



Reti di calcolatori

► 0.0 - Info

▼ 1.0 - Integrazione di sistemi e uso oculato delle risorse

Introduzione

Tipologie di programmi

Esistono 2 paradigmi ispiratori dell'espressione dei programmi:

- Programmazione imperativa

Le istruzioni specificano l'algoritmo e i dati senza "libertà" di cercare soluzioni non specificate. Linguaggi come C, Java, C++ sono tutti di tipo imperativo, e per questo motivo usano stato, inteso come variabili e loro tipo, che viene manipolato dall'algoritmo passo passo attraverso le istruzioni.

- Programmazione dichiarativa

Le soluzioni devono soddisfare una serie di requisiti specificati dall'utilizzatore, senza arrivare alla specifica completa dell'algoritmo. La specifica esecuzione viene lasciata alla libertà di supporto del motore di base.

In questo corso non ci occuperemo di questo tipo di programmazione.

Possiamo distinguere inoltre i programmi in 2 categorie:

- Programmi in the small: reti di dimensioni limitate e con un uso limitato di risorse.
- Programmi in the large: grandi dimensioni per soddisfare larghe requisiti di un'organizzazione.

Programmazione di sistema

Programmazione di sistema: accanto alla programmazione applicativa, per realizzare e ottimizzare il supporto di programmi che consentono di preparare l'ambiente e forniscono le funzionalità di servizio. Tali programmi devono essere ottimizzati poichè eseguono molte volte.

L'obiettivo del corso è quello di imparare uno stile ingegneristico nel progetto di piccole applicazioni che consenta la creazione di programmi semplici e che disponga di un uso ottimale delle risorse.

Parte statica e dinamica di un'applicazione

- Parte statica: design dell'algoritmo e codifica in un linguaggio di programmazione.
- Parte dinamica: stabilisce quali sono le risorse hardware e fisiche su cui poter eseguire. La scelta di tale architettura viene detta scelta di deployment.

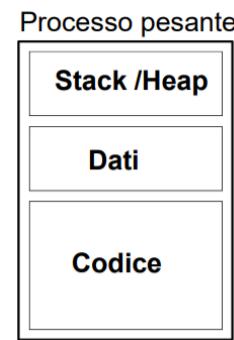
In questo corso viene data massima importanza a questa parte. L'obiettivo è quello di avere fasi dinamiche molto efficienti senza sprecare risorse.

Alcune entità non sono prevedibili a priori e possono cambiare locazione. Il binding assegna tali entità al programma.

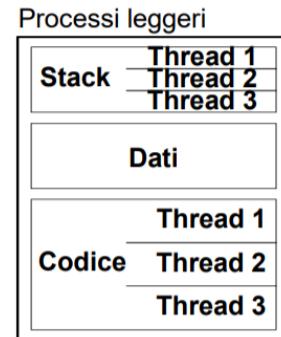
Standard di esecuzione

Per i sistemi operativi esiste uno standard di processo per operazioni primitive (API) con modello architettonico Unix (es. fork, exec, open, ...).

Per i processi esiste condiviso il modello a processi pesanti, mentre meno accettata è la specifica di processi leggeri. Purtroppo Unix non ha standardizzato processi leggeri, e tali processi richiedono molte risorse: ad esempio in UNIX il cambiamento di contesto è un'operazione molto pesante con overhead soprattutto per la parte di sistema.



Processi leggeri sono attività che condividono tra di loro la visibilità di un'ambiente contenitore (processo pesante) caratterizzate da uno stato limitato e a overhead limitato.



File

Un file è una sequenza di dati, di dimensione grandi a piacere.

Il file prevede un descrittore che ne dice la lunghezza, e ha un contenuto di byte senza contenere una fine del file (un carattere o una sequenza ad-hoc) al suo interno. I file possono essere generali (file binari) o limitati in contenuto, ad esempio i file testo sono file che contengono solo caratteri ASCII e fine linea '\n'.

Quando interagiamo con un file, in lettura, le azioni primitive sui file hanno il modo di farci capire che siamo arrivati alla fine del file e non possiamo leggere ulteriori caratteri, tramite risultati specifici che ci fanno capire che abbiamo raggiunto la fine del file (es. EOF, che non è un'eccezione, e viene definito come una costante negativa, solitamente -1).

Modello di lavoro con primitive

In generale, il nostro modello di lavoro è quello di effettuare le azioni richieste con l'ausilio del sistema di supporto, senza passare per librerie.

Il sistema per le azioni mette infatti a disposizione operazioni primitive sui file e dispositivi che sono visti come stream esterni ai linguaggi. Il modello di interazione è quello detto Open, Read/Write, Close, le quali primitive agiscono in modo atomico sugli stream e non in modo mediato da una libreria.

Semplicità delle strutture dati

Per realizzare programmi di sistema efficienti occorre fare attenzione a usare:

- Strutture statiche

Le variabili in memoria non dinamica sono predefinite e non hanno costi aggiuntivi durante l'esecuzione dati dall'allocazione dinamica delle aree sullo stack e dall'heap.

- Strutture dati limitate

Le variabili con memoria limitata (es. array) non hanno costi in eccesso per memoria non usata (es. linked list).

- Strutture dati semplici e poco astratte

Le strutture dati semplici (es. array) non hanno capacità espressiva non richiesta.

Semplicità degli algoritmi

- Codice in linea

In generale, e più specificatamente, se procedure e funzioni sono invocate una volta sola, fare codice in linea (non passare attraverso catene di invocazioni di procedure innestate, se si può evitare).

- Strutture con controllo semplici

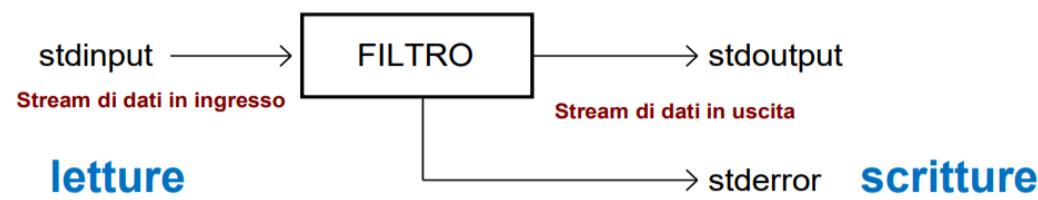
Usare sequenze, alternative e ripetizioni semplici in modo ordinato ed evitare i break. Si faccia uso di cicli while.

- Definizione singola dei dati

Le strutture dati devono essere allocate una volta per tutte e non nei cicli per evitare costi aggiuntivi e non necessari.

Filtri

Un filtro è un modello che prevede un programma che riceve in ingresso da un input e produce risultati su output (uno o più).



Il filtro deve consumare tutto lo stream in ingresso e portare in uscita il contenuto filtrato.

C come linguaggio di sistema

Input e Output in C

C non definisce l'input/output ma lo prende dal sistema operativo, che prevede una gestione integrata di I/O e dell'accesso ai file.

Per I/O, le azioni di base del Sistema Operativo sono le primitive di accesso `read()` / `write()` con semantica di atomicità (ossia di mutua esclusione e di non interruzione):

- `int read(int fd, char *buff, int len)`

Legge i caratteri e ritorna il numero di caratteri letti. In caso di errore, restituisce un valore negativo. In caso di fine file restituisce 0.

▼ Esempio

Programma che usa un buffer per il trasferimento di un file:

```
#define dim 100

int nread, fdsorg, dimbuff, sd;
char buff[dim];

if((fdsorg = open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, 00640)) < 0) {
    perror("Error opening file \n");
    exit(1);
}

while((nread = read(fdsorg, buff, dimBuffer)) > 0)
    write(sd, buff, nread);
```

- `int write(int fd, char *buff, int len)`

Scrive i caratteri e ritorna il numero di caratteri scritti. In caso di errore, restituisce un valore negativo.

Altre primitive per operare sui file sono le seguenti:

- `int open(const char *pathname, int flags)`

Apre il file specificato e restituisce il suo file descriptor. Possibili diversi flag di apertura, combinabili con OR bit a bit (|), ad esempio:

- `O_RDONLY` : apertura in sola lettura.
- `O_WRONLY` : apertura in sola scrittura.
- `O_RDWR` : apertura in lettura e scrittura.
- `O_CREAT` : crea il file se non esiste (richiede il terzo argomento `mode`).
- `O_TRUNC` : trunca il file a 0 se esiste ed è aperto in scrittura.
- `O_APPEND` : scrive i dati alla fine del file.

- `int close(int fd)`

Chiude un file descriptor precedentemente aperto. Ritorna 0 se la chiusura ha successo, 1 in caso di errore.

- `int lseek(int fd, int offset, int whence)`

Sposta il file offset associato a un file descriptor. Whence specifica la posizione di riferimento per calcolare il nuovo offset. Può assumere i seguenti valori:

- `SEEK_SET` : rispetto all'inizio del file.
- `SEEK_CUR` : rispetto alla posizione corrente del file.
- `SEEK_END` : rispetto alla fine del file.

Per I/O, è possibile utilizzare anche comandi da libreria, come:

- Input/Output a caratteri `getchar()`, `puchar()`
Restituiscono EOF in caso di end-of-file o errore.
- Input/Output a stringhe di caratteri `gets()`, `puts()`
Restituiscono NULL in caso di end-of-file o errore.
- Input/Output con formato per tipi diversi `printf()`, `scanf()`

▼ Formati:

- signed int: %d %hd %ld
- unsigned int: %u (decimale) %hu %lu
- ottale: %o %ho %lo
- esadecimale: %x %hx %lx
- float: %e %f %g
- double: %le %lf %lg
- carattere singolo: %c
- stringa di caratteri: %s
- puntatori (indirizzi): %p

Scarf restituisce il numero di valori validi letti in caso di successo, EOF in caso di lettura di end-of-file o errore. Inoltre, in caso di errore di formato di lettura, viene consumato lo stream di input fino al valore sbagliato, che deve essere quindi consumato (con `getchar()` o `gets()`).

▼ Esempio

Programma che chiede un intero all'utente in modo ripetuto fino a EOF.

```
int ok;
char ch;

while ((ok = scanf("%d", &ch)) != EOF) {
    if (ok != 1) { // errore di formato – puntatore scanf bloccato
        // consuma l'input
        do {c=getchar(); printf("%c ", c);}
        while (c!= '\n');
        printf("Inserisci un int), EOF per terminare: ");
        continue;
    }
    // opera sull'intero
}
```

- Input/Output con strutture FILE `fopen()`, `fclose()`

NON USARE `char* fgets(char *s, int #car, FILE *f)` e `char* fgets(char *s, int #car, stdin)`. Infatti le azioni mediate attraverso le strutture dai FILE sono meno dirette e agiscono appunto attraverso la libreria del sistema operativo.

Stringhe in C

In C è preferibile lavorare con stringhe come sequenze di caratteri (`char *` e terminate del carattere '`\0`').

- `int strlen(const char *s)` : lunghezza della stringa.

- `char *strcat(char *dest, const char *src)`: aggiunge la stringa src alla fine della stringa dest.
- `int strcmp(const char *s1, const char *s2)`: confronta due stringhe s1 e s2 e restituisce un valore che indica se sono uguali (0) o quale è maggiore.
- `int strncmp(const char *s1, const char *s2, int n)`: uguale a strcmp ma controlla solo i primi n caratteri delle due stringhe.
- `char *strcpy(char *s1, const char *s2)`: copia il contenuto della stringa s2 nella stringa s1.
- `char *strstr(const char *ss, const char *s)`: restituisce un puntatore della prima occorrenza della stringa s nella stringa ss, altrimenti null.

Controllo argomenti

È importante anche il controllo degli argomenti per non avere incongruenze durante l'esecuzione di un programma.

Ad esempio, per controllare se un argomento è un intero si utilizza il seguente codice:

```
int num, dimBuffer = 0;

// controllo numero argomenti
if(argc != 2) {
    printf("Usage Error: %s DimBuffer\n", argv[0]);
    exit(1);
}

// controllo secondo parametro int
while(argv[1][num] != '\0') {
    if ((argv[1][num] < '0') || (argv[1][num] > '9')) {
        printf("Secondo argomento non intero\n");
        printf("Usage Error: %s DimBuffer\n", argv[0]);
        exit(2);
    }
    num++;
}
dimBuffer = atoi(argv[1]);
```

Java come linguaggio di sistema

Alcune informazioni su Java:

- Java non rende visibili gli indirizzi anche se la JVM ha una semantica per riferimento tra oggetti in una JVM.
- Java non prevede strutture dati, ma solo classi che possono contenere variabili e metodi.
- Le istanze, classi e interfacce, sono processi leggeri.
- Un'istanza contiene le variabili descritte dalla classe e riferisce i metodi nella classe. Vengono utilizzati i valori per i dati di tipo primitivi e dei riferimenti per gli altri tipi di oggetti. Gli oggetti non sono contenuti dentro gli oggetti ma si puntano tra loro creando un grafo per ogni oggetto.
- La JVM gestisce un unico processo pesante che contiene al suo interno thread con stack separati che condividono i dati della JVM. L'heap della JVM ospita tutti gli oggetti, che sono allocati (solo quando effettivamente necessari alla esecuzione) e deallocati dinamicamente. Gli oggetti presenti nello stack puntano a quelli presenti nell'heap.
- Tra classi l'ereditarietà è a singolo genitore, mentre tramite interfacce si può sfruttare l'ereditarietà multipla.

Input e Output in Java

Per l'input si può usare `BufferedReader`, utilizzato per leggere i dati da flussi di input testuale in modo efficiente, il quale costruttore prende come argomento un flusso di input a caratteri (`Reader`) come `FileReader` o `InputStreamReader`.

```
BufferedReader reader = new BufferedReader(
    new FileReader("file.txt")
);
BufferedReader reader = new BufferedReader(
    new InputStreamReader(System.in)
);
```

```

);
BufferedReader reader = new BufferedReader(
    new InputStreamReader(socket.getInputStream())
);

```

Sul bufferedReader possono essere poi richiamati metodi come `readLine()` (restituisce `null` per EOF), che legge un'intera riga alla volta, e `read()`, che legge un carattere alla volta (restituisce -1 per EOF).

È possibile anche utilizzare `BufferedInputStream` per leggere dati da flussi di input a byte in modo efficiente. Il costruttore in questo caso prende come argomento un flusso di input a byte (`InputStream`) come `FileInputStream` o `DataInputStream`.

```

BufferedInputStream inputStream = new BufferedInputStream(
    new FileInputStream("file.txt");
);
BufferedInputStream inputStream = new BufferedInputStream(
    new DataInputStream(socket.getInputStream());
);

```

Per quanto riguarda le classi Buffered, queste sono disponibili anche per quanto riguarda l'output (`BufferedWriter` presenta i metodi `write()` e `newLine()`). Inoltre, per l'output si può anche utilizzare la classe `PrintWriter`, che consente di scrivere dati in un flusso di output in modo semplice e formattato.

```

PrintWriter writer = new PrintWriter("file.txt")
PrintWriter writer = new PrintWriter(System.out);

```

Su un oggetto PrintWriter possono essere richiamati i metodi `print()`, `println()` e `printf()`.

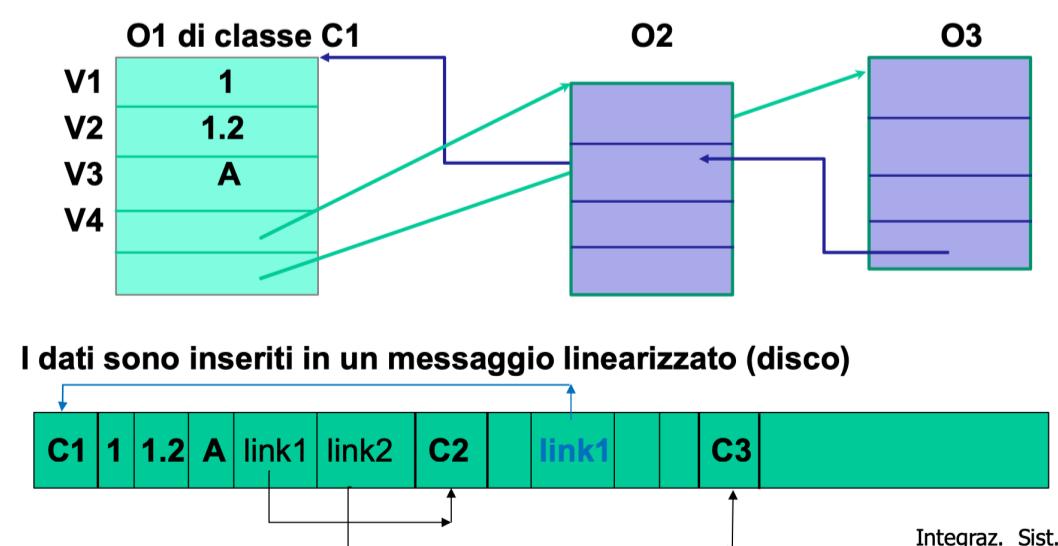
BufferedReader, PrintWriter e BufferedWriter devono essere chiusi tramite il metodo `close()` al termine del loro utilizzo per assicurarsi che tutte le risorse associate vengano rilasciate e che le operazioni di I/O vengano completate correttamente.

Per l'invio e la ricezione di tipi primitivi tra JVM diverse vengono invece utilizzati oggetti di tipo `DataOutputStream` e `DataInputStream`.

	DataOutputStream	DataInputStream
<code>String</code>	<code>void writeUTF(String str)</code>	<code>String readUTF()</code>
<code>char</code>	<code>void writeChar(int v)</code>	<code>char readChar()</code>
<code>int</code>	<code>void writeInt(int v)</code>	<code>int readInt()</code>
<code>float</code>	<code>void writeFloat(float v)</code>	<code>float readFloat()</code>
...

