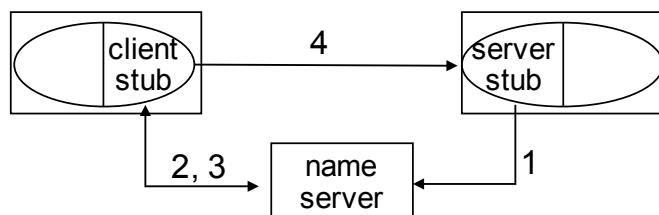
- Binding: stabilisce come agganciare il client al server che fornisce la procedura/metodo
 - Statico o dinamico
- **Binding statico**
 - Indirizzo del server cablato nel codice del client
 - Semplice e senza overhead, ma mancanza di trasparenza e flessibilità
- **Binding dinamico**
 - Si ritarda la decisione al momento dell'esecuzione
 - Maggiore overhead, ma trasparenza e flessibilità
 - Ad es. consente di dirigere le richieste sul server più scarico
 - L'overhead deve essere comunque limitato ed accettabile

Binding dinamico

- Si distinguono due fasi nella relazione client/server
- **Naming**: fase statica, prima dell'esecuzione
 - Il client specifica a chi vuole essere connesso, con un nome unico identificativo del servizio
 - Si associano dei nomi unici di sistema alle operazioni o alle interfacce astratte e si attua il binding con l'interfaccia specifica di servizio
- **Addressing**: fase dinamica, durante l'esecuzione
 - Il client deve essere realmente collegato al server che fornisce il servizio al momento dell'invocazione
 - Si cercano gli eventuali server pronti per il servizio
 - Addressing esplicito o implicito
 - **Addressing esplicito**: broadcast o multicast da parte del client, attendendo solo la prima risposta
 - Il supporto runtime di ogni macchina risponde se il servizio richiesto è fornito da un suo server in esecuzione

Binding dinamico

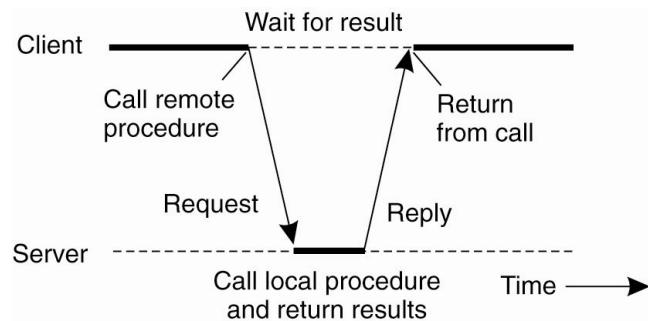
- **Addressing implicito**: presenza di **name server** (anche **binder**, **directory service**, **registry service**) che registra i servizi ed agisce su opportune tabelle di binding, fornendo funzioni di:
 - ricerca, registrazione, aggiornamento, eliminazione di servizi



- Frequenza del binding dinamico
 - Ogni chiamata richiede un collegamento dinamico
 - Spesso, per questioni di costo, dopo aver ottenuto il primo binding lo si continua ad usare come se fosse statico
 - Il binding può così avvenire meno frequentemente delle chiamate stesse
 - In genere, si usa lo stesso binding per molte chiamate allo stesso server

Remote Procedure Call (RPC)

- Idea (proposta da Birrel e Nelson, 1984): utilizzare il modello client/server con la stessa semantica di una chiamata di procedura
 - Un processo sulla macchina A invoca una procedura sulla macchina B
 - Il processo chiamante su A viene sospeso
 - Viene eseguita la procedura chiamata su B
 - Input ed output sono veicolati da parametri
 - Nessuno scambio di messaggi è visibile al programmatore



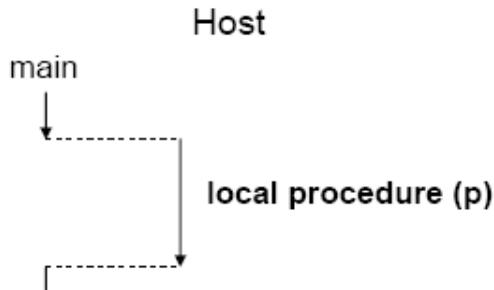
Why RPC

- Implemented and used in most distributed systems, including cloud computing systems
- Developed and employed in many technologies and languages, among which:
 - C (Sun RCP)
 - Java (Java RMI)
 - Go
 - Ice <https://zeroc.com/products/ice>
 - Microsoft .NET
 - Distributed Ruby (DRb)
 - Remote Python Call (RPyC)
 - JSON-RPC
 - CORBA

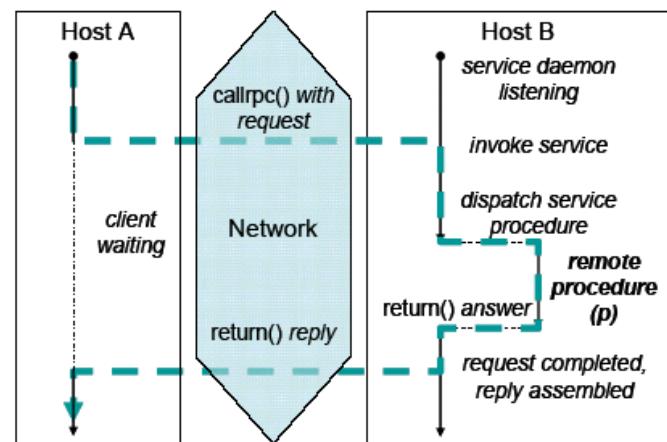
Our case studies

Chiamata di una procedura

- Procedura p **locale**

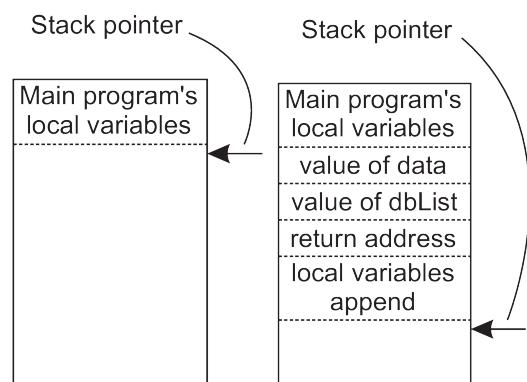


- Procedura p **remota**



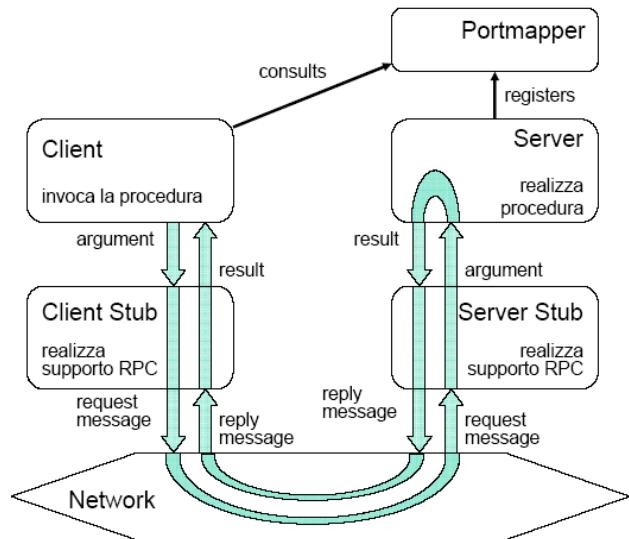
Chiamata di una procedura locale

- Esempio di chiamata “locale” (convenzionale):
`newlist = append(data, dbList)`
- La procedura chiamante inserisce nello stack i parametri di input e l’indirizzo di ritorno
- Al termine, la procedura chiamata (append) restituisce il controllo alla procedura chiamante



