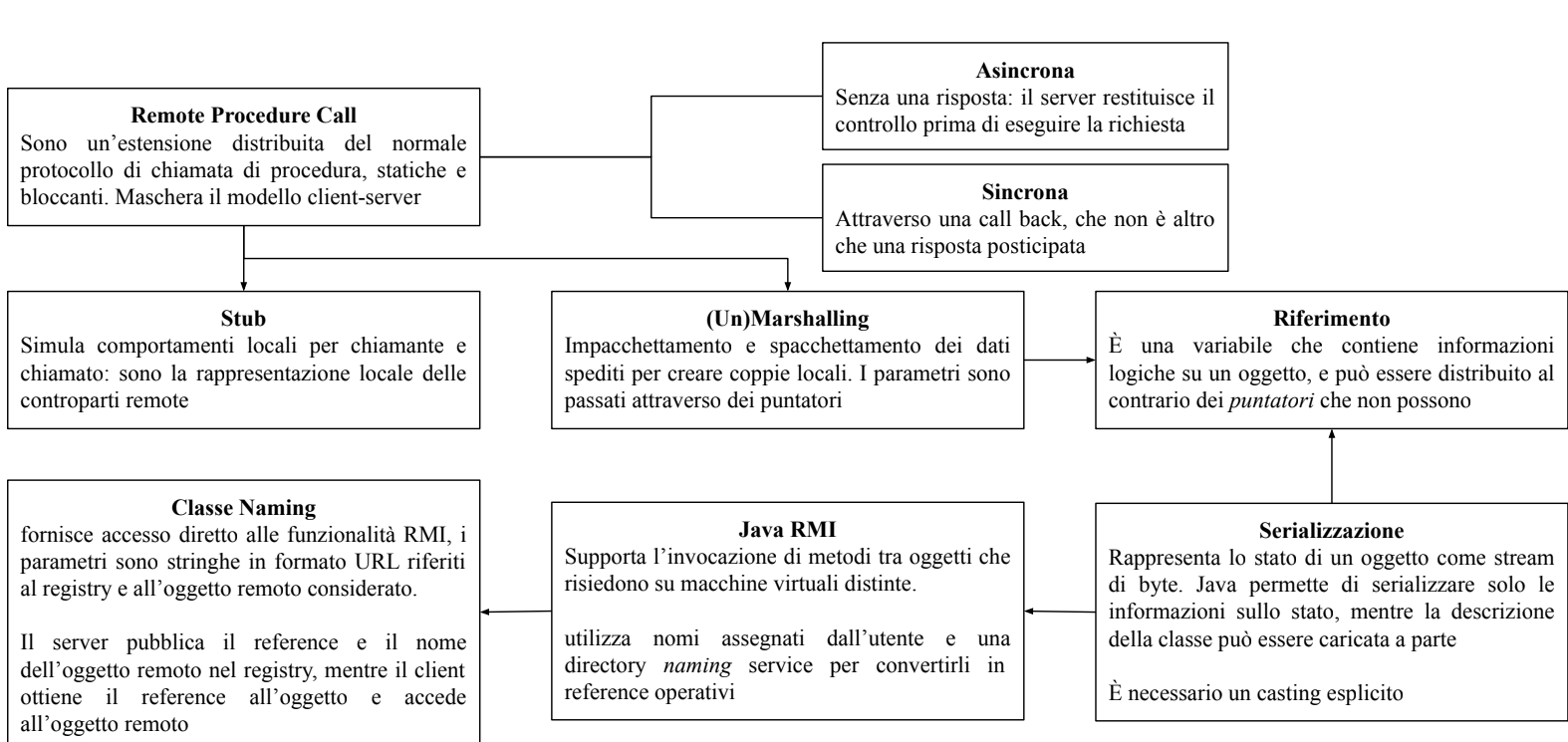
# Remote Procedure Call- RMI

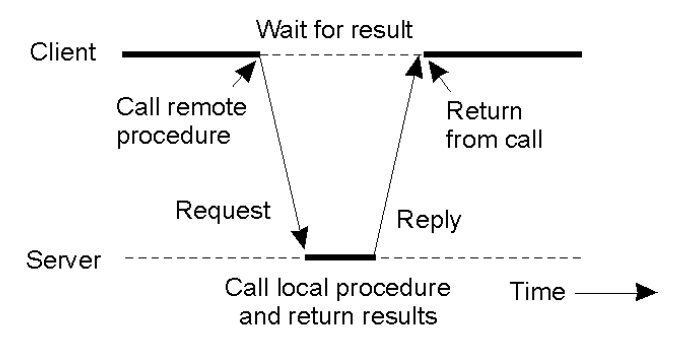


Sono come le chiamate di procedura ma adattate per gli SD (ovvero in 2 macchine distinte). Hanno il vantaggio di essere semplici da implementare ed essere concettualmente vicine al modello client-server.