Esternalizzazione da JVM a out

I dati contenuti in un oggetto Java possono essere usati in Java ma hanno poco senso nel mondo esterno, fatto di byte e memoria e risorse fisiche e non filtrati attraverso JVM. Per questo motivo, per esportare il contenuto di un oggetto su disco occorre serializzarlo, ovvero portarlo in un formato a byte visibile all'esterno, salvabile sul disco, ed eventualmente ripristinabile nella macchina virtuale.



È possibile utilizzare la serializzazione offerta direttamente a livello di supporto al linguaggio implementando l'interfaccia `Serializable`. In questo modo è possibile utilizzare i metodi `writeObject()` e `readObject()` per serializzare e deserializzare. Esempio:

```
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();
```

▼ 2.0 - Generalità, obiettivi e modelli di base

Oggetto del corso

Realizzazione di sistemi distinti in località diverse che usano la comunicazione e la cooperazione per ottenere risultati coordinati da mostrare all'utente.

Protocolli

Per un'applicazione distribuita c'è bisogno della definizione e dell'utilizzo di protocolli per ottenere azioni coordinate su più nodi. Un protocollo è un complesso di regole e procedure cui ci si deve attenere in determinate attività per la esecuzione corretta.

Nella comunicazione tra processi i protocolli possono essere di due tipi:

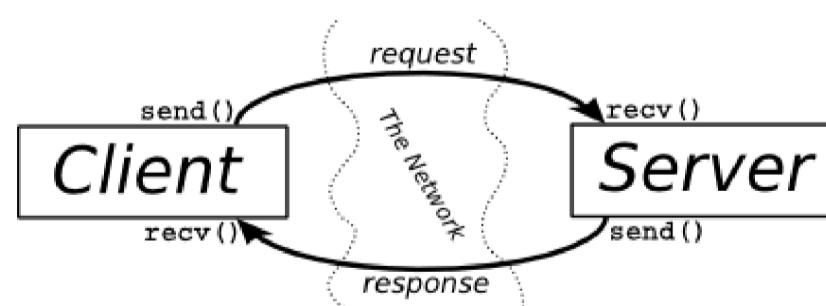
- Cliente servitore: prevede un fruitore e un fornitore dello stesso servizio.
- Scambio di messaggi: si ha un mittente che manda informazioni ad uno o più riceventi senza prevedere risposta.

Entrambi possono realizzare qualunque tipo di programma nel distribuito, ma differiscono nelle capacità espressive. Le principali differenze sono le seguenti:

Client Server	Scambio di Messaggi
sincrono	asincrono
bloccante	non bloccante
comunicazione diretta	comunicazione indiretta
singolo ricevente	Riceventi molteplici

Modello cliente servitore base

Si può implementare un protocollo Client/Server con due scambi di messaggi e alcune regole di protocollo per i processi interagenti. Considerando due API locali `send()` e `receive()` si ha:



Il modello cliente servitore può avere implementazioni molto varie, noi lo utilizzeremo sempre con le seguenti caratteristiche:

- Molti a uno: 1 servitore, più clienti.
- Sincrono: si prevedere la risposta del servitore al cliente.
- Bloccante: il cliente aspetta la risposta del servitore.

- Asimmetrico: il cliente conosce il servitore per inviare la invocazione, mentre il servitore non conosce a priori i clienti possibili.
- Dinamico: può cambiare il servitore che risponde alle richieste tra diverse invocazioni.

Inoltre, il server deve realizzare il servizio con azioni locali, come accedere alle risorse del sistema, considerando molteplici clienti e quindi problemi di integrità dei dati, accessi concorrenti, autenticazione utenti, autorizzazione all'accesso e privacy delle informazioni. Nel modello base trascuriamo però questi problemi.

Modello di interazione push e pull

Se la risposta del server non arriva non si può assumere che questo sia down, ma può essere che sia congestionato da tante richieste. Il cliente però non può aspettare all'infinito una risposta, per questo un modello possibile è quello impostare un intervallo predeterminato detto timeout, dopo il quale scatta una eccezione. L'eccezione consente poi di ripetere la richiesta, cambiare server, chiudere tutto o riportare l'errore all'utente.

Oltre al modello appena presentato per la gestione della congestione del server, detto pull, esiste un altro modello di interazione detto push, nel quale il cliente fa la richiesta una volta sola per poi sbloccarsi. Nel mentre il servitore ha il compito di svolgere il servizio richiesto e consegnare il risultato al cliente.

Server sequenziale o parallelo

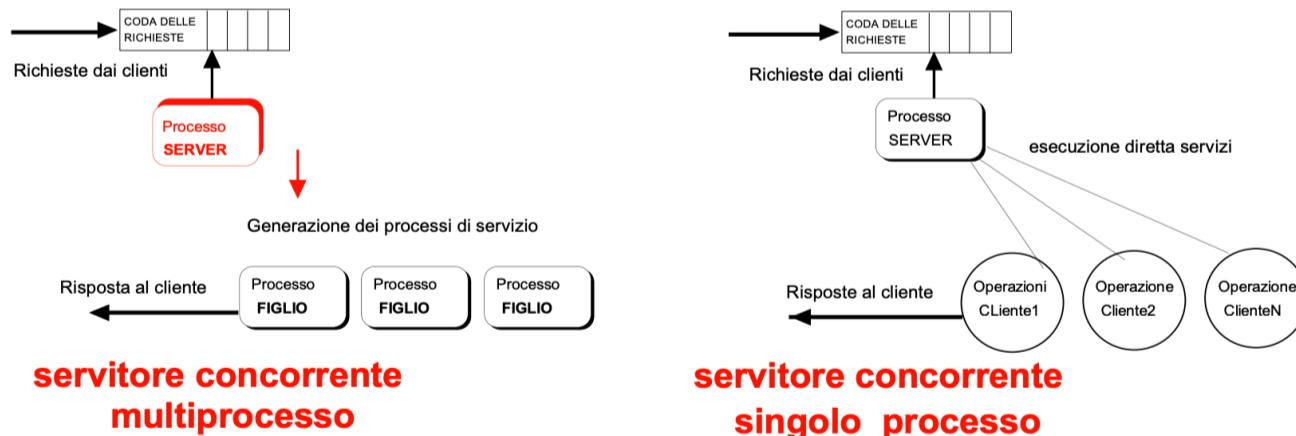
Il server dura all'infinito e viene dunque definito processo demone. Java riconosce thread daemon rispetto a thread user e li esegue per tutta la durata di una sessione della virtual machine (JVM), terminandone l'esecuzione solo quando termina l'ultimo user thread.

Per questo motivo, ricevendo più richieste da più clienti, un servitore si può differenziare in sequenziale e parallelo. Entrambi mantengono una coda delle richieste da servire, ma mentre il servitore sequenziale svolge un'operazione alla volta quello parallelo può lavorare su più richieste insieme.

La concorrenza può ridurre significativamente il tempo medio di risposta del server nel caso in cui la risposta richiede un tempo di attesa significativo di I/O, se le richieste richiedono tempi di elaborazione molto variabili e se il server è eseguito in un multiprocessore.

Un servitore concorrente può seguire diversi schemi:

- Servitore concorrente multiprocesso: un processo server si occupa della coda delle richieste e genera processi figli, uno per ogni servizio.
- Servitore concorrente monoprocesso: un unico processo server si divide tra il servizio della coda delle richieste e le operazioni vere e proprie. In questo caso non c'è il costo di generazione dei processi figli.



Modello a eventi e scambio di messaggi

Oltre al modello cliente servitore esiste un modello asincrono e fortemente disaccoppiato, ovvero nel quale non viene imposta la compresenza delle parti.

È un modello molti a molti nel quale i consumatori si registrano ad un gestore, i produttori segnalano gli eventi al gestore e quest'ultimo effettua push ai consumatori.

Modello con connessione e senza connessione

Nella interazione C/S si considerano due tipi principali riguardo all'insieme delle richieste:

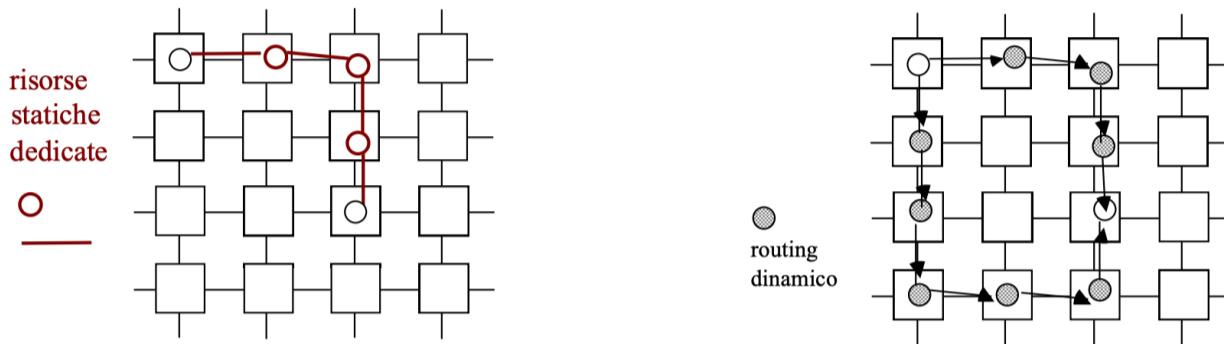
- Interazione connection-oriented (con connessione): si stabilisce un canale di comunicazione virtuale prima di iniziare lo scambio dei dati. Le richieste arrivano in ordine di emissione del cliente.

- Interazione connection-less (senza connessione): senza connessione virtuale, ma semplice scambio di messaggi isolati tra loro. Ogni richiesta arriva in modo indipendente (fuori ordine).

La scelta tra le due forme dipende dal tipo di applicazione e anche da vincoli imposti dal livello di comunicazione sottostante. Per esempio, in Internet il livello di trasporto prevede i protocolli TCP e UDP. TCP è un protocollo con connessione, affidabile e che preserva l'ordine di invio dei messaggi. UDP invece è senza connessione, non affidabile e non preserva l'ordine messaggi. Al di sotto di entrambi i protocolli c'è però sempre IP, il quale è connectionless e best effort.

Tale differenza c'è anche nell'interconnessione fisica (livello rete):

- OSI, connessione: tutti i messaggi seguono la stessa strada per la coppia mittente destinatario decise staticamente e impegnano risorse nei nodi interessati e nei nodi intermedi predeterminati.
- IP, senza connessione: i messaggi possono seguire strade diverse decise dinamicamente e non impegnano staticamente risorse intermedie.



Alcuni modelli a connessione TCP basato su IP, non vengono impegnate risorse intermedie ma solo sul mittente/destinatario.

Stato nel cliente servitore

Nella interazione C/S, un aspetto centrale è lo stato dell'interazione, ossia che si tenga traccia della comunicazione e delle azioni precedenti.

Nelle interazioni stateless non si tiene traccia dello stato, ogni messaggio è completamente indipendente dagli altri, in quelle stateful invece si mantiene lo stato dell'interazione tra chi interagisce, dunque un messaggio può dipendere da quelli precedenti.

Un Server stateful garantisce efficienza in quanto le dimensioni dei messaggi sono più contenute e si ottiene una migliore velocità di risposta del Server. Un Server stateless invece è più leggero e affidabile in presenza di malfunzionamenti, ma lo stato deve essere mantenuto da ogni cliente → stateful: client più leggero, server più pesante; stateless: client più pesante, server più leggero.

Per funzionare, un server stateful deve potere identificare il Client e tenerne traccia per interazioni future. Lo stato si distingue in due tipologie in base alla durata massima:

- Stato permanente: mantenuto per sempre.
- Stato soft o a tempo: rimane per un tempo massimo.

In caso di progetti stateless, il protocollo è corretto e il progetto del servitore è molto semplificato se le operazioni sono idempotenti, ovvero che, se eseguite più volte con lo stesso input, producono sempre lo stesso risultato.

Progetto servitore

In base alle caratteristiche viste, un servitore può distinguersi nelle seguenti categorie:

		Tipo di comunicazione	
		connessione	senza connessione
S E R V E R	sequenziale iterativo		
	concorrente singolo processo		La scelta del tipo di Server dipende dalle caratteristiche del servizio da fornire
	concorrente multi processo		

Lo stato ha effetto sul protocollo e sulle entità

Modello ad agenti multipli

Oltre ai modelli già visti è possibile utilizzare un modello in cui i servizi richiesti dai clienti vengono forniti tramite il coordinamento di più servitori detti agenti in modo trasparente al cliente.

In questo caso anche i servitori, tra loro, devono utilizzare protocolli di coordinamento, sincronizzazione, rilevazione e tolleranza ai guasti.

Semantiche

Semantiche della comunicazione

Esistono diverse semantiche della comunicazione:

- MAY-BE (o BEST-EFFORT): per limitare i costi ci si basa su un solo invio asincrono di ogni datagramma/informazione, il messaggio può arrivare o meno.
In UDP viene utilizzata questa semantica.
- AT-LEAST-ONCE: ritrasmissioni ad intervallo da parte del mittente, il messaggio può arrivare anche più volte a causa dei messaggi duplicati dovuti alle ritrasmissioni e il server non se ne accorge.
- AT-MOST-ONCE: cliente e servitore lavorano in modo coordinato per ottenere garanzie di correttezza e affidabilità. Il messaggio, se arriva, viene considerato al più una volta dal servitore, il quale mantiene uno stato per riconoscere i messaggi già ricevuti e per non eseguire azioni più di una volta.
In TCP viene utilizzata questa semantica.
- EXACTLY-ONCE: il messaggio o è arrivato (una volta sola) o non è stato considerato da entrambi. Entrambi i pari sanno se l'operazione è stata fatta o meno.

Semantica di trasporto

Per quanto riguarda la semantica di trasporto:

- UDP è un servizio senza connessione
- TCP è un servizio con connessione.

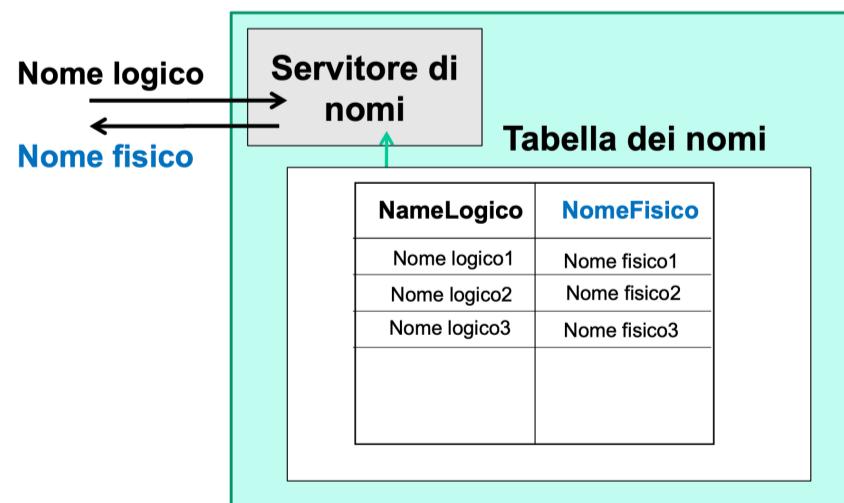
Consente una connessione bidirezionale tra gli endpoint, offre un controllo di flusso a byte (byte in ordine corretto e ritrasmissione dei messaggi persi) e con bufferizzazione.

Modelli e sistemi di nomi

Per accedere a servizi è necessario poterli identificare e fare binding (legame tra la risorsa logica, nome, e la risorsa fisica, target) alle risorse disponibili. Questa funzione viene svolta dai sistemi di nomi, ossia servitori tipicamente capaci di fornire servizi di mantenimento e di gestione dei nomi.

In generale nei sistemi distribuiti i clienti possono fare richieste al servizio di nomi per ottenere il nome del servitore di interesse e poterlo poi riferire. Affinchè il sistema di nomi conosca il nome del servitore quest'ultimo deve registrarsi.

Il sistema di nomi contiene solitamente al suo interno una tabella di corrispondenze che mappa un nome logico ad un nome fisico.



Server di nomi in-the-large

Siccome un unico server che funge da sistema di nomi non può registrare tutti i server del mondo per questione di memoria e di esecuzione, sono necessari progetti più ampi, con servizi che usano modalità ad agenti multipli o simili.

È possibile fare ciò utilizzando:

- Gestori partizionati: una molteplicità di gestori ciascuno responsabile di una sola parte dei riferimenti.
- Gestori replicati: una molteplicità di gestori ciascuno responsabile degli stessi riferimenti, in maniera tale da far fronte ai guasti.

I gestori devono essere organizzati tra loro in architetture come ad esempio in un albero (es. DNS).

Binding statico e dinamico

La risoluzione (binding) dei nomi può essere:

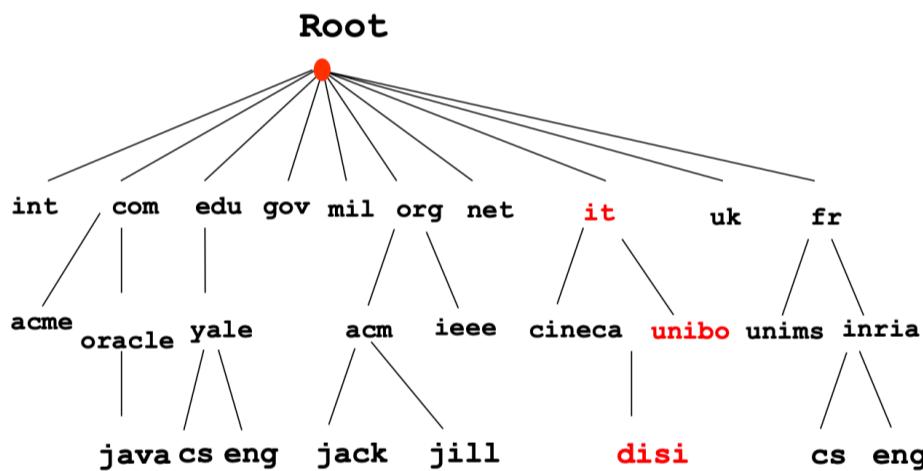
- Statico: i nomi sono risolti prima dell'esecuzione e non è consentito cambiare alcuna allocazione. Si risolve dunque tutto staticamente e non si necessita di un servizio di nomi.
- Dinamico: i nomi sono risolti durante l'esecuzione in quanto le risorse sono dinamiche e dunque non prevedibili staticamente. Tipico dei sistemi distribuiti, se le entità cambiano allocazione, non cambiano i nomi. Si necessita di un servizio di nomi.