Architettura RPC

- Modello con **stub** (proxy)
- Il client invoca il **client stub**, che si incarica di tutto:
 - Dall'identificazione del server alla gestione dei parametri, dalla richiesta al supporto run-time all'invio della richiesta
- Il server riceve la richiesta dal **server stub**, che gestisce i parametri dopo avere ricevuto la richiesta dal livello di trasporto. Al completamento del servizio, il server stub invia il risultato al client stub
- Trasparenza
 - Stub prodotti **in modo automatico**
 - Lo sviluppatore progetta e si occupa solo delle parti applicative e logiche

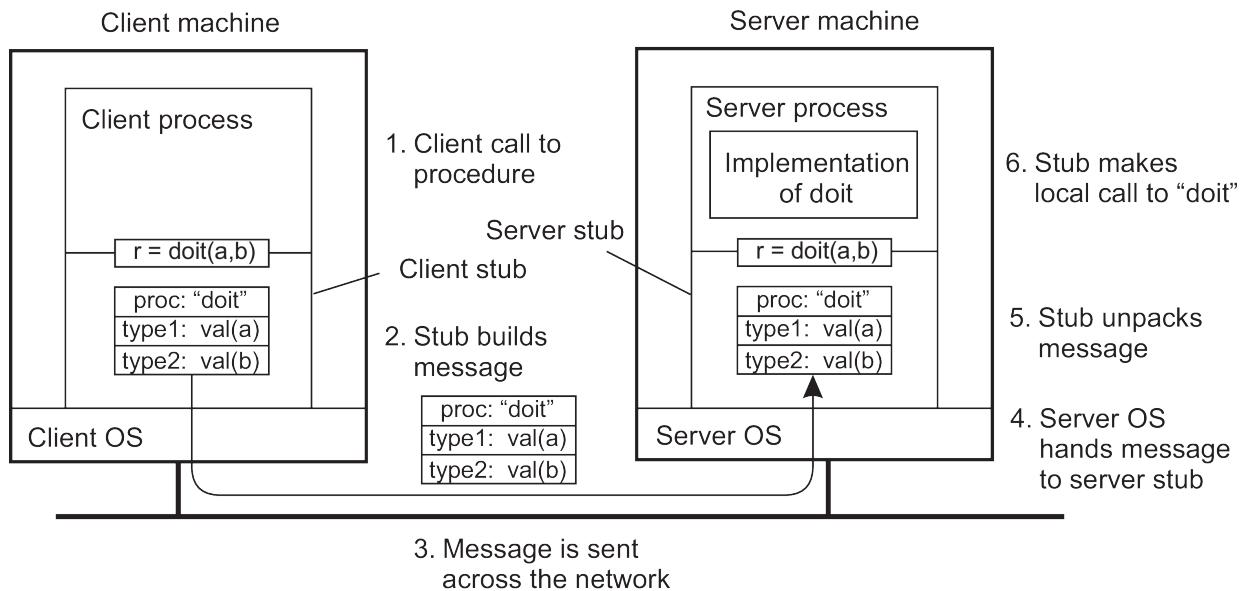


Passi di RPC

1. Lato client, la procedura chiamata invoca il client stub tramite una normale chiamata di procedura locale
2. Il client stub costruisce un messaggio e richiama il SO locale
 - **Marshaling dei parametri**: operazione di impacchettare in un messaggio i parametri, convertendoli in formato comune
3. Il SO del client invia il messaggio al SO remoto
4. Il SO remoto passa il messaggio al server stub
5. Il server stub spacchetta il messaggio prelevando i parametri e convertendoli in formato locale, e invoca il server come una procedura locale
 - **Unmarshaling dei parametri**
6. Il server esegue il lavoro e restituisce il risultato al server stub
7. Il server stub lo impacchetta in un messaggio e richiama il suo SO (marshaling dei parametri)
8. Il SO del server invia il messaggio al SO del client
9. Il SO del client passa il messaggio al client stub
10. Il client stub spacchetta il messaggio (unmarshaling dei parametri) e restituisce il risultato al client

Example: steps in RPC

- Steps when calling a remote procedure `doit(a,b)`



Problemi per RPC

1. Rappresentazione dei dati
2. Procedura chiamante e chiamata in esecuzione su macchine con diverso spazio di indirizzi: come realizzare il **passaggio dei parametri per riferimento?**
3. In presenza di **errori**, cosa può il chiamante considerare certo rispetto all'esecuzione della procedura chiamata?
4. Come effettuare il binding alla macchina su cui viene eseguita la procedura chiamata?

Rappresentazione dei dati

- Il middleware RPC fornisce un supporto automatizzato
 - Il codice per il marshaling/unmarshaling viene generato automaticamente dal middleware e diviene parte degli stub
 - Tramite
 - Una rappresentazione della procedura indipendente dal linguaggio e dalla piattaforma, scritta usando un **Interface Definition Language (IDL)**
 - Un **formato comune di rappresentazione dei dati** usato per la comunicazione

IDL per RPC

- **Linguaggio per la definizione delle interfacce** (**Interface Definition Language - IDL**)
- Permette la descrizione di operazioni remote, la specifica del servizio (detta *signature*) e la generazione degli stub
- Deve consentire
 - Identificazione non ambigua del servizio
 - Uso di nome astratto del servizio, spesso prevedendo versioni diverse del servizio
 - Definizione astratta dei dati da trasmettere in input ed output
 - Uso di un linguaggio astratto di definizione dei dati (operazioni e parametri dell'interfaccia)
- Supporta lo sviluppo dell'applicazione
 - L'interfaccia specificata dal programmatore in IDL può essere usata per generare automaticamente gli stub

Tipi di passaggio dei parametri

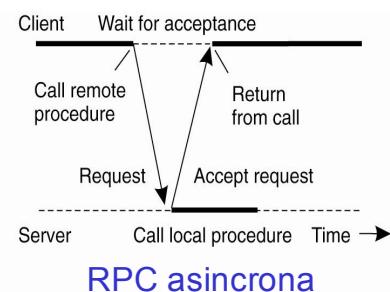
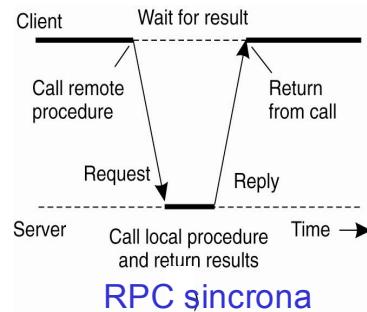
- **Passaggio per valore (call by value)**
 - I dati sono copiati nello stack
 - Il chiamato agisce su tali dati e le modifiche non influenzano il chiamante
- **Passaggio per riferimento (call by reference)**
 - L'indirizzo (puntatore) dei dati è copiato nello stack
 - Il chiamato agisce direttamente sui dati del chiamante
- **Passaggio per copia/ripristino (call by copy/restore)**
 - I dati sono copiati nello stack dal chiamante e ricopiatati dopo la chiamata, sovrascrivendo il valore originale del chiamante
 - Implementato in pochi linguaggi di programmazione (ad es. Ada, Fortran)

Problemi per il passaggio di parametri

- Un riferimento è un indirizzo di memoria
 - E' valido solo nel contesto in cui è usato
- Soluzione: si simula il passaggio per riferimento usando il passaggio per **copia/ripristino**
 - Il client stub copia la struttura dati puntata nel messaggio e lo invia al server stub
 - Il server stub agisce sulla copia, usando lo spazio di indirizzi della macchina ricevente
 - Se viene effettuata una modifica sulla copia, questa sarà poi riportata dal client stub sulla struttura dati originale
 - Occorre conoscere la dimensione dei dati da copiare
 - Se la struttura dati contiene puntatori?