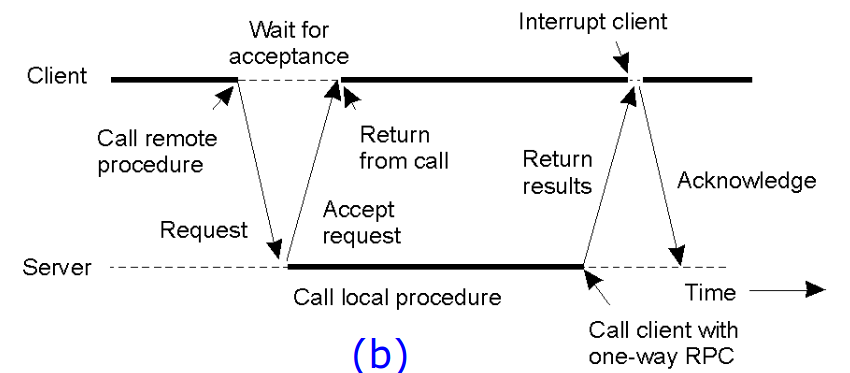
Lo svantaggio è che sono statiche (ma si risolve) ma soprattutto è che sono bloccanti, ovvero senza concorrenza.

### MODELLI RPC

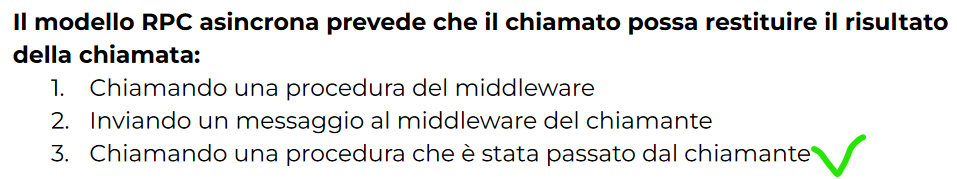
Modello RPC sincrono: so che la risposta del server è la risposta finale.



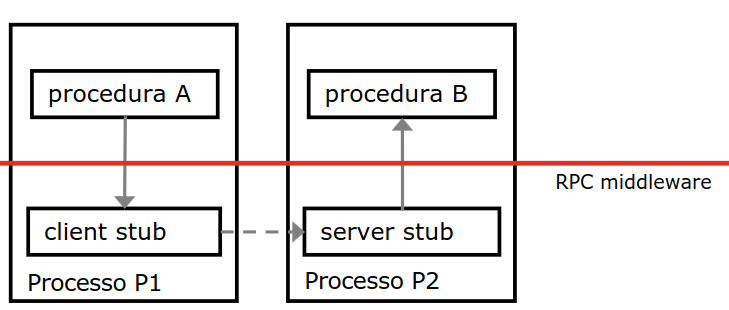
Modello RPC asincrono: mi ritorna il controllo subito dopo che la richiesta è stata ricevuta, ovvero il middleware del chiamato ha ricevuto il messaggio di invocazione. Ovviamente la chiamata è sospensiva. Tra l’altro questo modello risolve il problema della concorrenza delle RPC.



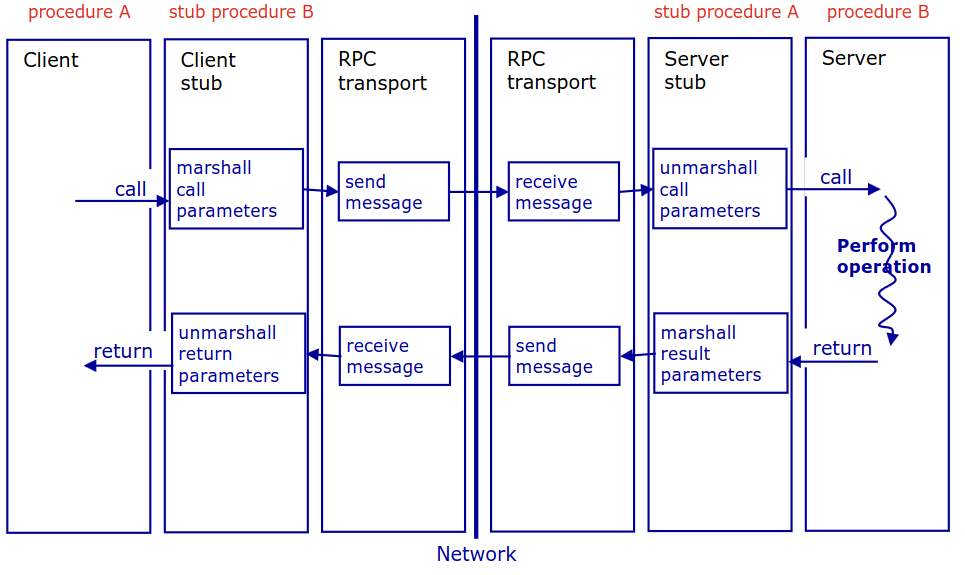
È asincrono perché quando il cLient manda il messaggio non gli interessa cosa sta facendo il server e lo manda lo stesso. Alla fine poi c’è una call-back (potrebbe non esserci).