DNS

Il DNS è un sistema di nomi globale in Internet basato su un insieme di gestori DNS coordinati che si organizzano per rispondere a query che chiedono il nome di IP corrispondente ad un nome di dominio.

L'insieme dei gestore del DNS sono organizzati in una gerarchia ad albero, in cui al vertice c'è il root server, e andando più in basso nella gerarchia troviamo i domini di livello inferiore.

Ogni organizzazione mantiene solitamente uno o più gestori DNS per migliorare la velocità delle risposte. In questo modo, quando un utente richiede la risoluzione di un nome di dominio, il sistema DNS cerca nella tabella locale del gestore DNS associato all'organizzazione. Se il dominio è presente, la risposta è veloce, altrimenti il gestore deve inviare la richiesta ad altri server, solitamente al gestore di livello superiore nella gerarchia, rendendo il tutto meno efficiente.



Ogni gestore effettua solitamente caching per migliorare le prestazioni senza effettuare ogni volta la richiesta dello stesso nome fisico allo stesso server. Ogni gestore sfrutta anche la replicazione al fine di fornire risposte anche in caso di problemi.

I singoli nomi logici utilizzati nei DNS presentano più domini (primo livello, secondo livello ecc.) e presentano le seguenti caratteristiche: sono case insensitive, max 63 char per dominio, il nome completo è lungo al max 255 char.

Il protocollo DNS utilizza due tipi principali di query per risolvere i nomi di dominio in indirizzi IP:

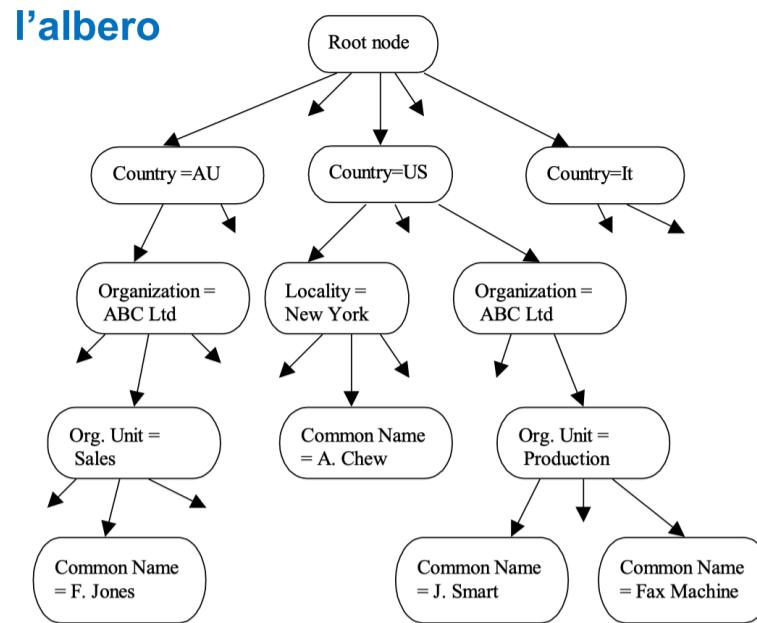
- Query ricorsiva: il server se possiede il nome risponde, altrimenti cerca altre risposte e rimane impegnato fino a risposta o time-out.
- Query iterativa: il server se possiede il nome risponde, altrimenti risponde con un riferimento al gestore più vicino che possa ragionevolmente rispondere (suggerimento).

Sono possibili anche query inverse, ovvero si entra con un numero IP e ci si aspetta un nome logico. Non tutti i server consentono però queste query.

X.500

Un altro sistema di nome è X.500. Questo è un insieme di nodi organizzati in un albero di interconnessione in cui ogni nodo è costituito da attributi tipizzati che possono assumere valori (es. età = intero) e che aggiungi attributi rispetto al genitore.

Ogni gerarchia ha dunque attributi in parte condivisi (quelli presenti nel nodo padre) e poi differenziati da ogni nodo.



La novità sta nell'organizzazione basata sui contenuti e le ricerche che si possono fare in modo molto flessibile per singole entità o anche per attributo, ritrovandosi gruppi di elementi anche molto numerosi (interi sottoalberi e foreste). Sono inoltre possibili condizioni logiche sugli attributi (es. CN=name AND C=country) anche con espressioni regolari e condizioni aritmetiche.

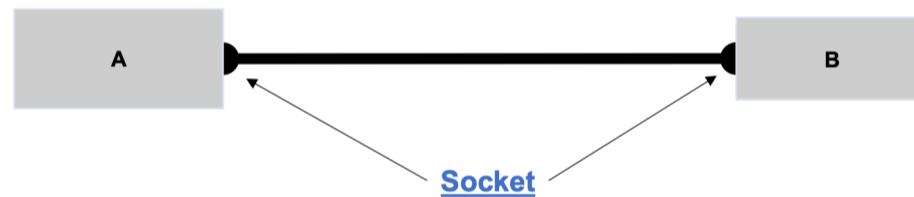
▼ 3.0 - Progetto Client/Server con Socket

▼ 3.1 - Progetto C/S con Socket in Java

Introduzione alle Socket

Vogliamo realizzare un C/S attraverso strumenti di comunicazione standard di scambio messaggi, le Socket, le quali si basano sulle primitive send e receive.

Le socket rappresentano il terminale locale (end point) di un canale di comunicazione bidirezionale.



Per far comunicare due macchine diverse in Java è possibile programmare la rete attraverso meccanismi di visibilità della comunicazione contenuti in classi specifiche del package di networking `java.net`. In base al modello di comunicazione si possono usare due tipologie di socket differenti:

- Con connessione (protocollo TCP) → Socket Stream (classe `Socket` lato client, `ServerSocket` lato server).
- Senza Connessione (protocollo UDP) → Socket Datagram (classe `DatagramSocket` lato client e server).

Sistema di nomi

Il primo problema da affrontare riguarda la identificazione reciproca dei processi (Client o Server) nella rete, i cui nomi hanno solo validità locale.

Ogni processo locale deve essere associato ad un nome globale, visibile in modo univoco, non ambiguo, e semplice che altri possano usare per raggiungerlo. Tale nome globale è composto da due componenti:

- Indirizzo IP (4 byte) → livello IP.
- Porta (numero intero di 16 bit) → livello TCP e UDP.



I messaggi sono consegnati su una specifica porta di una specifica macchina, e non direttamente a un processo. È la socket dunque che riceve direttamente il messaggio, in quanto è lei che lega il processo ad un nome globale.

Socket Datagram

DatagramSocket

La classe `java.net.DatagramSocket` consente a due thread di scambiarsi messaggi senza stabilire una connessione tra i thread coinvolti.

Uno dei costruttori prevede:

```
DatagramSocket(InetAddress localAddress, int localPort) throws SocketException
```

Su una istanza `DatagramSocket` si fanno azioni di:

- `void send(DatagramPacket p)`: invia un messaggio al suo supporto locale.
- `void receive(DatagramPacket p)`: aspetta fino a ricevere il primo datagramma disponibile.

Siccome tale metodo è sincrono bloccante, è possibile richiamare sulla classe `DatagramSocket` il metodo `void setSoTimeout(int millis)`, il quale consente di impostare un timeout dopo il quale l'operazione termina lanciando un'eccezione da gestire.

- `void close()`: chiude la socket.
- `void setSendBufferSize(int size)` e `void setReceiveBufferSize(int size)`: variano il buffer di invio e ricezione.
- `void setReuseAddress(boolean on)`: consente di specificare se la socket può essere associata a un indirizzo e porta già in uso da un'altra socket.

Tramite le socket datagram il mittente deve specificare nel messaggio un ricevente, mentre il ricevente riceve solo il primo messaggio che è in coda.

DatagramPacket

`DatagramPacket` è la classe che si utilizza per effettuare le operazioni di `send` e `receive`, in quanto consente di preparare e usare datagrammi che specificano cosa comunicare e con chi:

- Parte controllo: `InetAddress` e un intero per la porta.
- Parte dati: specifica un array di byte da/su cui scrivere.

La classe `InetAddress` consente di specificare un indirizzo IP e presenta solo metodi statici:

- `public static InetAddress getByName(String hostname)`
fornisce un oggetto `InetAddress` per l'host (nome logico) specificato.
- `public static InetAddress[] getAllByName(String hostname)`
fornisce un array di oggetti `InetAddress` per più indirizzi IP sullo stesso nome logico.
- `public static InetAddress getLocalHost()`
fornisce `InetAddress` per la macchina locale.

Questi metodi possono sollevare l'eccezione `UnknownHostException` se l'indirizzo specificato non può essere risolto.

Un costruttore per la classe `DatagramPacket` è il seguente:

```
DatagramPacket(  
    byte[] buf, // array di byte dati  
    int offset, // indirizzo inizio  
    int length, // lunghezza dati  
    InetAddress address, int port // indirizzo IP e porta  
)
```

Su un oggetto `DatagramPacket` si possono richiamare diversi metodi getter e setter:

```
InetAddress getAddress(), // ottiene indirizzo associato  
void setAddress(InetAddress addr) // cambia indirizzo  
int getPort(), // ottiene porta associata  
void setPort(int port) // cambia porta associata  
byte[] getData(), // estrae i dati dal pacchetto  
void setData(byte[] buf), // inserisce i dati nel pacchetto
```

Esempio

Un esempio di utilizzo di DatagramSocket è il seguente:

```
// invio
DatagramSocket socket = new DatagramSocket();

byte[] buf = {'C','i','a','o'};
InetAddress remoteAddr = InetAddress.getByName("137.204.59.72");
int remotePort = 1900;
DatagramPacket packet =
    new DatagramPacket(buf, buf.length, remoteAddr, remotePort);

socket.send(packet);

// ricezione
InetAddress addr = InetAddress.getByName("137.204.59.72");
int port = 1900;
DatagramSocket socket = new DatagramSocket(addr, port);

byte[] buf = new byte[200];
DatagramPacket packet = new DatagramPacket(buf, buf.length);

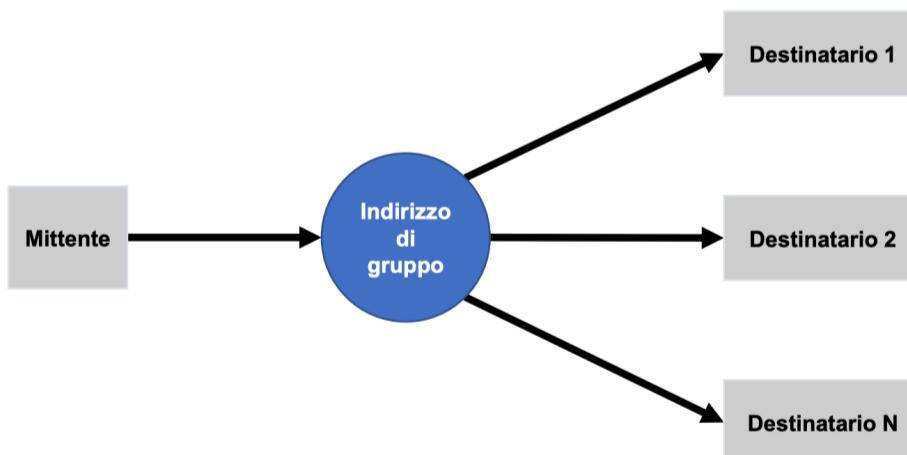
socket.receive(packet);

int remoteAddr = packet.getAddress(); // indirizzo del mittente
int remotePort = packet.getPort(); // porta del mittente
// lavora con i dati presenti in buf
```

Comunicazione multicast

La comunicazione a datagrammi consente anche l'uso di protocolli di multicast, potendo così inviare messaggi a una serie di destinatari che siano registrati su un indirizzo di gruppo.

Ricordiamo che gli indirizzi di gruppo sono di classe D e vanno dal 224.0.0.0 al 239.255.255.255.



Per fare ciò si utilizza un'ulteriore classe `MulticastSocket` che consente di gestire gruppi e ricevere messaggi multicast:

```
InetAddress groupAddr = InetAddress.getByName("229.5.6.7");
MulticastSocket multSocket = new MulticastSocket(multicastPort);

multSocket.joinGroup(gruppo); // ingresso nel gruppo
multSocket.leaveGroup(gruppo); // uscita dal gruppo
```

Si possono avere molti gruppi sullo stesso indirizzo IP di classe D, distinti dalla porta.

Esempio di invio e ricezione multicast:

```

// invio
MulticastSocket multSocket = new MulticastSocket();

byte[] buf = {'C','i','a','o'};
InetAddress groupAddr = InetAddress.getByName("229.5.6.7");
int groupPort = 6666;
DatagramPacket packet =
    new DatagramPacket(buf, buf.length, groupAddr, groupPort);

multSocket.send(packet);

// ricezione
int groupPort = 6666;
DatagramSocket multSocket = new DatagramSocket(groupPort);
InetAddress groupAddr = InetAddress.getByName("229.5.6.7");
multSocket.joinGroup(groupAddr);

byte[] buf = new byte[200];
DatagramPacket packet = new DatagramPacket(buf, buf.length);

multSocket.receive(packet);

int remoteAddr = packet.getAddress(); // indirizzo del mittente
int remotePort = packet.getPort(); // porta del mittente
// lavora con i dati presenti in buf
multSocket.leaveGroup(groupAddr);

```

Socket Stream

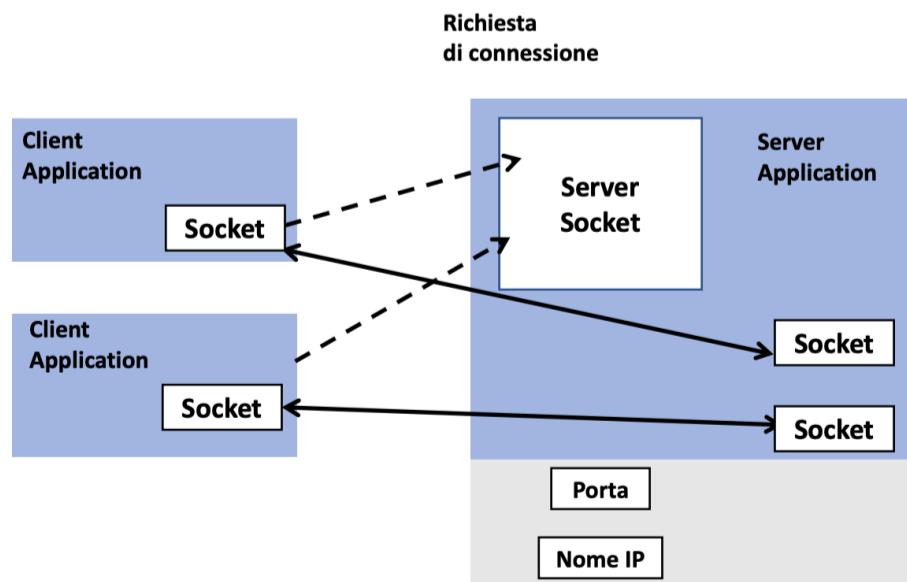
Le Socket Stream consentono di identificare i due terminali di un canale bidirezionale tra due entità che vogliono comunicare tra loro ripetutamente e a lungo (con connessione).

Il protocollo utilizzato è TCP. Si garantiscono ritrasmissioni al livello di trasporto, non visibili a livello applicativo. La semantica è at-most once, ovvero si riceve una volta sola anche se i dati vengono reinviati.

La connessione tra i processi Client e Server è definita da una quadrupla univoca

<indirizzo IP Client; porta Client; indirizzo IP Server; porta Server>

Java implementa le Socket Stream tramite due classi distinte per i ruoli di Client (`java.net.Socket`) e Server (`java.net.ServerSocket`).



Siccome il protocollo utilizzato è TCP e non UDP, e dunque i dati devono essere ricevuti nello stesso ordine in cui sono arrivati, per inviare e ricevere i dati non vengono forniti dei metodi send e receive come per la classe DatagramSocket, ma si utilizzano stream di input e output.

Socket

La classe `Socket` consente di creare una socket Stream (TCP) per il collegamento di un Client a un Server con canale TCP che permette una comunicazione dati bidirezionale (full duplex).

Un costruttore per creare una Socket stream lato cliente e collegarla a un indirizzo IP e a una porta data è il seguente:

```
public Socket(InetAddress remoteAddr, int remotePort) throws IOException
```

È possibile creare una Socket anche passando come parametro direttamente il nome logico dell'host remoto:

```
public Socket(String remoteHost, int remotePort) throws IOException
```

La costruzione delle Socket produce in modo atomico anche la connessione al server corrispondente o lancia l'eccezione.

Su un'istanza `Socket` si possono richiamare i seguenti metodi:

- `InputStream getInputStream()` : ritorna lo stream di input da cui leggere.
- `OutputStream getOutputStream()` : ritorna lo stream di output sul quale inserire i dati da inviare.
- `void close()` : chiude la socket. La chiusura è necessaria al fine di non impegnare troppe risorse di sistema.
- `InetAddress getInetAddress()` : restituisce l'indirizzo del nodo remoto a cui la socket è connessa.
- `int getPort()` : restituisce il numero di porta sul nodo remoto a cui la socket è connessa.
- `InetAddress getLocalAddress()` : restituisce l'indirizzo della macchina locale.
- `int getLocalPort()` : restituisce il numero di porta locale a cui la socket è legata.