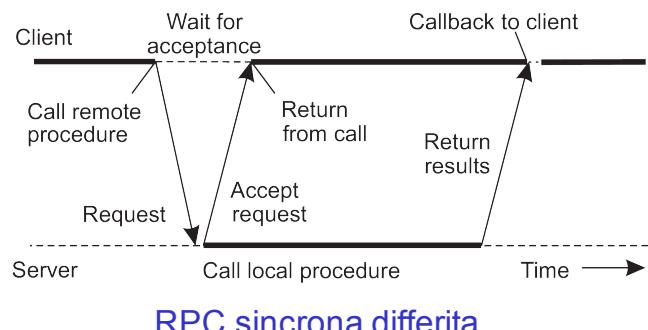
RPC asincrona

- In modalità sincrona, la chiamata RPC provoca il blocco del client
 - Non necessario se il server non deve restituire nessun risultato
- Alcuni middleware RPC supportano **RPC asincrona**
 - Il client può dedicarsi ad altri task, una volta effettuata la chiamata ed aver ricevuto dal server un ack che la chiamata di procedura è stata ricevuta ed avviata
 - Ad es. Go



RPC asincrona

- Se il client riprende l'esecuzione senza aspettare l'ack, la RPC asincrona è detta **one-way**
- Se la RPC produce un risultato, si può spezzare l'operazione in due (**RPC sincrona differita**):
 - Una prima RPC da client a server per avviare l'operazione
 - Una seconda RPC da server a client per restituire il risultato



RPC and transparency

- Is RPC truly transparent? Can we really just treat remote procedure calls as local procedure calls?
- Performance
 - Local call: maybe 10 cycles = ~3 ns
 - RPC: 0.1-1 ms on a LAN => ~100K slower
 - Major source of overhead: context switching, copies, inter-process communication
 - In WAN: can easily be millions of times slower
- Failures
- Also security, concurrent requests, ...

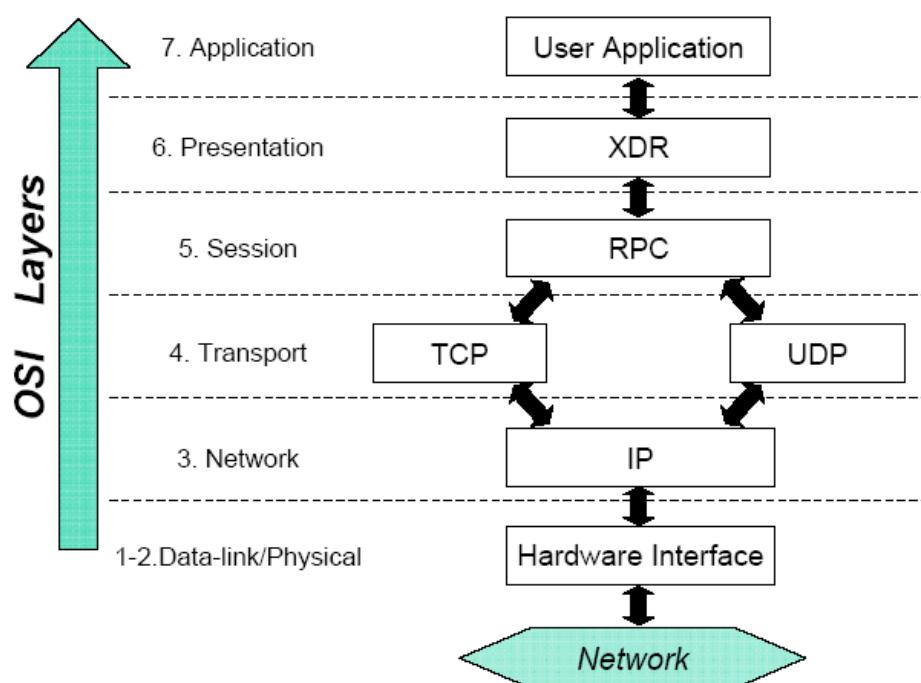
Implementing RPC: Sun RPC

- Example of first generation RPC
- An implementation and RPC middleware provided by Sun Microsystems: **Open Network Computing** (ONC) RPC, aka **Sun RPC**
 - Basic and widely used implementation
- ONC is a suite of products including:
 - **eXternal Data Representation** (XDR): IDL used by Sun RPC to manage heterogeneity in data representation
 - **Remote Procedure Call GENerator** (RPCGEN): program to automatically generate the client stub and server stub
 - **Port mapper**: service to bind the client to the server
 - Network File System (NFS): distributed file system

References for Sun RPC

- man rpc
- man rpcgen
- Chapter 16 of “Unix Network Programming, Volume 2: Interprocess Communication”, 1999.
- E. Petron, “Remote Procedure Calls”, Linux Journal, 1997.

SUN RPC and OSI model



Definizione del programma RPC

- Due parti descrittive in linguaggio XDR
- 1. **Definizioni di programmi RPC**: specifiche del protocollo RPC per le procedure (servizi) offerte, ovvero l'identificazione delle procedure ed il tipo di parametri
- 2. **Definizioni XDR**: definizioni dei tipi di dati dei parametri
 - Presenti solo se il tipo di dato non è già noto in XDR
- Raggruppate in un unico file con estensione .x
 - Esempio: square.x

Definizione della procedura remota

```
struct square_in {                                /* input (argument) */  
    long arg1;  
};  
struct square_out {      /* output (result) */  
    long res1;  
};  
program SQUARE_PROG {  
    version SQUARE_VERS {  
        square_out  SQUAREPROC(square_in) = 1; /* procedure number = 1 */  
    } = 1;                      /* version number */  
} = 0x31230000;          /* program number */
```

Esempio: square.x

Definizione della procedura remota SQUAREPROC

- Ogni procedura ha **un solo parametro d'ingresso e un solo parametro d'uscita**
- Gli identificatori (nomi) usano lettere maiuscole
- Ogni procedura è associata ad un numero di procedura unico all'interno di un programma (nell'esempio 1)

Implementazione del programma RPC

- Il programmatore deve sviluppare:
 - il **programma client**: implementazione del main() e della logica necessaria per reperimento e binding del servizio/i remoto/i (esempio: `square_client.c`)
 - il **programma server**: implementazione di tutte le procedure (servizi) (esempio: `square_server.c`)
- Attenzione: il programmatore **non** realizza il main()
lato server...
 - Chi invoca la procedura remota (lato server)?