Una RPC deve essere gestita da un middleware, trasforma le chiamate da locali a remote. Ovvero le trasforma da mono-processo a 2 processi divisi. Il client stub sfrutta un protocollo per comunicare la RPC al server stub, che poi gli risponde.



Proviamo a vedere un esempio più articolato: per passare i parametri spesso si fa per valore oppure per puntatore. Non possiamo passare però il puntatore,in quanto non conosciamo le allocazioni in memoria del server, quindi gli devo passare per forza i valori, poi il server replicherà lo stack. Questa operazione si chiama marshalling/unmarshalling. Unmarshalling = Tradurre i dati ricevuti in un formato adeguato alla chiamata locale.

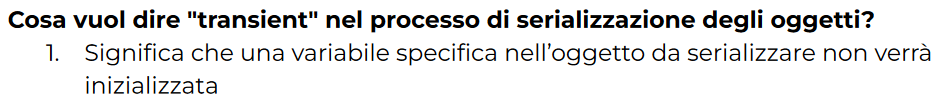
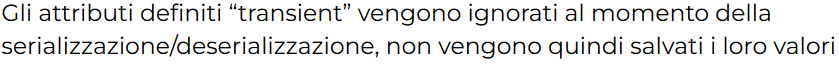


### OGGETTI DISTRIBUITI

I Distributed Objects sono oggetti che operano in un contesto distribuito, dove la loro esecuzione può avvenire su VM distinte.

* **Incapsulazione dei dati**: gli oggetti contengono dati (stato) e definiscono operazioni su questi dati attraverso metodi.
* **Accesso ai dati**: i metodi delle classi definiscono l'accesso ai dati, mentre le interfacce definiscono i metodi di accesso.
* **Definizione a compile-time e run-time**: gli oggetti sono definiti tramite interfacce e classi a compile-time, ma sono accessibili tramite adattatori (wrappers) a run-time.
* **Persistenza e transienti**: gli oggetti possono essere persistenti o transienti

transiente: il destinatario non è connesso e i dati vengono scartati;

persistente: il middleware conserva i dati fino alla consegna del messaggio al destinatario e non è necessario che i processi siano in esecuzione prima e dopo l’invio/ricezione dei messaggi.

* **Riferimento agli oggetti remoti**: consentono l'interazione con oggetti su macchine virtuali diverse. Un reference distribuito è composto da IP, porta, ID oggetto.

Esempi di Distributed Objects includono le RPC ma anche le:

### Java Remote Method Invocation (RMI)

Java RMI è un middleware che estende l'approccio OOP al contesto distribuito, supportando l'invocazione di metodi tra oggetti su macchine virtuali distinte.

In Java/RMI l’access transparency viene realizzata con l’utilizzo della stessa sintassi per accedere a oggetti locali e remoti.

Class loader va prima a vedere se la classe ce l’ha localmente, altrimenti la richiede ad un’altro.

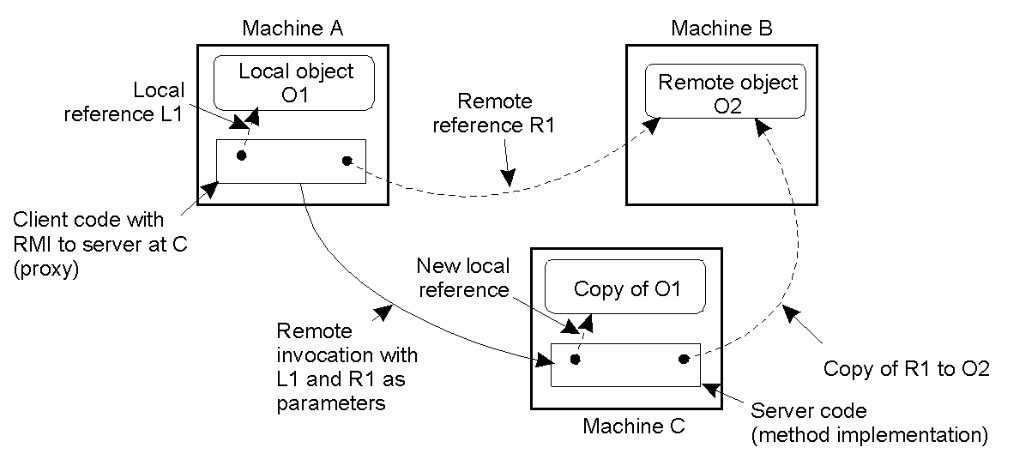
**Trasferimento per valore**: si usa per passare gli oggetti non remoti, vengono passati per valore, se serializzabili

**Trasferimento per reference**: i riferimenti ad oggetti remoti vengono passati per valore per permettere invocazioni remote. Le informazioni necessarie contenute in un reference in sono Id del metodo e argomenti.