Esempio di input e output tramite la classe `Socket`:

```
try {  
    socket = new Socket(hostname, 7);  
  
    socketOut = new PrintWriter(socket.getOutputStream(), true);  
    socketIn = new BufferedReader(  
        new InputStreamReader(socket.getInputStream())  
    );  
    userInput = new BufferedReader(  
        new InputStreamReader(System.in)  
    );  
  
    while((line = userInput.readLine()) != null) {  
        socketOut.println(line);  
        System.out.println(socketIn.readLine());  
    }  
  
    socket.close();  
} catch (IOException e) {System.err.println(e);}
```

ServerSocket

La classe `ServerSocket` lato server consente di definire una socket capace solo di accettare richieste da una coda di connessione provenienti da diversi client.

Un costruttore della classe `ServerSocket` è il seguente:

```
public ServerSocket(int localPort) throws IOException, BindException
```

il quale crea una socket in ascolto sulla porta specificata.

Un altro costruttore consente anche di specificare la lunghezza della coda:

```
public ServerSocket(int localPort, int length)
```

Richiamando il metodo `Socket accept()` su un oggetto di tipo `ServerSocket` si ottiene un oggetto `Socket` lato server per una specifica connessione e trasmissione dati. La connessione tra client e server viene dunque stabilita su iniziativa del

server dopo l'invocazione di accept. La chiamata di accept è sospensiva, in attesa di richieste di connessione.

La trasmissione dei dati avviene con i metodi visti per il lato client in modo del tutto indifferente.

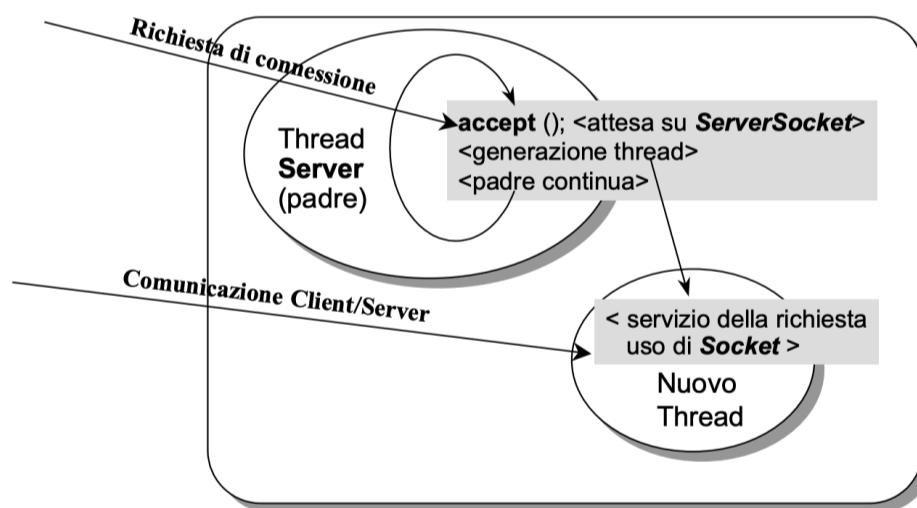
Esempio di utilizzo di una `ServerSocket` per stabilire connessioni e inviare dati:

```
try {  
    serverSocket = new ServerSocket(localPort);  
  
    while (true) { // il server invia la data al cliente  
        connSocket = serverSocket.accept();  
  
        out = new PrintWriter(connSocket.getOutputStream(), true);  
        Date now = new Date();  
        out.write(now.toString() + "\r\n");  
  
        connSocket.close();  
    }  
} catch (IOException e) {  
    oggConnessione.close();  
    oggServer.close();  
    System.err.println(e);  
}
```

Server parallelo

In caso di server parallelo all'accettazione di una nuova connessione il servitore può generare un nuovo thread responsabile del servizio che eredita la connessione nuova e lo chiude al termine dell'operazione.

In questo modo, una volta che il servitore genera un nuovo thread per gestire la richiesta può continuare immediatamente ad aspettare nuove richieste e servire nuove operazioni.



Chiusura Socket

Le socket in Java impegnano una serie di risorse di sistema necessarie per l'operatività fino alla `socket.close()`.

In caso di una socket chiusa, la memoria di input viene eliminata subito, mentre quella di output viene mantenuta per il tempo necessario per spedire tutte le informazioni al pari. Tramite il metodo `void SetSoLinger(boolean on, int lingerSec)` è anche possibile specificare un periodo di tempo in secondi dopo il quale si possono scartare i pacchetti in attesa di essere consegnati.

Il pari della connessione si accorge della chiusura della socket tramite eccezioni, predicati o eventi che gli vengono notificati in caso di operazioni di lettura o scrittura sulla socket chiusa dal pari.

È possibile utilizzare anche i metodi `void shutdownInput()` e `void shutdownOutput()` per chiudere un solo lato della connessione.

▼ Esempio di copia remota di un file tramite socket stream

Per un programma di questa tipologia è adatto l'uso di connessione in quanto la copia di un file richiede che i byte inviati arrivino tutti in ordine e una volta sola.

Il programma client ha la seguente invocazione:

```
rpc_client nodoserver portaserver nomefilesorg nomefiledest
```

Client:

```
socket = new Socket(host, port);

socketOut = new DataOutputStream(socket.getOutputStream());
socketIn = new DataInputStream(socket.getInputStream());

socketOut.writeUTF(nomeFileDest);
response = socketIn.readUTF();

if (response.equalsIgnoreCase("MSGsrv: attendo file") == true) {
    fileSorg = new File(nomeFileSorg);
    fileSorgInStream = new FileInputStream(fileSorg);
    int singleByte = 0;
    while ((singleByte = fileSorgInputStream.read()) >= 0)
        socketOut.write(singleByte);
} catch(IOException e) {}

socket.close();
```

Server sequenziale:

```
try {
    serverSocket = new ServerSocket(port);
    System.out.println("Attesa su porta" + serverSocket.getLocalPort());

    while(true) {
        connSocket = serverSocket.accept();
        System.out.println("conn" + connSocket);

        socketOut = new DataOutputStream(connSocket.getOutputStream());
        socketIn = new DataInputStream(connSocket.getInputStream());

        nomeFileDest = socketIn.readUTF();
        fileDest = new File(nomeFileDest);

        if(fileDest.exists() == true) {
            socketOut.writeUTF("MSGsrv: file presente, bye");
        } else {
            socketOut.writeUTF("MSGsrv: attendo file");
            fileDestOutStream = new FileOutputStream(fileDest);
            int singleByte = 0;
            while ((singleByte = socketIn.read()) >= 0)
                fileDestOutStream.write(singleByte);
        }
        connSocket.close();
    }
} catch (IOException e) {System.out.println(e);}
```

Server parallelo:

```
try {
    serverSocket = new ServerSocket(port);
    System.out.println("Attesa su porta" + serverSocket.getLocalPort());
```

```

        while(true) {
            connSocket = rcpSocket.accept();
            serviceThread = new RcpService(connSocket);
            serviceThread.start();
        }
    } catch (IOException e) {System.err.println(e);}

    public class RcpService extends Thread {
        Socket connSocket;

        public RcpService(Socket connSocket) {
            this.connSocket = connSocket;
        }

        public void run() {
            System.out.println("thread numero " + Thread.currentThread());
            System.out.println("Connesso con" + connSocket);

            try {
                socketOut = new DataOutputStream(connSocket.getOutputStream());
                socketIn = new DataInputStream(connSocket.getInputStream());

                nomeFileDest = socketIn.readUTF();
                fileDest = new File(nomeFileDest);
                if(fileDest.exists() == true) {
                    socketOut.writeUTF("MSGsrv: file presente, bye");
                } else {
                    socketOut.writeUTF("MSGsrv: attendo file");
                    fileDestOutStream = new FileOutputStream(fileDest);
                    int singleByte = 0;
                    while ((singleByte = socketIn.read()) >= 0)
                        fileDestOutStream.write(singleByte);
                }
                connSocket.close();
                System.out.println("Fine servizio thread numero " + Thread.currentThread());
            } catch (IOException e) {System.err.println(e);}
        }
    }
}

```

▼ 3.2 - Progetto C/S con Socket in C

▼ 3.2.1 - Introduzione alle Socket in C

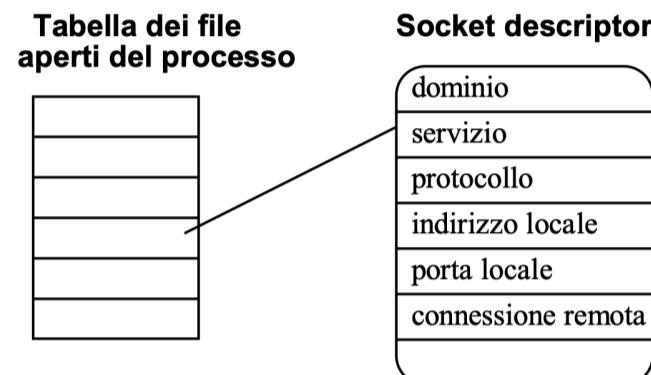
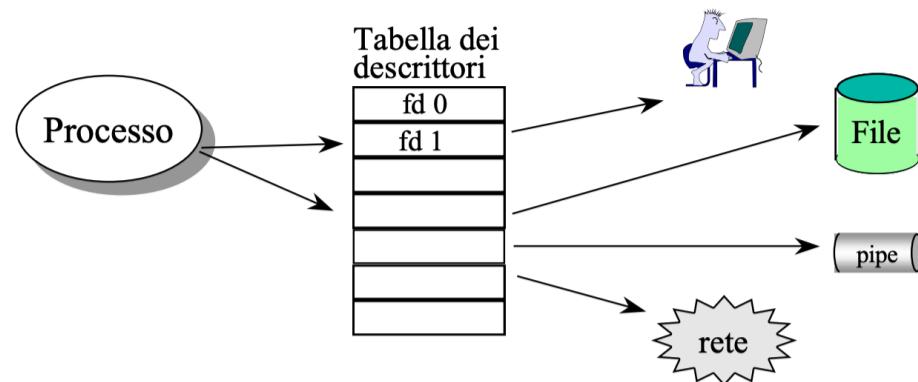
Unix: modello di uso

In Unix le operazioni tramite socket sono conformi al paradigma utilizzato per comunicare localmente tramite file. Ogni processo mantiene infatti una tabella di kernel (tabella dei file aperti del processo) in cui ogni sessione aperta sui file viene mantenuta attraverso uno specifico file descriptor. Il paradigma di uso è dunque del tipo open-read-write-close e consiste in:

- apertura della sessione
- operazioni della sessione (read/write)
- chiusura della sessione

Ovviamente, internamente alla comunicazione, si specificano più parametri per definire un collegamento con connessione rispetto ad uno con file. Tra questi si specificano il protocollo di trasporto e i due endpoint tramite la quadrupla

<indirizzo locale; processo locale; indirizzo remoto; processo remoto>



Tipi di comunicazione

I tipi di socket tra i cui è possibile scegliere che si differenziano in base al tipo di servizio offerto sono i seguenti:

- datagram: scambio di messaggi senza garanzie (best effort).
- stream: scambio bidirezionale di messaggi in ordine, senza errori, non duplicati, nessun confine di messaggio, out-of-band flusso.
- seqpacket: messaggi con numero di ordine (XNS).
- raw: messaggi scambiati senza azioni aggiuntiva.

I domini in cui avvengono le comunicazioni sono invece i seguenti:

- UNIX (AF_UNIX)
- Internet (AF_INET)
- XEROX (AF_NS)
- CCITT (AF_CCITT)

Abbiamo dunque le seguenti combinazioni tra tipo di socket e dominio:

Tipo socket	AF_UNIX	AF_INET	AF_NS
Stream socket	Possibile	TCP	SPP
Datagram socket	Possibile	UDP	IDP
Raw socket	No	ICMP	Possibile
Seq-packet socket	No	No	SPP

Sistema di nomi per le socket

Ogni socket, al fine di comunicare, presenta un nome logico o locale, costituito dall'indirizzo della socket nel dominio, e un nome fisico, composto da una porta sul nodo.

Per comunicare all'esterno dunque occorre effettuare un binding tra socket logica e entità fisica. Ciò avviene nei seguenti modi in base al diverso dominio:

- Internet → socket collegata a porta locale al nodo
{famiglia indirizzo, indirizzo Internet, numero di porta}
- UNIX → socket legata al file system locale
{famiglia indirizzo, path nel filesystem, file associato}

- CCITT → indirizzamento legato al protocollo di rete X.25

Strutture per gli indirizzi

Per rappresentare gli indirizzi utilizzati dalle socket in C occorre utilizzare strutture dati che considerano la flessibilità di questi.

Le strutture dati principali sono le seguenti:

- `sockaddr`: struttura generica utilizzata per rappresentare un indirizzo di socket.

```
struct sockaddr {
    u_short sa_family; // Famiglia di indirizzamento (es. AF_INET per IPv4)
    char sa_data[14]; // Dati specifici dell'indirizzo
};
```

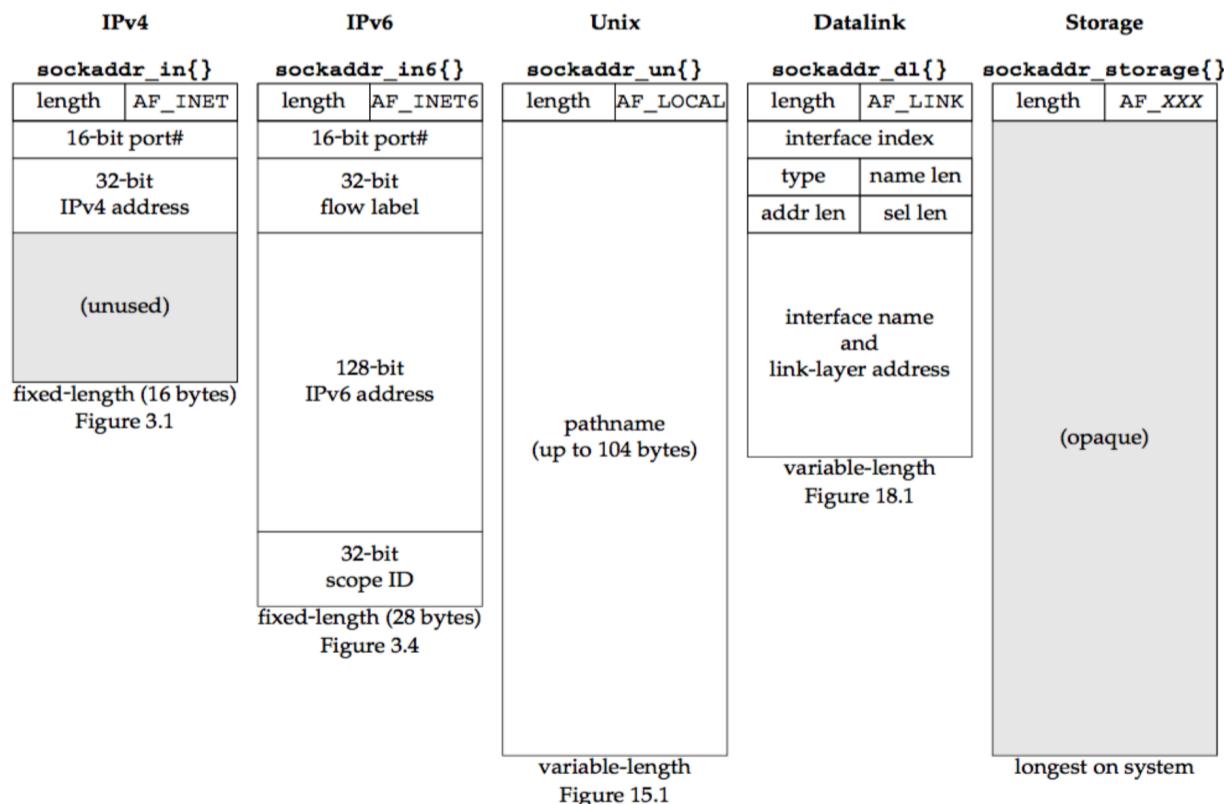
- `sockaddr_in`: struttura specifica per un indirizzo della famiglia internet (AF_INET).

```
struct sockaddr_in {
    u_short sin_family; // Famiglia di indirizzamento (sempre AF_INET)
    u_short sin_port; // Numero di porta (in formato di rete, big-endian)
    struct in_addr sin_addr; // Indirizzo IP del nodo
    char sin_zero[8]; // Padding (non usato)
};
```

- `in_addr`: struttura per rappresentare un indirizzo IP in formato binario.

```
struct in_addr {
    u_long s_addr; // Indirizzo IPv4 in formato binario (32 bit)
};
```

Esistono strutture apposite anche per altre tipologie di indirizzi:



Per quanto riguarda l'indirizzo `s_addr` è anche possibile utilizzare costanti utili come `INADDR_ANY`, la quale consente di indicare che il socket ascolterà su tutte le interfacce di rete disponibili (es. se un server ha due interfacce di rete, una con l'indirizzo IP `192.168.1.1` e un'altra con `10.0.0.1`, il socket sarà in grado di accettare connessioni su entrambi).

Siccome gli indirizzi nelle strutture viste sono binari a 32 bit, esistono delle funzioni per traslare da IP binario a 32 bit a stringa decimale a byte separato a punti (es. "123.34.56.78"):

- `inet_addr(stringadot)`: converte l'indirizzo dalla forma con punto decimale a binario a 32 bit.
- `inet_ntoa(indirizzo)`: converte l'indirizzo dalla forma binario 32 bit a quella con punto decimale.