square: procedura convenzionale

- Esempio: quadrato di un numero intero
- Esaminiamo il codice della tradizionale procedura convenzionale (o **locale**)

```
#include <stdio.h>
#include <stdlib.h>

struct square_in { /* input (argument) */
    long arg;
};

struct square_out { /* output (result) */
    long res;
};

typedef struct square_in square_in;
typedef struct square_out square_out;

square_out *squareproc(square_in *inp) {
    static square_out out;

    out.res = inp->arg * inp->arg;
    return(&out);
}
```

Esempio: `square_local.c`

square: procedura convenzionale

- Procedura **locale**: quadrato di un numero intero (*continua*)

```
int main(int argc, char **argv) {  
    square_in in;  
    square_out *outp;  
  
    if (argc != 2) {  
        printf("usage: %s <integer-value>\n", argv[0]);  
        exit(1);  
    }  
    in.arg = atol(argv[1]);  
  
    outp = squareproc(&in);  
    printf("result: %ld\n", outp->res);  
    exit(0);  
}
```

Come si trasforma nel caso di procedura remota?

square: remote procedure

- Il codice di servizio della procedura remota è *quasi* identico alla procedura locale

```
#include <stdio.h>  
#include <rpc/rpc.h>  
#include "square.h" /* generated by rpcgen */  
square_out *squareproc_1_svc(square_in *inp, struct svc_req *rqstp) {  
  
    static square_out out;  
  
    out.res1 = inp->arg1 * inp->arg1;  
    return(&out);  
}
```

Esempio: **server.c**

- Osservazioni:
 - I parametri di ingresso e uscita vengono passati per riferimento
 - Il parametro di uscita punta ad una variabile statica (allocazione globale), per essere presente anche oltre la chiamata della procedura
 - Il nome della procedura cambia leggermente (si aggiunge _ seguito dal numero di versione), tutto in caratteri minuscoli

square: client

- Il client viene lanciato passando hostname remoto e valore intero e richiede il servizio di square remoto

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "square.h" /* generated by rpcgen */
int main(int argc, char **argv) {
    CLIENT *clnt;
    char *host;
    square_in in;
    square_out *result;
    if (argc != 3) {
        printf("usage: client <hostname> <integer-value>\n");
        exit(1);
    }
    host = argv[1];
    clnt = clnt_create(host, SQUARE_PROG, SQUARE_VERS, "tcp");
```

Esempio: client.c

square: client

```
if (clnt == NULL) {
    clnt_pcreateerror(host);
    exit(1);
}
in.arg1 = atol(argv[2]);
if ((result = squareproc_1(&in, clnt)) == NULL) {
    printf("%s", clnt_sperror(clnt, argv[1]));
    exit(1);
}
printf("result: %ld\n", result->res1);
exit(0);
}
```

square: client

- `cInt_create()`: crea il **gestore di trasporto client** che gestisce la comunicazione col server
 - TPC o UDP
- Il client deve conoscere:
 - nome dell'host su cui è in esecuzione il servizio
 - informazioni per invocare il servizio (programma, versione e nome della procedura)
- Per la chiamata della procedura remota:
 - Cambia il nome della procedura: si aggiunge il carattere “_” seguito dal numero di versione (nome in caratteri minuscoli)
 - Due parametri di ingresso della procedura chiamata:
 - parametro di ingresso vero e proprio
 - gestore di trasporto del client
- Gestione degli errori che si possono presentare durante la chiamata remota
 - `cInt_pcerror()` e `cInt_perror()`

Passi di base per lo sviluppo in Sun RPC

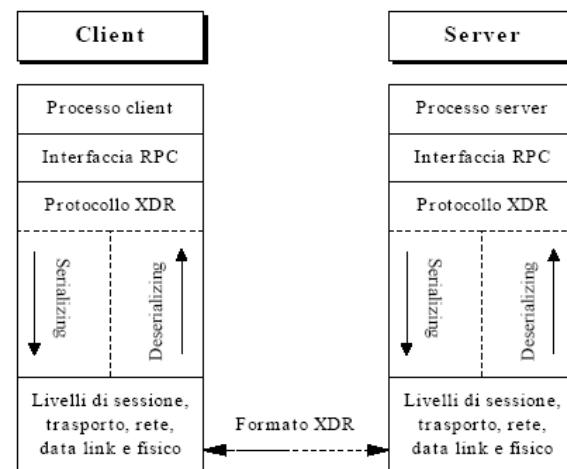
1. Definire servizi e tipi di dato (se necessario) → [square.x](#)
2. Generare tramite **rpcgen** gli stub del client e del server e le eventuali funzioni di conversione XDR
 - Genera `square_cInt.c`, `square_svc.c`, `square_xdr.c`, `square.h`
3. Realizzare i programmi client e server ([client.c](#) e [server.c](#)), compilare tutti i file sorgente (programmi client e server, stub e file per la conversione dei dati) e fare il linking dei file oggetto
4. Pubblicare i servizi (lato server)
 - Avviando il server, i servizi sono registrati presso il port mapper (servizio **rpcbind**)
5. Reperire (lato client) l'endpoint del server tramite il port mapper e creare il gestore di trasporto per l'interazione col server

Caratteristiche di Sun RPC

- Un programma tipicamente contiene **più procedure remote**
 - Possibili **versioni multiple** di ogni procedura
 - Un **unico argomento in ingresso ed in uscita** per ogni invocazione (**passaggio per copia-ripristino**)
- **Mutua esclusione** garantita dal programma (e server): di default no concorrenza lato server
 - Server **sequenziale** ed **una sola invocazione eseguita per volta**
 - Per server multithreaded (non su Linux): rpcgen con opzioni **-M e -A**
- Fino al ritorno della procedura, il client è in attesa **sincrona bloccante** della risposta
- Semantica **at-least-once** della comunicazione
 - Ritrasmissione allo scadere di un intervallo di time-out
 - UDP come protocollo di trasporto di default

eXternal Data Representation (XDR)

- Eterogeneità dei dati gestita usando un **formato comune di rappresentazione**
- Funzioni XDR di conversione built-in, relative a:
 - Tipi atomici predefiniti
 - Ad es. `xdr_bool()`, `xdr_char()`, `xdr_int()`
 - Tipi strutturati predefiniti
 - Ad es. `xdr_string()`, `xdr_array()`
- Ciascuna funzione XDR creata da rpcgen può essere usata sia per serializzare (encode) che per deserializzare (decode)



Definizione del file.x

- Prima parte del file
 - Definizioni XDR delle **costanti**
 - Definizioni XDR dei **tipi di dato** dei parametri di ingresso e uscita per tutti i tipi di dato per i quali non esiste una corrispondente funzione XDR built-in
- Seconda parte del file
 - Definizioni XDR delle **procedure**
- Esempio in square.x: SQUAREPROC è la procedura numero 1 della versione 1 del programma numero 0x31230000
- In base alle specifiche di RPC:
 - Il numero di procedura zero è riservato a NULLPROC
 - Un solo parametro d'ingresso e d'uscita per ogni procedura
 - Gli identificatori di programma, versione e procedura usano tutte lettere maiuscole

Sun RPC: binding

- La procedura deve essere registrata prima di poter essere invocata
 - Occorre specificare l'identificatore RPC: numero di programma (*proignum*), numero di versione (*versnum*), numero di procedura
 - Occorre specificare anche il protocollo di trasporto (*protocol*)
 - Il client deve conoscere il numero di porta (*port*) su cui il server risponde
- Il server RPC registra il programma RPC nella **port map**
 - Tabella dinamica dei servizi RPC dell'host
 - Ogni riga della port map contiene la tripla {*proignum*, *versnum*, *protocol*} e *port*
 - Manca il numero di procedura: tutte le procedure all'interno di un programma condividono lo stesso gestore di trasporto
- La tabella di port map è gestita da un solo processo (detto **port mapper**) per ogni host

Port mapper (rpcbind)

- Port mapper (server rpcbind) in ascolto su porta 111
- All'avvio il server stub registra sul port mapper le informazioni sui servizi RPC offerti
proignum, versnum, protocol e port
- Il client stub contatta il port mapper per conoscere la porta corrispondente prima di invocare la procedura remota
- Il port mapper registra i servizi offerti e supporta:
 - Inserimento di un servizio
 - Eliminazione di un servizio
 - Ricerca della porta per un servizio
 - Lista dei servizi registrati

Port mapper (rpcbind)

- Per la lista di tutti i programmi RPC invocabili sull'host: `rpcinfo -p nomehost`