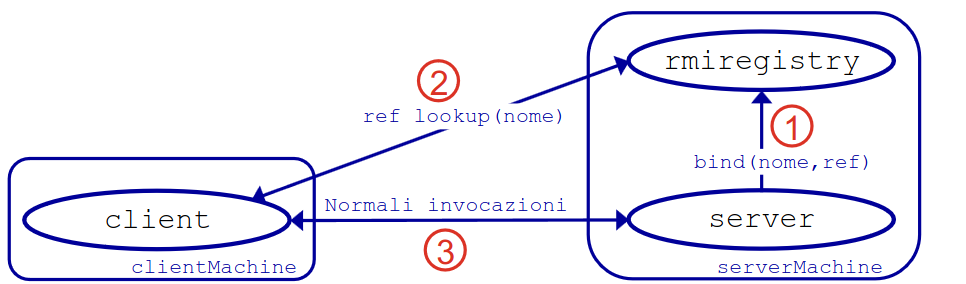
La serializzazione trasmette un oggetto su reti locali come parametro su uno stream di byte TCP/IP. La serializzazione mi da i dati, la struttura invece me la dà la classe.

Metodi obbligatori da implementare? nessuno, maker

### Passaggio dei parametri per reference oppure per valore:



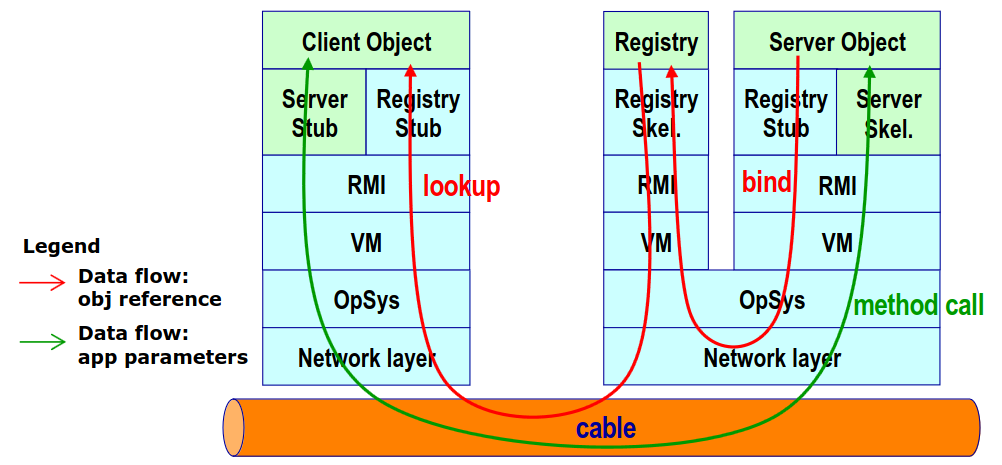
Come faccio a far conoscere la macchina A (e tutte le sue classi) a tutte le altre macchine, che sono autonome e indipendenti? Deve esserci una terza parte che conosce tutti i riferimenti agli oggetti delle altre macchine, è chiamato rmiregistry. Risiede sulla macchina dove risiedono gli oggetti (in questo caso è il server).



La classe Naming ha dei metodi statici, che permettono di identificare gli oggetti nel registro, attraverso un URL. Questo dà accesso diretto alle funzionalità del RMI Registry.

In Java RMI è possibile oggetto in LAN? SI, applicativo

### Architettura RMI



Lo stub azzurro è quello che compete al sistema, la parte verde è quello che compete all’applicazione.

* Il server pubblica l’oggetto remoto nel RMIregistry con Bind
* Il client ottiene il reference all’oggetto remoto con Lookup
* Il client accede all’oggetto remoto

Ma come fanno le VM a parlarsi? Si usa RMI-IIOP, protocollo standard per scambiare informazioni sugli oggetti. Poi ovviamente si usa TCP per il layer di rete.

### PROBLEMI

* NAMING = trovo la controparte usando RMIregistry
* ACCESS POINT = accedo alla controparte usando RMIregistry
* PROTOCOLLI = comunico usando RMI-IIOP
* FORMATO DATI/SEMANTICA = si scambiano oggetti java/reference serializzabili

**Qual è la semantica in un programma Java e nello scambio di messaggio?**

Tipizzazione: so che dati contiene, che operazioni posso fare… Stabilisce il formato dei dati e la loro semantica.