Siccome gli indirizzi IP non sono stringhe in quanto non hanno un terminatore, è utile conoscere delle funzioni che non richiedono fine stringa ma assumono solo blocchi di byte senza terminatore per lavorare su indirizzi e fare operazioni di set, copia, confronto:

```
// funzioni BSD
bcmq (byte[] addr1, byte[] addr2, int length)
bcopy (byte[] addr1, byte[] addr2, int length)
bzero (byte[] addr1, int length)
// funzioni System V
memset (byte[] addr1, char c, int length)
memcpy (byte[] addr1, byte[] addr2, int length)
memcmp (byte[] addr1, byte[] addr2, int length)
```

Da nome logico a nome fisico

Un utente conosce il nome logico Internet di un Host remoto come stringa e non conosce il nome fisico corrispondente.

Per questo motivo C offre la primitiva `struct hostent *gethostbyname (char *name)` che restituisce un puntatore alla descrizione completa di un host dato come parametro il suo nome logico. Solitamente serve a ricavarsi l'indirizzo IP di un host dato il suo nome logico. La struttura ritornata è del seguente tipo:

```
struct hostent {
    char *h_name; /* nome ufficiale dell'host */
    char **h_aliases; /* lista degli aliases */
    int h_addrtype; /* tipo dell'indirizzo host */
    int h_length; /* lunghezza dell'indirizzo */
    char **h_addr_list; /* lista indirizzi dai nomi host */
#define h_addr h_addr_list[0] /* indirizzo IP principale host */
}
```

Per accedere dunque al valore dell'indirizzo IP fisico ritornato il codice è il seguente, il quale usa una variabile di appoggio riferita tramite puntatore che ha valore in caso di successo per assegnare l'indirizzo IP alla variabile `sockaddr_in peeraddr`.

```
struct hostent *hp;
struct sockaddr_in peeraddr;
peeraddr.sin_family = AF_INET;
peeraddr.sin_port = 22375;
if (hp = gethostbyname (argv[1])) /* caso di successo */
    peeraddr.sin_addr.s_addr =
        ((struct in_addr *) (hp->h_addr)) -> s_addr;
else /* errore o azione alternativa */
```

In modo simile, per consentire ad un utente di usare dei nomi logici di servizio senza ricordare la porta, la seguente funzione restituisce il numero di porta relativo ad un servizio:

```
struct servent *getservbyname(char *name, char *protocol)
```

Per accedere dunque al valore della porta restituito si utilizza dunque il seguente codice:

```
struct servent *sp;
struct sockaddr_in peeraddr;
sp = getservbyname("echo", "tcp");
peeraddr.sin_port = sp->s_port;
```

Socket in C

Tutte le primitive

Lista di tutte le primitive che si possono utilizzare per le socket in C:

Chiamata	Significato
<code>socket()</code>	Crea un descrittore da usare nelle comunicazione di rete
<code>connect()</code>	Connette la socket a una remota
<code>write()</code>	Spedisce i dati attraverso la connessione
<code>read()</code>	Riceve i dati dalla connessione
<code>close()</code>	Termina la comunicazione e dealloca la socket
<code>bind()</code>	Lega la socket con l'endpoint locale
<code>listen()</code>	Socket in modo passivo e predisponde la lunghezza della coda per le connessioni
<code>accept()</code>	Accetta le connessioni in arrivo
<code>recv()</code>	Riceve i dati in arrivo dalla connessione
<code>recvmsg()</code>	Riceve i messaggi in arrivo dalla connessione
<code>recvfrom()</code>	Riceve i datagrammi in arrivo da una destinazione specificata
<code>send()</code>	Spedisce i dati attraverso la connessione
<code>sendmsg()</code>	Spedisce messaggi attraverso la connessione
<code>sendto()</code>	Spedisce i datagrammi verso una destinazione specificata

Chiamata	Significato
<code>shutdown()</code>	Termina una connessione TCP in una o in entrambe le direzioni
<code>getsockname()</code>	Permette di ottenere la socket locale legata dal kernel (vedi parametri <code>socket, sockaddr, length</code>)
<code>getpeername()</code>	Permette di ottenere l'indirizzo del pari remoto una volta stabilita la connessione (vedi parametri <code>socket, sockaddr, length</code>)
<code>getsockopt()</code>	Ottiene le opzioni settate per la socket
<code>setsockopt()</code>	Cambia le opzioni per una socket
<code>perror()</code>	Invia un messaggio di errore in base a <code>errno</code> (stringa su <code>stderr</code>)
<code>syslog()</code>	Invia un messaggio di errore sul file di log (vedi parametri <code>priority, message, params</code>)

Sono tutte sincrone, quelle scritte in bold possono avere una durata elevata e le vedremo in seguito più nello specifico.

Primitive preliminari

Per lavorare sulle socket sono preliminari le due primitive `socket` e `bind`.

La primitiva `socket` prende in input il dominio (UNIX, internet, ecc.), il tipo (datagram, stream ecc.) e il protocollo utilizzato e restituisce il file descriptor associato alla socket appena creata:

```
int socket(int dominio, int tipo, int protocollo)
```

Per il tipo di connessione si utilizzano le costanti `SOCK_DGRAM` e `SOCK_STREAM`.

Al fine di agganciare la socket ad un nome fisico si utilizza la primitiva `bind`, la quale prende come argomento il file descriptor della socket, un indirizzo `sockaddr` e la lunghezza dell'indirizzo locale, restituendo un valore positivo in caso di successo:

```
int bind(int s, struct sockaddr *addr, int addrlen)
```

Presentazione dei dati

Per quanto riguarda l'inserimento dei dati da inviare nel messaggio questi possono essere rappresentati in diversi modi.

Gli interi ad esempio sono composti da più byte e possono essere rappresentati in memoria secondo due modalità diverse di ordinamento:

- Little endian: byte più significativo nell'indirizzo più alto.
- Big endian: byte più significativo nell'indirizzo più basso.

Siccome diversi sistemi utilizzano uno o l'altro dei due sistemi, possiamo distinguere il sistema da utilizzare in NBO (Network Byte Order), ovvero l'ordinamento dei byte che viene utilizzato di default in rete (big endian), e HBO (Host

Byte Order), ossia l'ordinamento utilizzato nell'host corrente (non c'è una modalità di ordinamento di default, dipende dal sistema). È possibile convertire da HBO a NBO e viceversa in C tramite le primitive `hton()`, `htonl()` (conversione da HBO a NBO, la prima per word short 16 bit, la seconda per double word long 32 bit) e `ntohs()` e `ntohl()` (conversione da NBO a HBO, la prima a 16 bit, la seconda a 32):

```
shortlocale = ntohs(shortrete);
longlocale = ntohl(longrete);

shortrete = htons(shortlocale);
longrete = htonl(longlocale);
```

Si utilizza solitamente htons al fine di utilizzare in una struttura di indirizzo di rete un indirizzo o una porta data in input dall'utente/inserita a mano dal programmatore, altrimenti quando si utilizzano gethostbyname() e getservbyname() l'indirizzo e la porta restituiti sono già in formato NBO.

Funzioni utili

Alcune funzioni utili per le socket e gli indirizzi in C sono le seguenti:

- `int getsockname(int s, struct sockaddr *addr, socklen_t *addrlen)`

Consente di inserire nella struttura `addr` passata tramite puntatore informazioni sull'indirizzo associato alla socket `s`. Ritorna 0 in caso di successo, -1 in caso di errore.

- `struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type)`

Ritorna una struttura hostent che contiene informazioni sull'host a partire dall'indirizzo `addr` passato per parametro tramite puntatore.

Interferenza tra primitive e segnali

In alcuni kernel, le primitive sospensive hanno un qualche problema in caso di interruzione con segnali, dovuto a interferenza tra spazio kernel e utente.

Una primitiva sospensiva interrotta da un segnale deve dunque essere riattivata dall'inizio usando uno schema come il seguente:

```
while (1) {
    ns = accept(f, (struct sockaddr *) &from, &len);
    if (g < 0) {
        if (errno == EINTR) {
            /* ripetizione primitiva */
            syslog(LOG_ERR, "...");
            continue;
        }
    }
}
```

Opzioni per le socket

Attraverso le opzioni sulla singola socket si possono cambiare molti comportamenti, anche in modo molto granulare e con molto controllo.

Opzioni	Descrizione
SO_DEBUG	abilita il debugging (valore diverso da zero)
SO_REUSEADDR	riuso dell'indirizzo locale
SO_DONTROUTE	abilita il routing dei messaggi uscenti
SO_LINGER	ritarda la chiusura per messaggi pendenti
SO_BROADCAST	abilita la trasmissione broadcast
SO_OOBINLINE	messaggi prioritari pari a quelli ordinari
SO_SNDBUF	setta dimensioni dell'output buffer
SO_RCVBUF	setta dimensioni dell'input buffer
SO SNDLOWAT	setta limite inferiore di controllo di flusso out
SO RCVLOWAT	limite inferiore di controllo di flusso in input
SO_SNDTIMEO	setta il timeout dell'output
SO_RCVTIMEO	setta il timeout dell'input
SO_USELOOPBACK	abilita network bypass
SO_PROTOCOL	setta tipo di protocollo

È possibile leggere e modificare opzioni di una socket con i metodi `getsockopt()` e `setsockopt()`:

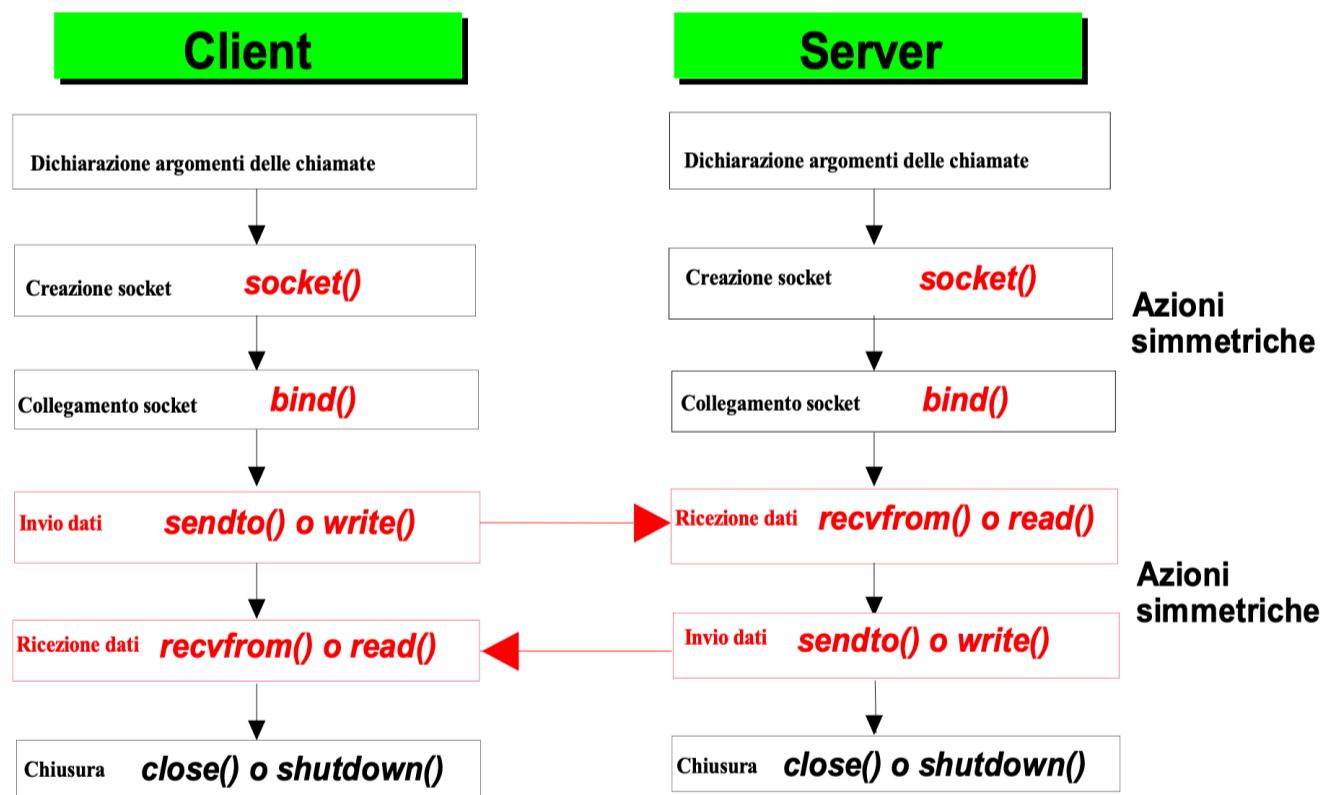
- `int getsockopt(int s, int level, int optname, int *optval, int *optlen)`
- `int setsockopt(int s, int level, int optname, int *optval, int *optlen)`

In entrambi i metodi optname indica l'opzione da leggere/modificare, e optval è un puntatore ad un'area di memoria che conterrà il valore dell'opzione in caso di lettura, e il valore che verrà dato all'opzione nel caso di modifica.

▼ 3.2.2 - Socket datagram in C

Protocollo

Le socket datagram sono usate con un protocollo che si basa sulla sequenza di primitive qui sotto:



Invio e ricezione

Per comunicare ci sono dunque due primitive (entrambi restituiscono il numero di byte trasmessi/ricevuti):

- `int sendto(int s, char *msg, int len, int flags, struct sockaddr_in *toaddr, int toaddrlen)`
- `int recvfrom(int s, char *buf, int len, int flags, struct sockaddr_in *fromaddr, int *fromaddrlen)`

Un esempio di invio e ricezione tramite datagram socket è il seguente:

```

struct sockaddr_in *servaddr; char msg[2000]; int count; ...
count = sendto(s, msg, sizeof(msg), 0,
               servaddr, sizeof(struct sockaddr_in));
...
close (s);

```

```

struct sockaddr_in *clientaddr; char buf[2000];
int count, addrlen; ...
addrlen = sizeof(sockaddr_in); /* valore di ritorno */
count = recvfrom(s, buf, sizeof(buf), 0,
    clientaddr, sizeof(struct sockaddr_in));
...
close (s);

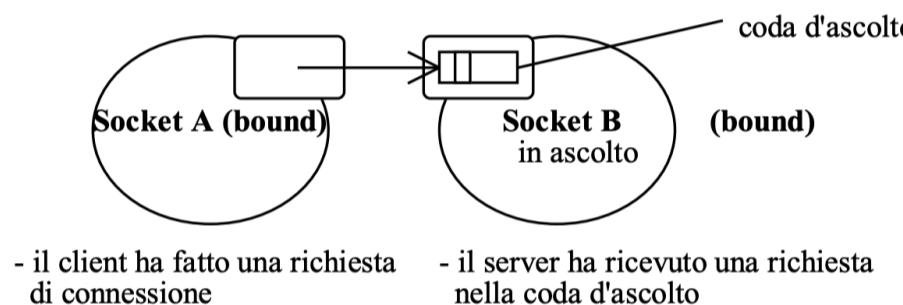
```

I datagrammi scambiati sono messaggi di lunghezza limitata su cui si opera con un'unica azione, in invio e ricezione senza affidabilità alcuna nella comunicazione (best effort, UDP). Nel caso di un server inattivo o di perdita di messaggi non vengono segnalati errori. La primitiva recvfrom restituisce un solo datagramma per volta, dunque per migliorare l'affidabilità è consigliato l'invio di più dati aggregati in un unico datagramma e la ritrasmissione dei messaggi con richiesta di datagramma di conferma.

▼ 3.2.3 - Socket stream in C

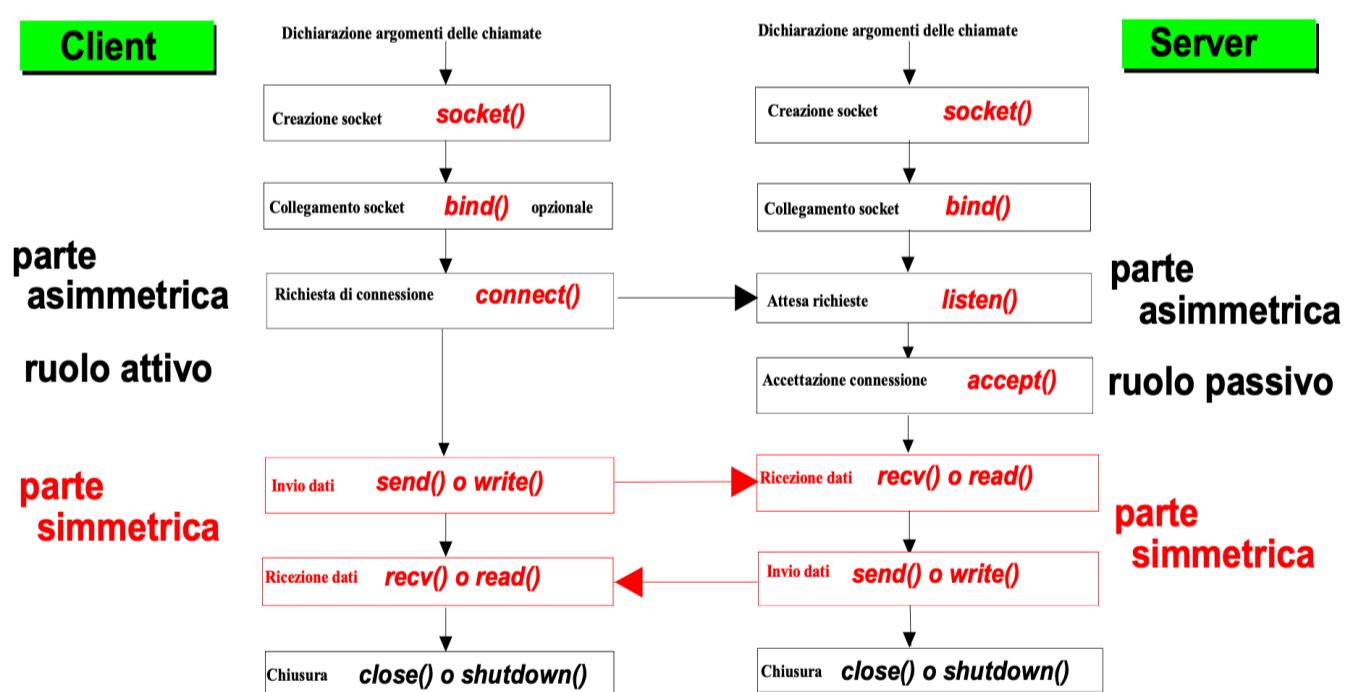
Protocollo

Le socket stream prevedono una risorsa che rappresenta una connessione virtuale tra due entità interagenti. Qui, a differenza delle socket datagram, il cliente e il servitore hanno protocollo e ruoli differenziati.