```
>$ rpcinfo -p
program      vers   proto   port
 100000        4     tcp     111      rpcbind
 100000        4     udp     111      rpcbind
 824377344      1     udp     59528
 824377344      1     tcp     49311
```

824377344 (= 0x31230000) è il numero di programma in square.x

- In /etc/rpc elenco dei programmi RPC disponibili

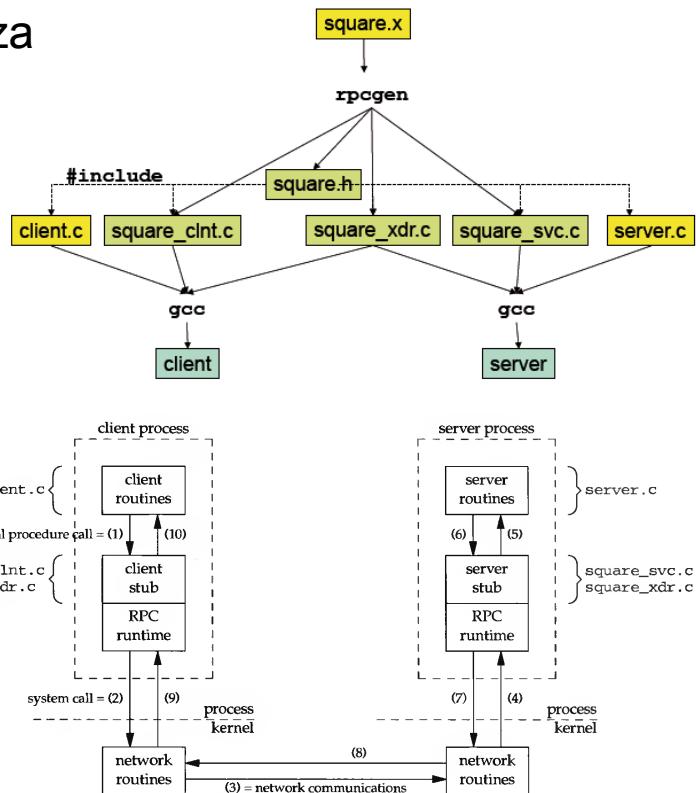
Processo di sviluppo in SUN RPC

Data una specifica di partenza

- Scritta in XDR: square.x

rpcgen produce

- Header: square.h
- Client stub: square_clnt.c
- Server stub: square_svc.c
- Routine XDR: square_xdr.c



Lo sviluppatore scrive

- programma client: client.c
- programma server: server.c

Esempi RPC

- Esempi di programmi RPC sul sito del corso
 - square: quadrato di un numero intero
 - Esaminiamo il codice di client stub e server stub
 - echo: ripetizione di una sequenza di caratteri
 - avg: valore medio di una sequenza di numeri reali
 - rls: listato di una directory remota
 - Routine XDR in dir_xdr.c generate automaticamente a partire dai tipi di dato definiti in dir.x: permettono la conversione da formato locale a XDR e viceversa

Second generation of RPC

- In the 1990s second generation of RPC
- Support for object oriented languages: distributed objects
 - Microsoft DCOM
 - CORBA
 - **Java RMI**

Java RMI: motivazioni

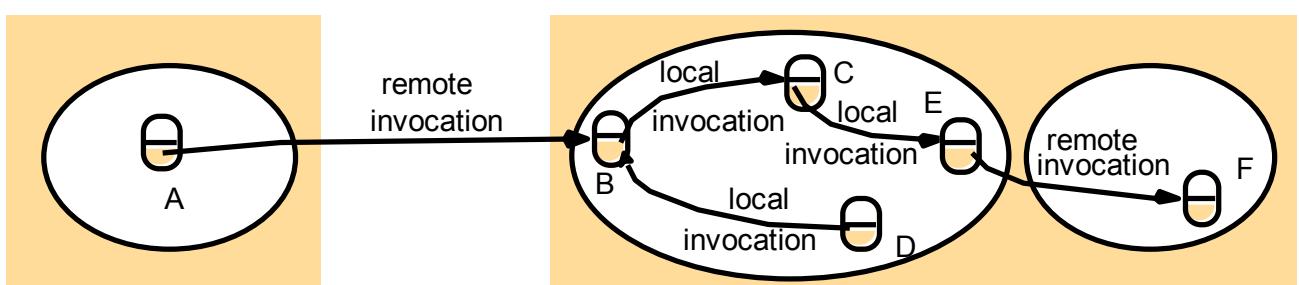
- Estendere RPC al modello a oggetti distribuiti
 - Java RMI (**Remote Method Invocation**): RPC in Java
 - RMI fornisce la possibilità di **invocare un metodo (di un'interfaccia remota) su un oggetto remoto**
- Java RMI: insieme di **strumenti, politiche e meccanismi** che permettono ad un'applicazione Java in esecuzione su un host di invocare i metodi di un oggetto remoto in esecuzione su un altro host
 - Obiettivo: trasparenza dell'accesso (invocazione locale e remota il più possibile simili)
 - Ma la trasparenza della distribuzione non è completa

References for Java RMI

- “Trail: RMI”, The Java Tutorials
<http://docs.oracle.com/javase/tutorial/rmi/>
 - Note: JDK 8 (current release JDK 11)
- W. Grosso, “Java RMI”, O'Reilly, 2001.

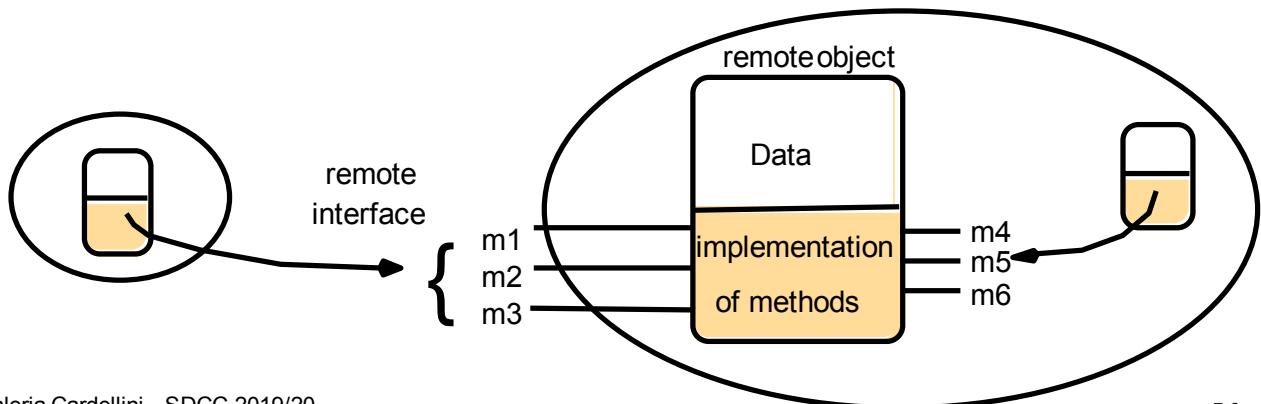
Java RMI: generalità

- Localmente viene creato il **riferimento ad un oggetto remoto**, che è attivo su un host remoto
- Il client invoca i **metodi remoti** attraverso il riferimento locale
- Quali sono le differenze rispetto all'invocazione di un **metodo locale**?
 - Affidabilità, semantica della comunicazione, durata, ...



Java RMI: generalità

- Separazione tra definizione del comportamento ([interfaccia](#)) e implementazione del comportamento ([classe](#))
- Separazione logica tra interfaccia ed oggetto consente anche la loro [separazione fisica](#)
 - **Interfaccia remota**: specifica quali metodi dell'oggetto possono essere invocati da remoto
 - Lo stato interno di un oggetto non viene distribuito!

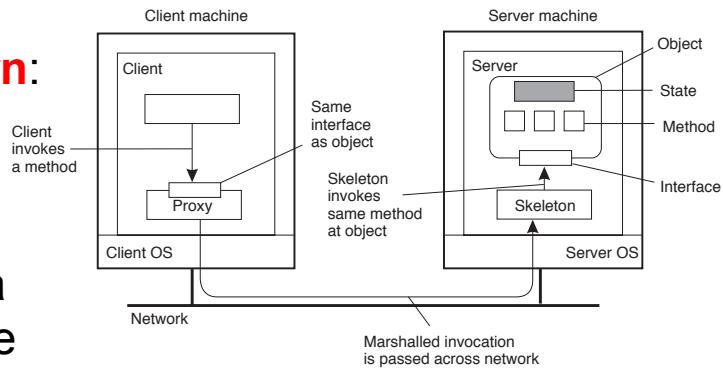


Java RMI: generalità

- Al binding del client con l'oggetto server remoto, la copia dell'interfaccia del server (**stub**) viene caricata nello spazio del client
 - Ruolo analogo al [client stub](#) in ambiente RPC
- La richiesta in arrivo all'oggetto remoto viene trattata da un “agente” del client, locale al server (**skeleton**)
 - Ruolo analogo al [server stub](#) in ambiente RPC
- Unico ambiente di lavoro come conseguenza del linguaggio Java, ma eterogeneità dei sistemi
 - Grazie alla portabilità del codice Java

Stub e skeleton

- Come in RPC, anche in RMI si usa il **proxy pattern**: i due **proxy (stub** dal lato client e **skeleton** dal lato server) nascondono al livello applicativo la natura distribuita dell'applicazione
 - Stub**: proxy locale **sul nodo client** su cui vengono invocati i metodi dell'oggetto remoto
 - Skeleton**: proxy remoto **sul nodo server** che riceve le invocazioni fatte sullo stub e le realizza, effettuando le corrispondenti chiamate sul server



- Stub e skeleton rendono possibile l'invocazione di un servizio remoto come se fosse locale, agendo da proxy
 - Generati automaticamente (a differenza di SUN RPC)

Serializzazione/deserializzazione

- (De)serializzazione supportata direttamente da Java
 - Grazie all'uso del bytecode, non c'è bisogno di (un)marshaling come in RPC, ma i dati vengono (de)serializzati utilizzando le funzionalità offerte direttamente a livello di linguaggio
- Serializzazione**: trasformazione di oggetti complessi in sequenze di byte
 - metodo **writeObject** su uno stream di output
- Deserializzazione**: decodifica di una sequenza di byte e costruzione di una copia dell'oggetto originale
 - metodo **readObject** da uno stream di input
- Stub e skeleton utilizzano queste due funzionalità per lo scambio dei parametri di ingresso e uscita con l'host remoto

Serializzazione/deserializzazione

- Esempio di oggetto serializzabile “Record” con scrittura su stream