Una volta stabilita la connessione però, la comunicazione tra le entità interagenti è del tutto simmetrica.

Il protocollo delle socket stream è dunque il seguente:



La connessione, una volta stabilita, permane fino alla chiusura di una delle due half-association, ossia alla decisione di una delle due entità interagenti.

Stabilire la connessione

Per creare una connessione il cliente deve utilizzare la primitiva `int connect(int s, struct sockaddr *addr, int addrlen)`, la quale è una primitiva di comunicazione, sincrona, e termina quando la richiesta è accodata o in caso di errore (risultato < 0) rilevato dopo avere fatto delle ritrasmissioni. In caso di successo, il client considera immediatamente la connessione stabilita (anche se il server non ha ancora accettato il tutto). I clienti possono decidere se fare la bind o meno prima di richiamare connect, perché non hanno necessità di essere visibili in modo esterno, ma solo con

meccanismo di risposta, e inoltre la primitiva connect è capace di invocare la bind assegnando al cliente la prima porta libera.

```
int s;
struct sockaddr_in *addr;

s = socket(AF_INET, SOCK_STREAM, 0);
res = connect(s, addr, sizeof(struct sockaddr_in));
if (res < 0) // errore e exit
else // procedi con la connessione
```

Al fine di poter portare la connect a buon fine il servitore deve creare una coda per possibili richieste di servizio tramite la primitiva `int listen(int s, int backlog)`, in cui il backlog indica il numero di posizioni sulla coda di richieste (1-10, tipicamente 5) e il valore ritornato è negativo in caso di errore e se positivo indica il file descriptor. Successivamente, al fine di gestire una singola richiesta accodata, il server deve richiamare la primitiva `int accept (int s, struct sockaddr *addr, int *addrlen)`, la quale ritorna un valore negativo in caso di errore, altrimenti il file descriptor di una nuova socket connessa al cliente.

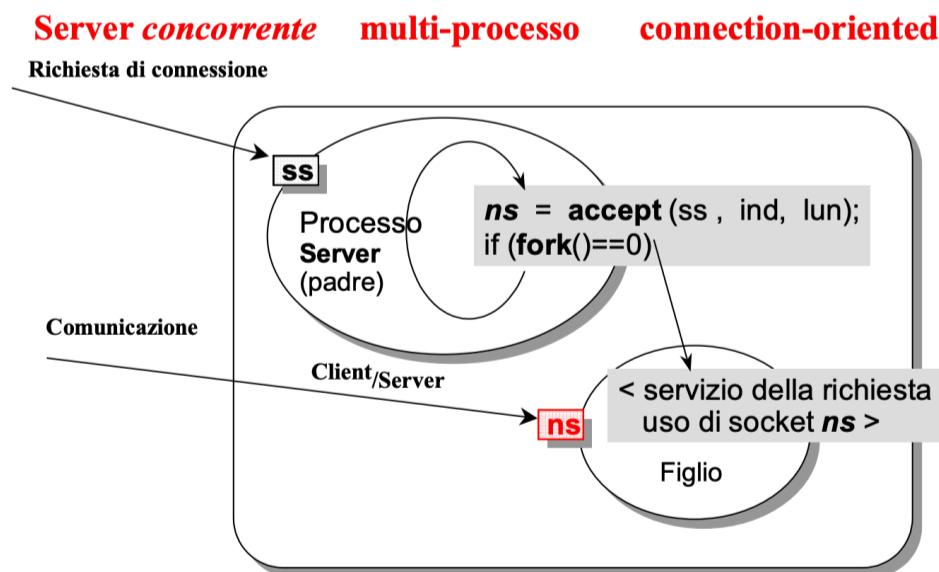
```
int s, res, backlog;
struct sockaddr_in *myaddr;
struct sockaddr_in *peeraddr;
int peeraddrlen;

s = socket(AF_INET, SOCK_STREAM, 0);
res = bind(s, myaddr, sizeof(struct sockaddr_in));
res = listen(s, backlog);
ns = accept(s, peeraddr, &peeraddrlen);

if (ns < 0) // errore e exit
else // procedi avendo la nuova socket ns
```

Server concorrenti

Per quanto riguarda server concorrenti, questi prima effettuando l'accept e poi creano il processo figlio che gestisce la richiesta.



Server demone

Al fine di utilizzare il server come un demone è possibile utilizzare la funzione `setsid()`, la quale separa il processo corrente dalla sessione e dal gruppo di processi che lo ha avviato. Nella pratica, se il processo è stato avviato da terminale, questo non sarà più connesso all'input e all'output del terminale e non terminerà con la chiusura di questo.

Invio e ricezione

Per quanto riguarda la comunicazione nella socket, client e server possono utilizzare quattro primitive (read e write in quanto le socket stream possono essere viste come un flusso continuo di dati, simile a quello dei file):

- `int send(int s, char *msg, int len, int flags)`
- `int recv (int s, char *buf, int len, int flags)`
- `int write (int s, char *msg, int len)`
- `int read (int s, char *msg, int len)`

La scrittura non è sincrona con la lettura a livello applicativo, dunque si possono fare molte scritture e consumare tutti i byte con una unica lettura e viceversa. Questo non vale nelle socket datagram in quanto i dati non sono visti come un flusso continuo di byte, ma datagrammi discreti. Le send/write e le recv/read sono dunque sincrone con la driver TCP che le servono, ma non con l'altro endpoint, quindi si sospendono se la driver ha esaurito la memoria o se non ci sono byte da leggere.

Per leggere un singolo messaggio dalla driver TCP, la quale può contenere byte appartenenti a più messaggi inviati, solitamente si utilizzano messaggi a lunghezza fissa e si specifica tale lunghezza tra i parametri della recv/read, altrimenti, se si vogliono utilizzare messaggi a lunghezza variabile, si alternano un messaggio a lunghezza fissa e uno variabile letti in due letture in successione, in cui il primo contiene la lunghezza del secondo. Un'altra tecnica è quella di utilizzare un delimitatore, ad esempio

`\n` o `\0`, il quale una volta letto fa capire che il messaggio è terminato.

Siccome i due endpoint non sono sincroni e pacchetti di grandi dimensioni possono essere suddivisi, la socket in lettura deve solitamente utilizzare un ciclo al fine di leggere tutti i byte che devono essere letti (lo stesso in scrittura, in quanto la driver potrebbe essere piena e non tutto il dato viene spedito insieme). Si utilizza dunque l'esempio di codice seguente per inviare e ricevere dati tramite socket stream:

- Invio:

```
/*
supponiamo che la richiesta sia composta da un long len
che indica la lunghezza del nome da ricevere e da tale nome
*/
typedef struct {
    long len;
    char name[100];
} Request;

int send_request(int s, Request *req) {
    int w_bytes, n = 0;
    int tot_bytes = sizeof *req;

    for (w_bytes = 0; w_bytes < tot_bytes; w_bytes += n) {
        n = send(s, ((char *) &req) + w_bytes, tot_bytes - w_bytes, 0);
        if (n <= 0) return n;
    }

    return w_bytes;
}
```

- Ricezione:

```
/*
supponiamo che la richiesta sia composta da un long len
che indica la lunghezza del nome da ricevere e da tale nome
*/
typedef struct {
    long len;
    char name[100];
} Request;

int recv_request (int s, Request *req) {
    int r_bytes, n = 0;
    int tot_bytes = sizeof *req;
```

```

for (r_bytes = 0; r_bytes < tot_bytes; r_bytes += n) {
    n = recv(s, ((char *) req) + r_bytes, tot_bytes - r_bytes, 0);
    if (n <= 0) return n;
}

return r_bytes;
}

```

Chiusura della connessione

Per non impegnare risorse non necessarie, si deve rilasciare ogni risorsa non usata con la primitiva `int close(int s)`, la quale rilascia immediatamente le risorse locali (il file descriptor della socket) e restituisce subito il controllo, ma impiega del tempo per rilasciare le risorse remote, in quanto alla chiusura ogni messaggio in uscita nel buffer associato alla socket deve essere spedito (mettendoci tutto il tempo necessario), mentre ogni dato in ingresso ancora non ricevuto viene buttato via. Solo dopo, il sistema può deallocare la memoria del buffer di trasporto e la si libera. Dopo la close il pari connesso alla socket chiusa, nel caso in cui legga dalla socket, alla fine dei byte ancora nella driver ottiene un finefile, mentre se scrive ottiene un segnale di connessione non più esistente.

In alternativa esiste la primitiva

`int shutdown(int s, int how)`, la quale permette di effettuare una chiusura direzionale (tipicamente si chiude solo il verso di uscita della connessione), la quale consente una migliore gestione delle risorse. Il parametro how può assumere i seguenti valori:

- `how = 0, SHUT_RD`
Chiude l'input, dunque non si ricevono più dati, ma si può trasmettere. La send() del pari ritorna con -1 ed il processo riceve `SIGPIPE`.
- `how = 1, SHUT_WR`
Chiude l'output, dunque si può solo ricevere dati dalla socket e non trasmettere. Il pari riceve end-of-file alla lettura, dopo il quale sa che non deve più occuparsi di quell'input e lo può chiudere.
- `how = 2, SHUT_RDWR`
Entrambi gli effetti.

▼ Esempio di copia remota di un file tramite socket stream

Il client presenta l'interfaccia:

```
rcp nodoserver nomefile
```

La scrittura del file nel directory specificato deve essere eseguita solo se in tale directory non è presente un file di nome nomefile, per evitare di sovrascriverlo. Per questo il server risponde al client con il carattere 'S' per indicare che il file non è presente, il carattere 'N' altrimenti.

Client:

```

...
// preparazione indirizzo remoto
memset((char *) &myaddr_in, 0, sizeof(struct sockaddr_in));
server_address.sin_family = AF_INET;
host = gethostbyname(argv[1]);
if (host == NULL) {
    printf("%s non trovato", argv[1]);
    exit(2);
}

server_address.sin_addr.s_addr =
    ((struct in_addr *) (host->h_addr))->s_addr;
server_address.sin_port = htons(12345); // big endian
sd = socket(AF_INET, SOCK_STREAM, 0);

```

```

// binding (automatico) e connessione al server
if(connect(sd, (struct sockaddr *) &server_address,
           sizeof(struct sockaddr)) < 0) {
    perror("Errore in connect");
    exit(1);
}

// connesso
if (write(sd, argv[2], strlen(argv[2])+1) < 0) { // invio nomefile
    perror("write");
    exit(1);
}
if ((nread = read(sd, buff, 1)) < 0) { // ricezione risposta
    perror("read");
    exit(1);
}
if(buff[0] == 'S') {
    printf("file non esiste, copia\n");
    if((fd = open(argv[2], O_RDONLY)) < 0) { // apertura file locale
        perror("open");
        close(sd);
        exit(1);
    }

    while((nread = read(fd, buff, DIM_BUFF)) > 0) // legge dal file
        write(sd, buff, nread); // invia nella socket

    close(sd);
    printf("File spedito\n");
} else {
    printf("File esiste, termino\n");
    close(sd);
}

```

Impostazione server:

```

...
// creazione socket
sd = socket(AF_INET, SOCK_STREAM, 0);
if(sd < 0) {
    perror("apertura socket");
    exit(1);
}

// binding socket
server_address.sin_family = AF_INET;
server_address.sin_port = htons(12345);
server_address.sin_addr = INADDR_ANY;
if(bind(sd, (struct sockaddr *) &server_address,
        sizeof(struct sockaddr_in)) < 0) {
    perror("bind");
    exit(1);
}

// attesa di connessioni
listen(sd, 5);
chdir("/ricevuti"); // cartella per le copie dei file

```

A questo punto sono possibili progetti differenziati per server sequenziali e concorrenti.

Server sequenziale:

```
setsid();
while (1) {
    ns = accept(sd, (struct sockaddr *) &client_address,
                sizeof(struct sockaddr_in));

    read(ns, buff, DIM_BUFF); // ricezione nomefile
    printf("server legge %s \n", buff);
    if((fd = open(buff, O_WRONLY|O_CREAT|O_EXCL)) < 0) { // creazione file output
        printf("file esiste, non opero\n");
        write(ns, "N", 1); // invio risposta
    } else {
        printf("file non esiste, copia\n");
        write(ns, "S", 1); // invio risposta
        while((nread = read(ns, buff, DIM_BUFF)) > 0) {
            write(fd, buff, nread);
            cont += nread;
        }
        printf("Copia eseguita di %d byte\n", cont);
    }
    // libera file e socket descriptor
    close(fd);
    close(ns);
}
exit(0);
```

Server parallelo:

```
setsid();
while (1) {
    ns = accept(sd, (struct sockaddr *) &client_address,
                sizeof(struct sockaddr_in));

    if (fork() == 0) {
        // figlio
        close(sd);
        read(ns, buff, DIM_BUFF);
        printf("server legge %s \n", buff);
        if((fd = open(buff, O_WRONLY|O_CREAT|O_EXCL)) < 0) { // creazione file output
            printf("file esiste, non opero\n");
            write(ns, "N", 1); // invio risposta
        } else {
            printf("file non esiste, copia\n");
            write(ns, "S", 1); // invio risposta
            while((nread=read(ns, buff, DIM_BUFF))>0) {
                write(fd,buff,nread);
                cont += nread;
            }
            printf("Copia eseguita di %d byte\n",cont);
        }
        // libera file e socket descriptor
        close(fd);
        close(ns);
        exit(0);
    }
}
```

```

    // padre
    close(ns); // libera socket descriptor
    wait(&status);
}

```

▼ 3.2.4 - Socket asincrona in C

Le primitive su socket bloccano il processo che le esegue. Pensiamo alle send e receive, queste hanno un'implementazione sincrona bloccante del processo nei confronti del kernel (non a livello applicativo):

- Send (write): aspetta fino a che non sono stati consegnati i dati al kernel.
- Receive (read): aspetta fino a che il kernel non ha qualche dato.

Eseguire una primitiva può bloccare il processo che l'ha fatta. In caso di possibile ricezione da sorgenti multiple è necessaria dunque la possibilità di attendere contemporaneamente su più eventi legati a socket diverse su cui le azioni da fare non siano bloccanti.

È possibile fare ciò in C tramite modificando le modalità della socket oppure tramite la primitiva `select()`.

Socket asincrona non bloccante tramite modalità

Un modo per rendere una socket in C asincrona non bloccante è quella di utilizzare le primitive `ioctl` e `fcntl`. In questo modo le operazioni saranno senza attesa, e al completamento l'utente viene avvisato tramite il segnale `SIGIO`, che segnala un cambiamento di stato nella socket.

Al fine di utilizzare in modo non sincrono bloccante una socket si deve prima richiamare `ioctl(s, FIOASYNC, &flag)` con `flag=1` e poi `ioctl(int s, SIOCSPGRP, &flag)`, dove la flag può assumere un valore:

- positivo → il segnale arriva a tutti i processi del process group.
- negativo → il segnale SIGIO viene inviato solo ad un processo con pid uguale al valore negato.