```
Record record = new Record();
FileOutputStream fos = new FileOutputStream("data.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(record);

InputStream fis = new FileInputStream("data.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
record = (Record)ois.readObject();
```

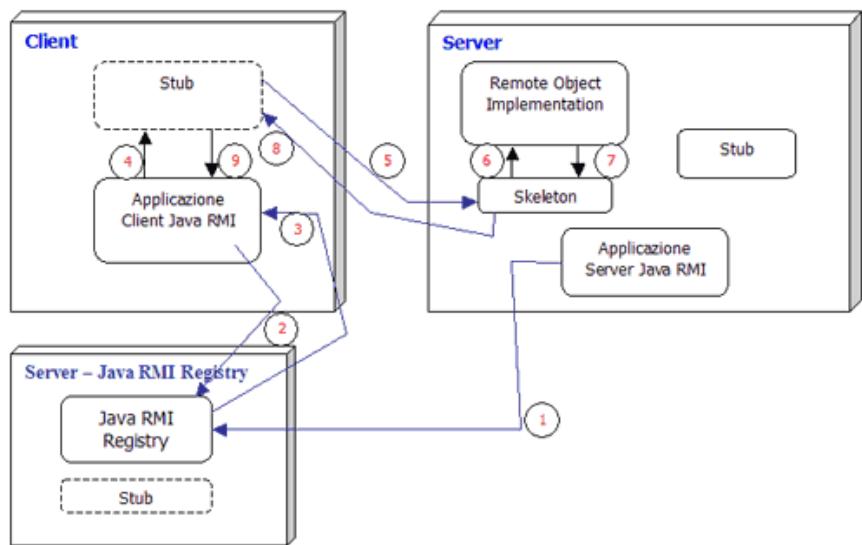
- Si possono usare soltanto istanze di oggetti serializzabili, ovvero che:
 - implementano l’interfaccia Serializable
 - contengono esclusivamente oggetti (o riferimenti a oggetti) serializzabili
- Alla deserializzazione sarà ricreata una copia dell’istanza “trasmessa” usando il .class (deve essere accessibile!) dell’oggetto e le informazioni ricevute

Interazione tra stub e skeleton

- Passi per la comunicazione
 1. Il client ottiene un’istanza dello stub (**come?**)
 2. Il client invoca i metodi sullo stub
 - Sintassi dell’invocazione remota identica a quella locale
 3. Lo **stub** effettua la **serializzazione** delle informazioni per l’invocazione (ID del metodo e parametri) e le **invia** allo skeleton in un messaggio
 4. Lo **skeleton riceve** il messaggio ed effettua la **deserializzazione** dei dati ricevuti, **invoca** la chiamata sull’oggetto che implementa il server (dispatching), effettua la **serializzazione** del valore di ritorno e lo **invia** allo stub in un messaggio
 5. Lo **stub** effettua la **deserializzazione** del valore di ritorno e restituisce il risultato al client

RMI registry

- **RMI Registry:** binder per Java RMI (porta 1099)
- Consente al server di registrare l'oggetto remoto e al client di recuperarne lo stub



- URL RMI: inizia con `rmi:` e contiene un hostname (opzionale), un numero di porta (opzionale) e il nome dell'oggetto remoto
- Alcune limitazioni: no trasparenza all'ubicazione, no gestione della sicurezza

Java RMI: passi essenziali

- Realizzare i componenti remoti lato server
 - Definizione del comportamento: **interfaccia** che
 - E' dichiarata **public** per poter essere utilizzata da altre JVM
 - Estende **java.rmi.Remote**
 - Ogni metodo remoto deve sollevare l'eccezione **java.rmi.RemoteException** per gestire anomalie derivanti da errori di rete o indisponibilità del server (può anche sollevare eccezioni specifiche dell'applicazione)
 - Implementazione del comportamento: **classe** che
 - Implementa l'interfaccia definita
 - Estende **java.rmi.UnicastRemoteObject**
 - unicast perché viene definito il riferimento ad un solo oggetto remoto (singolo indirizzo IP e porta: no replicazione dell'oggetto remoto)
 - Codice del server:
 - Istanzia l'oggetto remoto
 - Registra l'oggetto presso l'RMI registry, invocando **bind/rebind** (in **java.rmi.Naming**); **rebind** sostituisce un'eventuale associazione già esistente

Passi essenziali per usare Java RMI

- Realizzare i componenti locali lato client
 - Ottiene il riferimento all'oggetto remoto, invocando il metodo **lookup** sul registry
 - Lo assegna ad una variabile che ha l'interfaccia remota come tipo

Passi essenziali per usare Java RMI

- Dopo aver sviluppato il codice, occorre:
 1. Compilare le classi
 2. Attivare l'RMI registry (comando **rmiregistry**), che viene lanciato in un processo separato (rispetto a quello in cui è eseguito il server RMI) ed ha struttura e comportamento standard
 - In alternativa, si può creare all'interno del codice server un proprio registry locale tramite il metodo **createRegistry** (in `java.rmi.registry`)
 - Registry locale al server per motivi di sicurezza
 3. Avviare il server
 4. Avviare il client

Esempio echo: interfaccia

- L'interfaccia estende l'interfaccia **Remote**
- Ciascun metodo remoto
 - Deve lanciare una **RemoteException**
 - Per le eccezioni relative a failure durante la comunicazione
 - L'invocazione dei metodi remoti non è completamente trasparente
 - Un solo parametro di uscita; nessuno, uno o più parametri di ingresso
 - Passaggio dei parametri di ingresso del metodo remoto:
 - **per valore**, se sono tipi primitivi (boolean, char, int, ...) o oggetti che implementano l'interfaccia `java.io.Serializable`: serializzazione/deserializzazione ad opera di stub/skeleton
 - **per riferimento**, se sono oggetti `Remote`

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface EchoInterface
    extends Remote{
    String getEcho(String echo)
        throws RemoteException;
}
```

Esempio echo: server

- Classe che implementa il server
 - Estende la classe **UnicastRemoteObject**
 - Implementa tutti i metodi definiti nell'interfaccia
 - Il metodo `super()` chiama il costruttore della classe **UnicastRemoteObject** che esegue le inizializzazioni necessarie per consentire di rimanere in attesa di richieste su una porta e gestirle

```
public class EchoRMIServer
    extends UnicastRemoteObject
    implements EchoInterface{
    //Costruttore
    public EchoRMIServer()
        throws RemoteException
    { super(); }
    //Implementazione del metodo remoto
    //dichiarato nell'interfaccia
    public String getEcho(String echo)
        throws RemoteException
    { return echo; }
    public static void main(String[] args) {
        // Registrazione del servizio
        try
        { EchoRMIServer serverRMI =
            new EchoRMIServer();
            Naming.rebind("EchoService", serverRMI);
        catch (Exception e)
            {e.printStackTrace(); System.exit(1); }
    }
}
```

Esempio echo: server

- In **main** si crea l'istanza dell'oggetto server; una volta creato, l'oggetto è pronto per accettare richieste remote
- L'RMI registry locale al server registra i servizi
 - Metodi **bind/rebind** della classe **Naming** (**rebind** rimpiazza un binding già esistente)
 - Tante bind/rebind quanti sono gli oggetti server da registrare, ognuno con un proprio nome logico
- Registrazione nel servizio di naming offerto dall'RMI registry
 - bind/rebind solo su RMI registry locale per motivi di sicurezza

```
public class EchoRMIServer
    extends UnicastRemoteObject
    implements EchoInterface{
    //Costruttore
    public EchoRMIServer()
        throws RemoteException
    { super(); }
    //Implementazione del metodo remoto
    //dichiarato nell'interfaccia
    public String getEcho(String echo)
        throws RemoteException
    { return echo; }
    public static void main(String[] args) {
        // Registrazione del servizio
        try {
            EchoRMIServer serverRMI =
                new EchoRMIServer();
            Naming.rebind("EchoService", serverRMI);
        } catch (Exception e) {
            e.printStackTrace(); System.exit(1);
        }
    }
}
```

Esempio echo: client

- Servizi acceduti tramite l'interfaccia, ottenuta tramite **lookup** all'RMI registry
- Reperimento di un riferimento remoto
 - Ossia un'istanza di stub dell'oggetto remoto (non della classe dello stub, che di solito si assume già presente sul client)
- Invocazione del metodo remoto
 - Chiamata **sincrona bloccante** con i parametri specificati nell'interfaccia

```
public class EchoRMIClient
{
    //Avvio del client RMI
    public static void main(String[] args)
    {
        bufferedReader stdIn =
            new BufferedReader(
                new InputStreamReader(System.in));
        try {
            //Connessione al servizio RMI remoto
            EchoInterface serverRMI = (EchoInterface)
                Naming.lookup("EchoService");
            //Interazione con l'utente
            String message, echo;
            System.out.print("Messaggio? ");
            message = stdIn.readLine();
            //Richiesta del servizio remoto
            echo = serverRMI.getEcho(message);
            System.out.println("Echo: "+echo+"\n");
        } catch (Exception e) {
            e.printStackTrace(); System.exit(1);
        }
    }
}
```

Esempio completo
sul sito del corso

Esempio echo: compilazione ed esecuzione

- Lato server

1. Compilazione

```
javac EchoInterface.java
```

```
javac EchoRMIServer.java
```

2. Esecuzione lato server

Avvio in background del registry: `rmiregistry [registryPort]` &

Avvio del server: `java EchoRMIServer`

- Lato client

1. Compilazione

```
javac EchoRMIClient.java
```

2. Esecuzione lato client

```
java EchoRMIClient
```

Java RMI: passaggio dei parametri

- In locale:

- Per valore: tipi primitivi
 - Per riferimento: tutti gli oggetti Java (tramite indirizzo)

- In remoto (problemi nel riferimento/indirizzo):

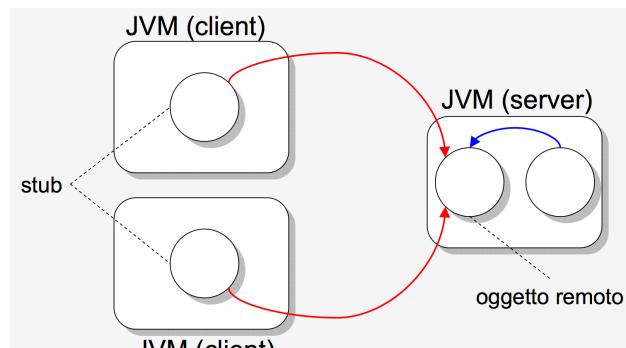
- Per valore: tipi primitivi e oggetti serializzabili
 - Oggetti la cui locazione non è rilevante per lo stato sono passati per valore: ne viene serializzata l'istanza che sarà deserializzata a destinazione per creare una copia locale
 - Per riferimento remoto: *Remote Object via RMI*
 - Oggetti la cui funzione è strettamente legata alla località in cui eseguono (server) sono passati per riferimento: ne viene serializzato lo stub, creato automaticamente a partire dalla classe dello stub
 - Ogni istanza di stub identifica l'oggetto remoto al quale si riferisce attraverso un identificativo che è univoco rispetto alla JVM dove l'oggetto remoto si trova

Java RMI: concorrenza su oggetti remoti

- I metodi di un oggetto remoto possono essere invocati in modo concorrente da molteplici client
 - Dalla specifica: “*Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe*”
- Per proteggere metodi remoti da accessi concorrenti “pericolosi” garantendo thread safety occorre definire il metodo come **synchronized**
- Esempio: metodo remoto che incrementa un contatore

Java RMI: distributed garbage collection

- Come eliminare gli oggetti remoti per i quali non ci sono più riferimenti?
 - L’oggetto remoto deve conoscere in ogni istante quanti client stub lo stanno riferendo
 - Ma sono possibili guasti di rete, crash del client, ...
- Per risolvere il problema, RMI richiede un alto grado di coordinamento
 - Limite: RMI adatto per applicazioni distribuite di piccola/media scala



Java RMI: Distributed garbage collection

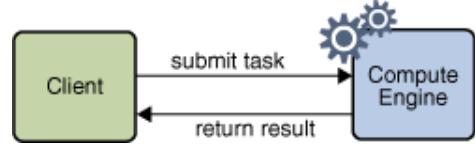
- Garbage collection basato sul conteggio dei riferimenti
 - Ogni JVM aggiorna una serie di contatori, ciascuno associato ad un determinato oggetto
 - Ogni contatore rappresenta il numero dei riferimenti ad un certo oggetto che in quel momento sono attivi nella JVM
 - Ogni volta che viene creato un riferimento ad un oggetto remoto, il relativo contatore viene incrementato
 - Alla prima occorrenza, il client invia un messaggio al server per avvertirlo
 - Quando un riferimento viene eliminato, il relativo contatore viene decrementato. Se si tratta dell'ultima occorrenza, il client avverte il server
 - Quando non ci sono più client stub che stanno usando l'oggetto remoto, il server dealloca la memoria attraverso una garbage collection locale

Java RMI: Distributed garbage collection

- Poiché il client può subire un crash, RMI utilizza un meccanismo basato su **lease** (contratto)
 - Il client detiene il contratto e deve rinnovarlo ad intervalli regolari
 - Se il client non rinnova il lease, il server assume che il client sia caduto o ci siano stati problemi di rete, quindi decrementa il contatore dei riferimenti remoti
- Soluzione costosa in termini di overhead di comunicazione
 - Poco adatta per applicazioni a larga scala e con distribuzione geografica

Esempio: compute engine

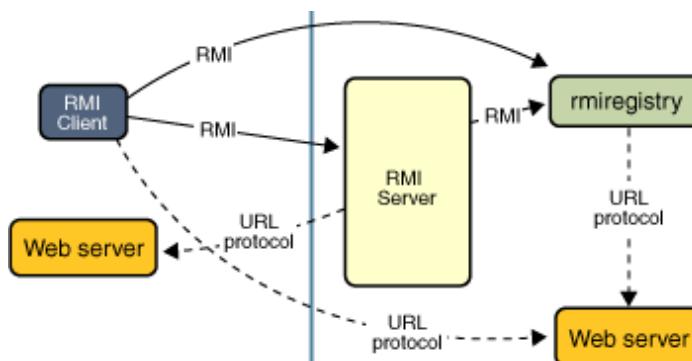
- Implementare un compute engine
 - Oggetto remoto che consente ad un server di ricevere dei task dai client, eseguirli e restituire il risultato
 - Il task viene definito dal client ma viene eseguito sulla macchina del server
 - Il task può variare indipendentemente dal server, l'importante è che rispetti una determinata interfaccia
 - Il compute engine scarica dal client il codice del task e lo esegue all'interno della propria JVM
- Due interfacce per implementare un compute engine
 - Interfaccia remota Compute, che consente ai client di inviare task al compute engine
 - Interfaccia Task, che consente al compute engine di eseguire i task



Codice su <http://docs.oracle.com/javase/tutorial/rmi/>

Esempio: compute engine

- Il codice dell'esempio include anche:
 - Caricamento dinamico delle classi tramite web server



- Creazione ed installazione di un **security manager** e definizione dei relativi file di policy
 - Garantisce che le classi caricate non eseguano operazioni non messe in_permsesse <https://docs.oracle.com/javase/tutorial/rmi/running.html>

Confronto tra SUN RPC e Java RMI

- **SUN RPC**: visione a processi, trasparenza all'accesso incompleta, no trasparenza all'ubicazione
- **Entità che si possono richiedere**: operazioni o funzioni
- **Comunicazione**: sincrona e asincrona
- **Semantica di comunicazione** (default): at-least-once
- **Durata massima ed eccezioni**: timeout per ritrasmissione e gestione di errori
- **Binding al server**: port mapper sul server
- **Presentazione dei dati**: IDL specifico (XDR) e generazione automatica di client stub e server stub
- **Passaggio dei parametri**: per copia-ripristino
- Varie estensioni, tra cui broadcast (più risposte da molteplici server anziché una sola) e sicurezza

Confronto tra SUN RPC e Java RMI

- **Java RMI**: visione ad oggetti, trasparenza all'accesso, no trasparenza all'ubicazione, distribuzione non totalmente trasparente (semantica passaggio parametri, interfacce ed eccezioni remote)
- **Entità che si possono richiedere**: metodi di oggetti via interfacce
- **Comunicazione**: sincrona
- **Semantica di comunicazione**: at-most-once
- **Durata massima ed eccezioni**: gestione di errori
- **Binding al server**: RMI registry
- **Presentazione dei dati**: Java come IDL e generazione automatica di stub e skeleton
- **Passaggio dei parametri**: per valore (tipi primitivi e oggetti serializable), per riferimento nel caso di oggetti con interfacce remotizzabili (oggetti remote)

RPC in Go

- Let us analyze the main features of the Go programming language and how Go supports RPC

<http://www.ce.uniroma2.it/courses/sdcc1920/slides/Go.pdf>

Comparing RPC implementations

- What are the differences in terms of distribution transparency among SUN RPC, Java RMI and Go?