Un esempio di socket resa asincrona è il seguente:

```

int handler(); /* gestore delle I/O sulle socket */
signal(SIGIO, handler); /* aggancio del gestore segnale */

int flag = 1; /* valore per FIOASYNC per socket asincrona */
if (ioctl(s, FIOASYNC, &flag) == -1) {
    perror("non posso rendere asincrona la socket");
    exit(1);
}

flag = -getpid();
if (ioctl(s, SIOCSPGRP, &flag) == -1) {
    perror("non si assegna il process group alla socket");
    exit(1);
}

```

Select

La primitiva `select()`, una volta invocata, sospende il processo fino al primo evento o al timeout (se si specifica il timeout). Tali eventi possono essere:

- Eventi di lettura: rendono possibile e non bloccante un'operazione. Segnalano:
 - presenza di dati da leggere con `recv()` o `read()`
 - richiesta di connessione da trattare con `accept()`
 - end-of-file o errore in lettura
- Eventi di scrittura: segnalano un'operazione completata. Segnalano:
 - connessione con `connect()` completata

- possibilità di spedire altri dati con `send()` o `write()`
- errore in scrittura per via della chiusura della connessione da parte del pari (`SIGPIPE`)
- Eventi anomali ed eccezionali: segnalano errore o urgenza. Segnalano:
 - arrivo di dati out-of-band
 - inutilizzabilità della socket per via di `close()` o `shutdown()`

La firma della select è la seguente:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)
```

dove:

- `nfds` : numero massimo di eventi attesi.
- `readfds` , `writefds` , `exceptfds` : 3 puntatori alle maschere di eventi.
- `timeout` : puntatore a time out massimo o null se attesa indefinita.

La struttura timeval da utilizzare per il timeout è definita nel seguente modo:

```
struct timeval {
    long tv_sec; // secondi
    long tv_usec; // microsecondi
};
```

Maschere di eventi

I bit delle maschere corrispondono ai file descriptor a partire dal fd 0 fino al fd dato come primo parametro. La chiamata della `select()` esamina dunque gli eventi per i file descriptor specificati nelle tre maschere (valore ingresso bit ad 1) relative alle tre tipologie. In seguito alla chiamata le maschere date in input verranno modificate e, per i file descriptor specificati con i bit a 1, troveremo uno 0 se l'evento non si è verificato, altrimenti 1.

Nel seguente esempio la select esamina solo i file descriptor 4, 5, 7 e 9 (nfds è 10). Nella maschera di uscita troviamo che si sono verificati gli eventi 4, 5 e 7.

9	8	7	6	5	4	3	2	1	0	posizione file descriptor
1	0	1	0	1	1	0	0	0	0	maschera ingresso
0	0	1	0	1	0	0	0	0	0	maschera uscita

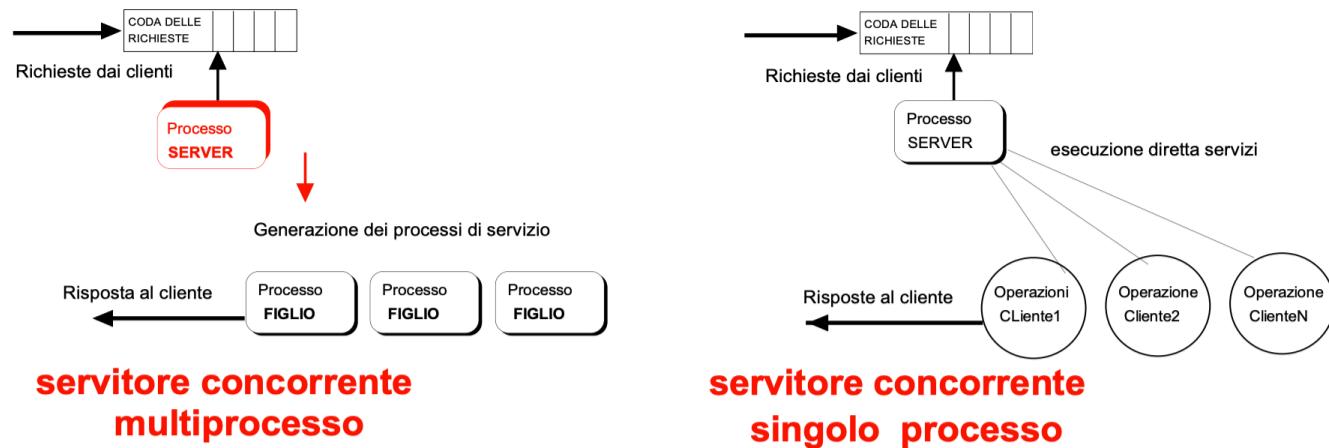
Anche un solo evento di lettura/scrittura/anomalo termina la primitiva select.

Per facilitare l'usabilità si introducono le seguenti operazioni sulle maschere:

- `void FD_SET(int fd, fd_set &fdset)` : imposta l'fd in fd_set ad 1.
- `void FD_CLR(int fd, fd_set &fdset)` : imposta l'fd in fd_set a 0.
- `int FD_ISSET(int fd, fd_set &fdset)` : restituisce il valore di fd in fd_set.
- `void FD_ZERO(fd_set &fdset)` : inizializza tutti gli fd in fd_set a zero.

Server concorrente monoprocesso con esempio

La select serve solitamente nel caso in cui si voglia creare un server concorrente monoprocesso. In questi casi è necessario considerare che tutte le socket sono legate alla stessa porta e, avendo un unico processo, i numeri delle socket sono facilmente prevedibili in quanto il fd 0, 1 e 2 sono solitamente impegnati per lo standard, ed i successivi verranno occupati ad ogni accept e liberati ad ogni close.



L'utilizzo più frequente che si fa della select è il seguente:

```

int nfds = s+1;

/* due maschere, 1 di stato stabile e 1 di supporto temporaneo
che si possa usare, cambiare valore, e ripristinare */
fd_set read_hs, temp_hs;
FD_ZERO(&read_hs);
FD_SET(s, &read_hs);
temp_hs = read_hs;

// ciclo di select per il processo servitore
while (1) {
    temp_hs = read_hs;
    select(nfds, &temp_hs, 0, 0, 0);

    int h;

    // nuova richiesta di connessione
    if (FD_ISSET(s, &temp_hs)) {
        h = accept(s, 0, 0);
        FD_SET(h, &read_hs);
        if (nfds <= h) nfds = h+1;
    }

    for (h = s+1; h < nfds; h++) {
        if (FD_ISSET(h, &temp_hs)) {
            // TODO: gestisci richiesta sulla socket h
            FD_CLR(h, &read_hs);
            close(h);
        }
    }
}

```

Cliente sequenziale o parallelo

Si può gestire la concorrenza anche dalla parte del cliente:

- soluzione concorrente: possibilità che il cliente unico gestisca più interazioni con necessità di gestione dell'asincronismo (invio senza attesa delle richieste e select per la ricezione delle risposte).
- soluzione parallela: possibilità di generare più processi (slave) che gestiscono ciascuno una diversa interazione con un server. Questo permette anche di interagire con più server contemporaneamente.

▼ 4.0 - Sistemi di chiamate remote - RPC

▼ 4.1 - Introduzione RPC

Remote Procedure Call - RPC

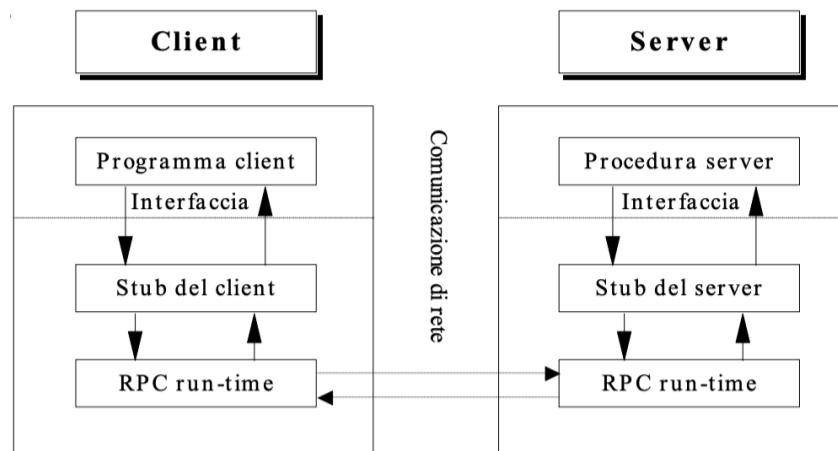
Remote Procedure Call (RPC) è un'estensione del normale meccanismo di chiamata a procedura locale. La differenza rispetto a tale meccanismo è che sono coinvolti processi distinti su nodi diversi che non condividono lo spazio di

indirizzamento. L'approccio delle RPC è un approccio applicativo di alto livello (livello 7 di OSI).

Per l'utilizzo delle RPC ogni sistema utilizza primitive diverse senza troppe regole standard.

RPC di SUN

SUN propone una chiamata RPC realizzabile tramite una primitiva `callrpc()`. Si introducono ai due endpoint di comunicazione delle routine stub per ottenere la trasparenza. In questo modo le chiamate diventano del tutto locali allo stub del proprio endpoint e si garantisce trasparenza.



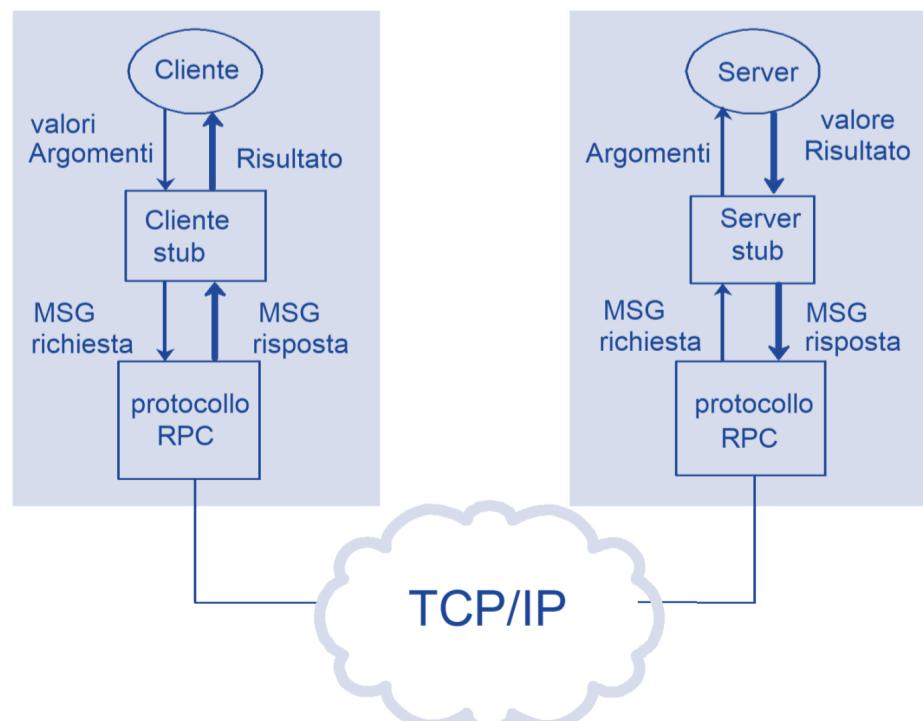
RPC di SUN permette sia semantica di comunicazione at-least-once (default) che at-most-once, e modi di comunicazione sia sincroni che asincroni.

Il modello è di tipo asimmetrico in quanto solitamente si hanno molti clienti e un solo servitore. Questo funziona in questo modo:

- Il cliente invoca uno stub che si incarica di tutto; dal recupero del server, al trattamento dei parametri e dalla richiesta al supporto runtime, al trasporto della richiesta.
- Il servitore riceve la richiesta dallo stub relativo, che si incarica del trattamento dei parametri dopo avere ricevuto la richiesta pervenuta dal trasporto. Al completamento del servizio, lo stub rimanda il risultato al cliente.

operazione(parametri)	
stub cliente: < ricerca del servitore> < marshalling argomenti> <send richiesta> <receive risposta> < unmarshalling risultato> restituisci risultato fine stub cliente;	stub servitore: < attesa della richiesta> < unmarshalling argomenti> invoca operazione locale ottieni risultato < marshalling del risultato> <send risultato> fine stub servitore;

Gli stub si occupano dunque di tutta la parte distribuita:



La semantica di RPC di SUN presenta una mutua esclusione garantita dal programma, in quanto a default il server è di tipo sequenziale e non si prevede concorrenza nell'esecuzione delle procedure.

Inoltre, fino alla restituzione del risultato di ritorno al programma cliente, il processo cliente è sospeso in modo sincrono bloccante in attesa della risposta.

Passaggio di parametri

I parametri devono passare tra ambienti diversi. In genere visto che non abbiamo memoria condivisa si ragiona con un passaggio per valore. Per effettuare un passaggio per riferimento si richiede l'utilizzo di un nuovo approccio (RMI).

Tale passaggio per valore avviene tramite un impaccamento dei parametri (marshalling) e disimpaccamento (unmarshalling) da parte degli stub.

Interface Definition Language - IDL

Interface Definition Language IDL sono linguaggi specifici per la descrizione delle operazioni remote, la specifica del servizio (detta firma) e la generazione degli stub. Esistono diversi IDL, per quanto riguarda SUN viene utilizzato l'XDR (eXternal Data Representation). Successivamente alla descrizione delle operazioni remote tramite IDL, si utilizza uno strumento RPCGEN (Remote Procedure Call Generator), ovvero un compilatore di protocollo RPC, con il quale vengono generate le procedure stub per il client e il server.

RPC Binding

Il binding prevede come ottenere l'aggancio corretto tra i clienti e il server capace di fornire l'operazione remota. Il binding può avvenire in due modi:

- Binding statico: la compilazione effettua un binding statico prima dell'esecuzione.
- Binding dinamico: Il binding viene effettuato durante l'esecuzione nel caso un cui si abbia necessità.

Questa tipologia di binding si divide a sua volta in due fasi:

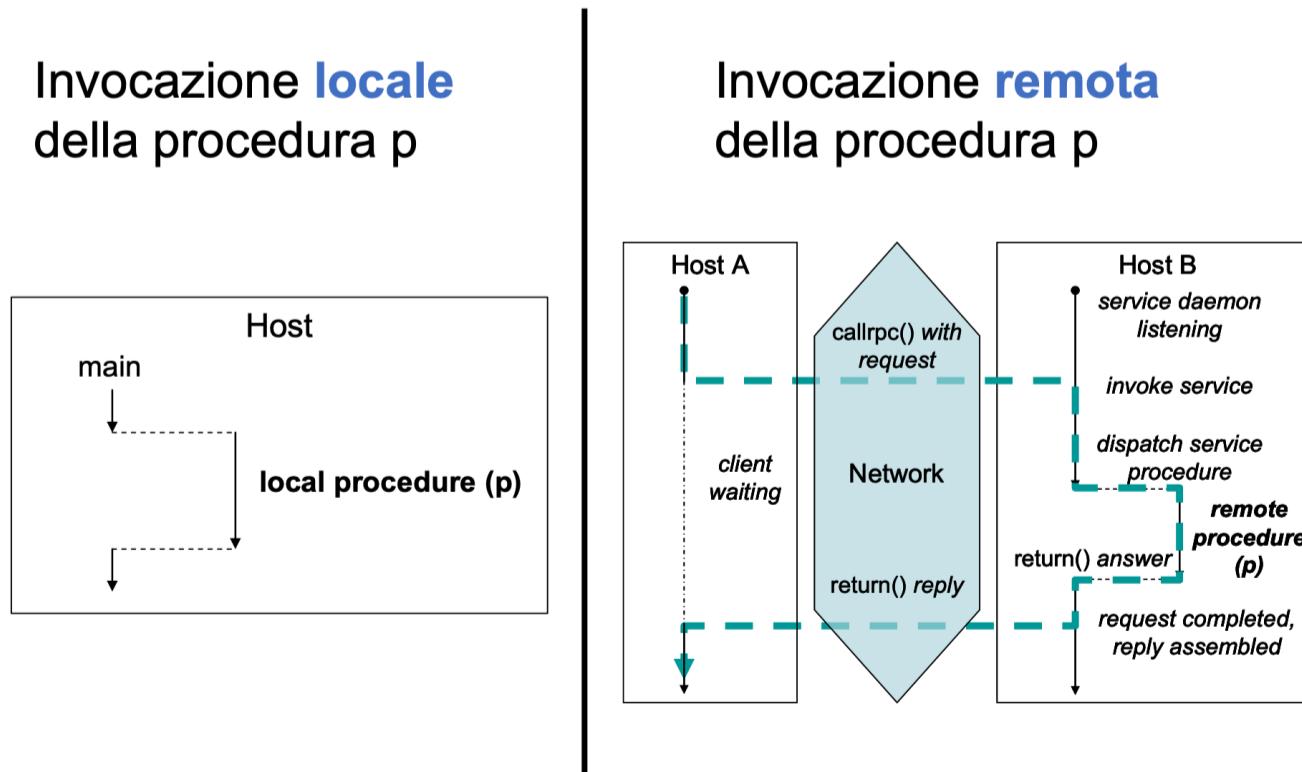
- Fase statica prima dell'esecuzione, nella quale viene effettuato il naming identificando il nome unico del servizio.
- Fase dinamica durante l'esecuzione, nella quale il cliente viene collegato realmente al servitore tramite addressing. Questa parte di addressing può essere effettuata esplicitamente dai processi, oppure implicitamente attraverso un sistema di nomi esterno.

Spesso dopo un primo legame, per questioni di costo, si usa lo stesso binding ottenuto come se fosse statico.

▼ 4.2 - Implementazione RPC di SUN

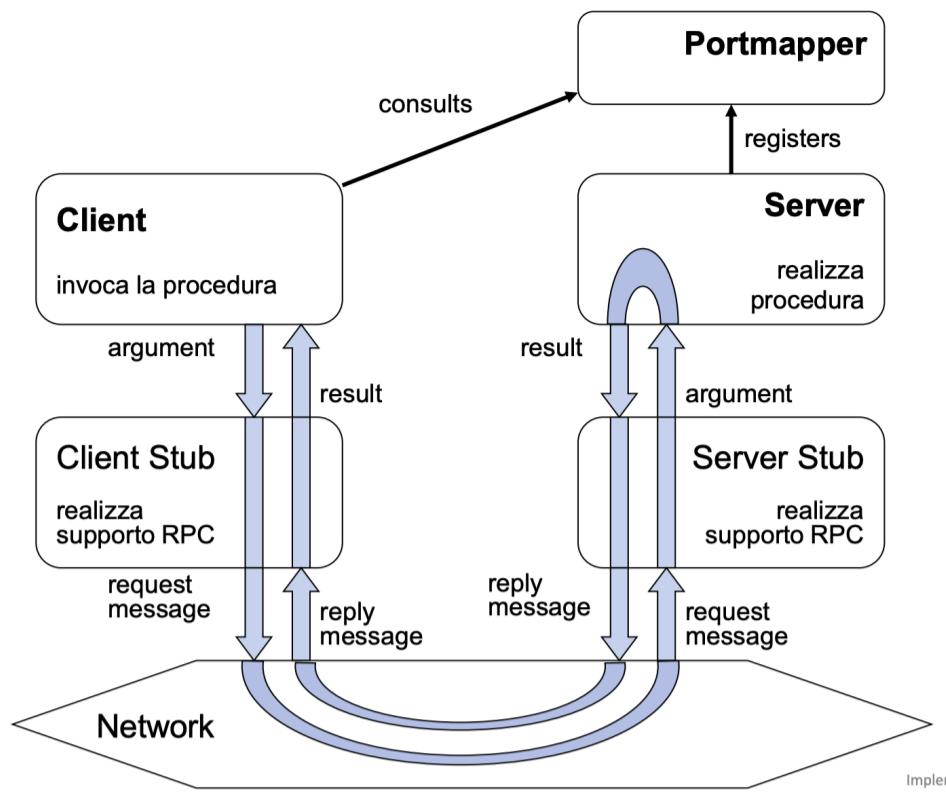
Introduzione

Le differenze principali nell'invocazione di procedure locali e remote tramite RPC di SUN sono le seguenti:

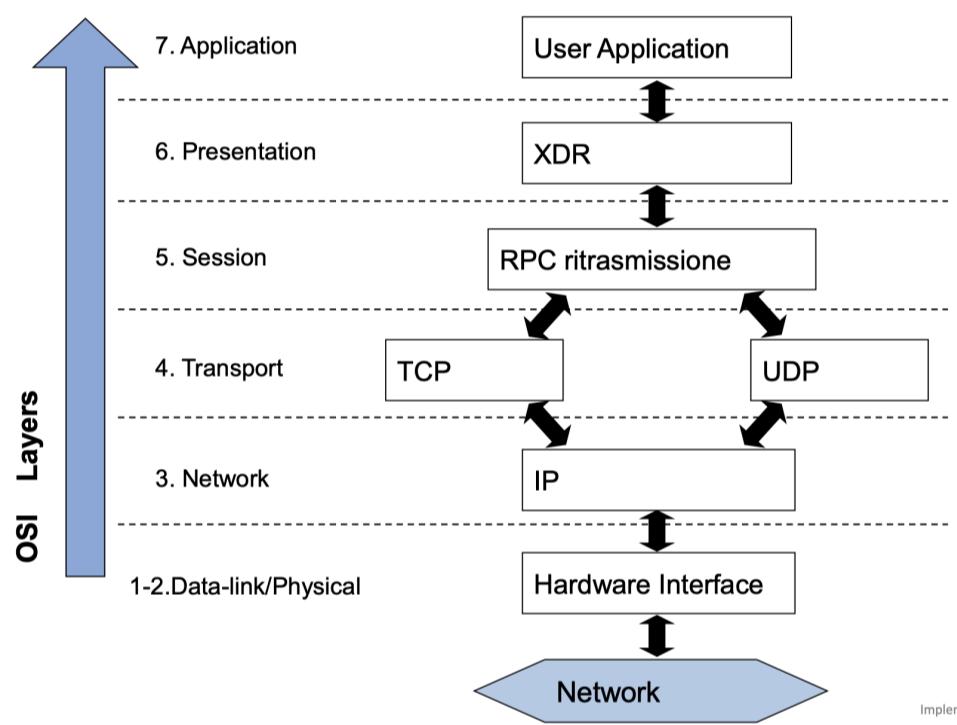


Implementazione RPC 3

RPC di SUN si basa essenzialmente sulla seguente struttura:



Lo stack OSI di RPC di SUN è invece il seguente (si può utilizzare sia TCP che UDP per il livello di trasporto):



Contratto RPC

Le RPC sono basate su un contratto esplicito sulle informazioni scambiate e che permetta un accordo preciso tra un processo che si trova su una architettura diversa da quello del fornitore del servizio. Ci sono due momenti:

- Contratto tra le operazioni che si possono chiedere e fornire in base ad un sistema di nomi che tutti conoscono. SUN sceglie di avere nomi basati sulla tripla:

<#programma, #versione, #procedura>

- Specifica della procedura da invocare usando argomenti del tipo specificato.

Tale contratto viene definito tramite due parti descrittive in linguaggio RPC:

- Definizioni del programma RPC, con specifica del protocollo utilizzato, cioè l'identificazione dei servizi e il tipo dei parametri.
- Definizioni XDR, con definizione dei tipi di dati dei parametri.

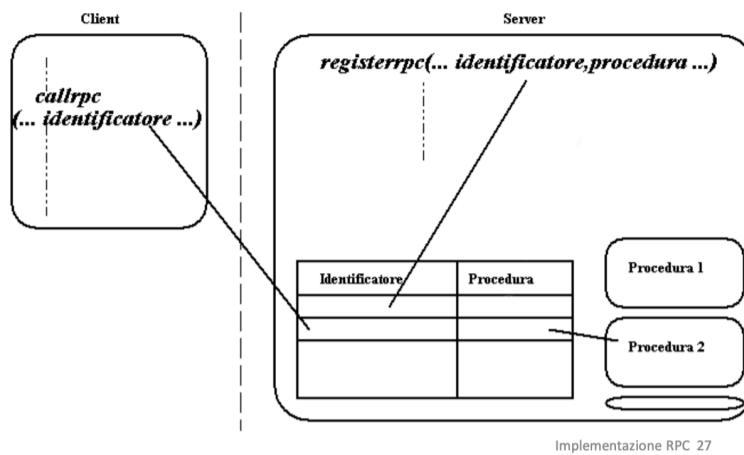
La specifica del contratto deve essere raggruppata all'interno di un unico file con estensione **.x**, cioè in formato XDR sorgente.

Sistema di nomi

Per regolare e coordinare i servizi RPC sono necessari accordi tra clienti e servitori attraverso un portmapper.

Lato server si effettua una registrazione, ovvero si associa un identificatore unico alla procedura remota implementata nell'applicazione, tramite il metodo **registerrpc()**.

Lato client si effettua una chiamata esplicita al meccanismo RPC con l'esecuzione della procedura remota tramite il metodo `callrpc()`.



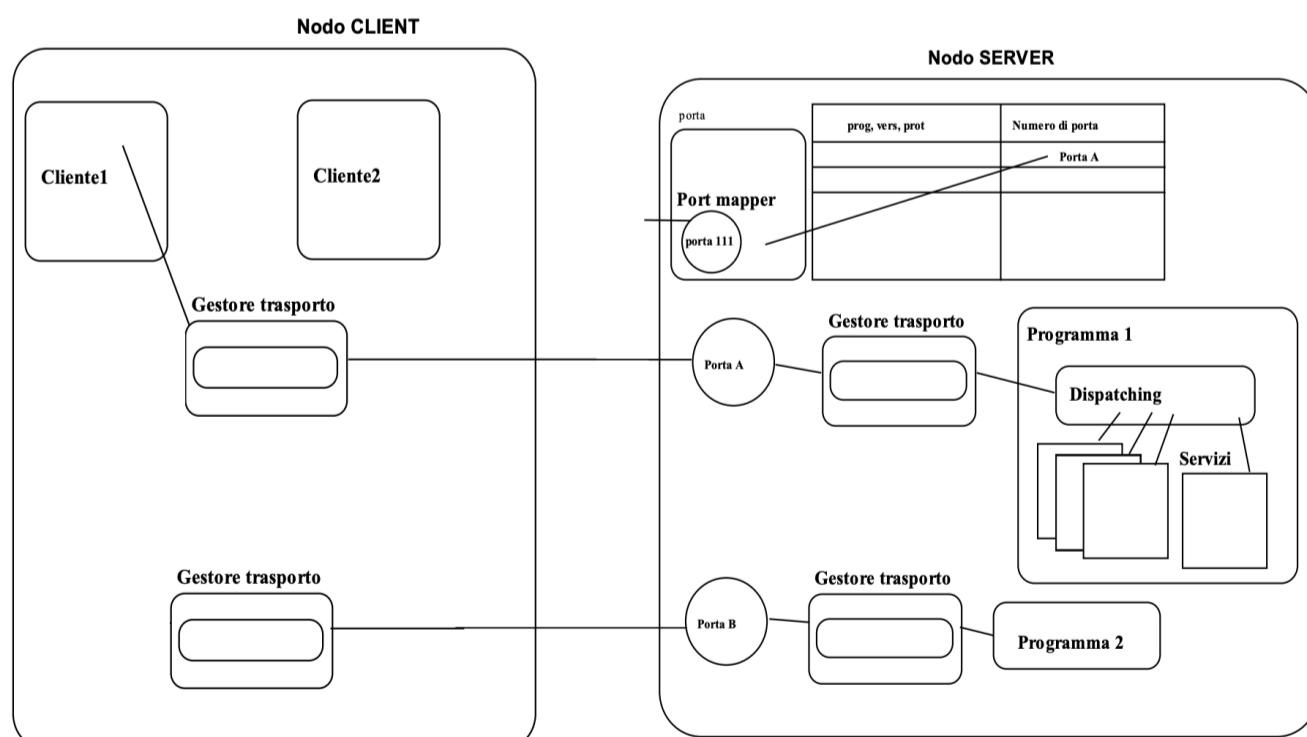
La tabella del portmapper è strutturata come un insieme di elementi con la seguente struttura:

```
struct mapping {
    unsigned long prog;
    unsigned long vers;
    unsigned int prot; // uso costanti IPPROTO_UDP ed IPPROTO_TCP
    unsigned int port; // corrispondenza con la porta assegnata
};

/* implementazione a lista dinamica della tabella */
struct *pmaplist { mapping map; pmaplist next;}
```

La fase di chiamata è preceduta da una funzione di naming con richiesta delle informazioni per il processo remoto (la porta a cui il processo è registrato). Questa informazione di stato, una volta ottenuta, viene poi mantenuta in un gestore di trasporto del cliente.

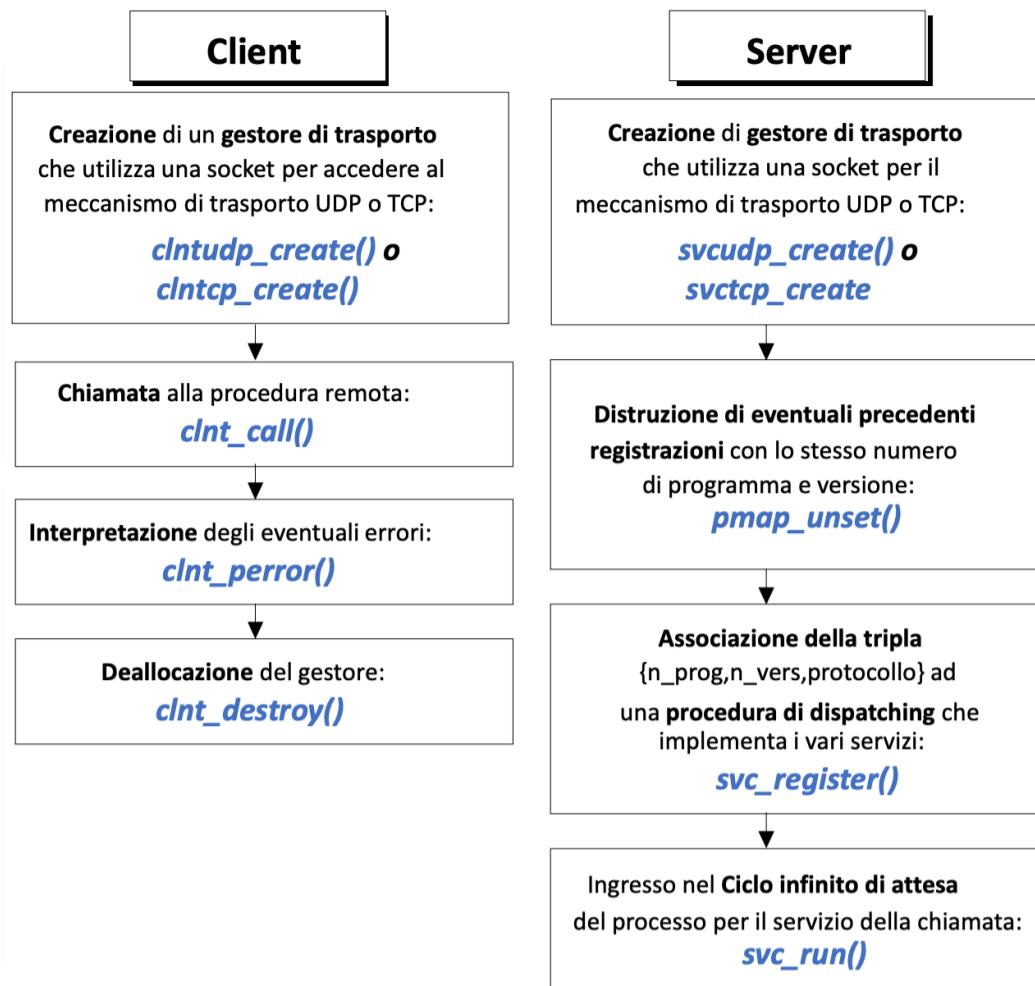
La gestione della tabella di port_map si basa su un processo unico per ogni nodo RPC detto port mapper che viene lanciato come demone in background. Il port mapper identifica il numero di porta associato ad un qualsiasi programma. Il port mapper abilita due gestori di trasporto propri, uno per UDP ed uno per TCP, con due socket legate allo stesso numero di porta (111). Il numero di programma e di versione del port mapper: 100000 2.



La richiesta del programma e del servizio sono dunque disaccoppiati: prima avviene la richiesta del numero di porta associato al programma, successivamente avviene la chiamata al servizio remoto.

Livello basso API RPC

Ora analizziamo le funzioni RPC di livello basso che vengono solitamente utilizzate all'interno degli stub (alcune di queste si usano anche nei file sviluppati dal programmatore).



Creazione di un gestore di trasporto lato server

Al fine di creare un gestore di trasporto lato server si utilizza `SVCXPRT *svcupd_create(int sock)` per ottenere un gestore UDP (default per SUN) e `SVCXPRT *svctcp_create(int sock, u_int send_buf_size, u_int recv_buf_size)` per ottenere un gestore TCP. Come argomento sock è possibile utilizzare la costante `RPC_ANYSOCK`, la quale rappresenta una socket generica che può essere utilizzata per qualsiasi tipo di connessione. Se inserito al posto di sock RPC gestisca la connessione su un socket arbitrario di sua scelta, altrimenti sulla specifica socket inserita.

Il gestore di trasporto creato ha la seguente struttura:

```

typedef struct {
#ifdef KERNEL struct socket *xp_sock;
#else int xp_sock; /* socket associata */
#endif
    u_short xp_port; /* numero di porta assoc.*/
    struct xp_ops {
        bool_t (*xp_recv)(); /* ricezione richieste */
        enum xprt_stat (*xp_stat)(); /* stato del trasporto */
        bool_t (*xp_getargs)(); /* legge gli argomenti */
        bool_t (*xp_reply)(); /* invia una risposta */
        bool_t (*xp_freeargs)(); /* libera memoria allocata */
        void (*xp_destroy)(); /* distrugge la struttura */
    } * xp_ops;
    int xp_addrlen; /* lunghezza ind. remoto */
    struct sockaddr_in xp_raddr; /* indirizzo remoto */
    struct opaque_auth xp_verf; /* controllore risposta */
    caddr_t xp_p1; caddr_t xp_p2; /* privato */
} SVCXPRT;

```

Associazione del programma alla procedura di dispatching

Al fine di associare numero programma e versione alla procedura di dispatching occorre utilizzare la funzione `svc_register()`:

```

bool_t svc_register(SVCXPRT *xprt, u_long progrnum,
    u_long versnum, char (*dispatch()), u_long protocol)

```

La procedura di dispatching seleziona il servizio da eseguire interpretando il messaggio `svc_req` consegnato dal gestore:

```

struct svc_req {
    u_long rq_prog; /* numero di programma */
    u_long rq_vers; /* versione */
    u_long rq_proc; /* procedura richiesta */
    struct opaque_auth rq_cred; /* credenziali */
    caddr_t rq_clntcred; /*credenziali a sola lettura */
    SVCXPRT *rq_xprt; /* gestore associato */
}

```

La procedura di dispatch deve avere la seguente firma:

```
void dispatch(svc_req *request, SVCXPRT *xpert)
```

Trasformazione del server in un demone

Lato server, dopo la registrazione della procedura, il processo non può terminare. Questo infatti deve rimanere in attesa di chiamate ed essere risvegliato in caso di richiesta. È possibile fare ciò tramite l'utilizzo della primitiva `svc_run()`, con la quale il processo diventa un demone in attesa di tutte le possibili richieste (è implementata come una select in un ciclo infinito).

Procedura di dispatching

Tramite il gestore di trasporto, all'interno della procedura di dispatching, è possibile ricavare i parametri per l'esecuzione tramite `bool_t svc_getargs(SVCXPRT *xpert, xdrproc_t inproc, char *in)`, e spedire la risposta tramite `bool_t svc_sendreply(SVCXPRT *xpert, xdrproc_t outproc, char *out)`.

Creazione di un gestore di trasporto lato client

Per creare il gestore di trasporto lato client l'applicazione chiamante utilizza `clntudp_create()` e `clienttcp_create()`:

```

// UDP: default
CLIENT *clntudp_create(struct sockaddr_in *addr, u_long progrnum,
    u_long versnum, struct timeval wait, int *sockp)

// TCP
CLIENT *clnttcp_create(struct sockaddr_in *addr, u_long progrnum,
    u_long versnum, int *sockp, u_int sendsz, u_int recvsz)

```

Il gestore di trasporto creato ha la seguente struttura:

```

typedef struct {
    AUTH *cl_auth; /* autenticazione */
    struct clnt_ops {
        enum clnt_stat (*cl_call)(); /* chiamata di procedura remota */
        void (*cl_abort)(); /* annullamento della chiamata */
        void (*cl_geterr)(); /* ottiene uno codice d'errore */
        bool_t (*cl_freeres)(); /* libera lo spazio dei risultati */
        void (*cl_destroy)(); /* distrugge questa struttura */
        bool_t (*cl_control)(); /* funzione controllo I/ORPC */
    } *cl_ops;
    caddr_t cl_private; /* riempimento privato */
} CLIENT;

```

Chiamata della procedura remota

Per effettuare una chiamata ad una procedura remota si utilizza la funzione `clnt_call()`:

```

enum clnt_stat clnt_call(CLIENT *clnt, u_long procnum,
    xdrproc_t inproc, char *in, xdrproc_t outproc,
    char *out, struct timeval tout)

```

Se UDP, il numero di ritrasmissioni è dato dal rapporto fra tout ed il timeout indicato nella `clntudp_create()`, se TCP, tout indica il timeout oltre il quale il server è considerato irraggiungibile.

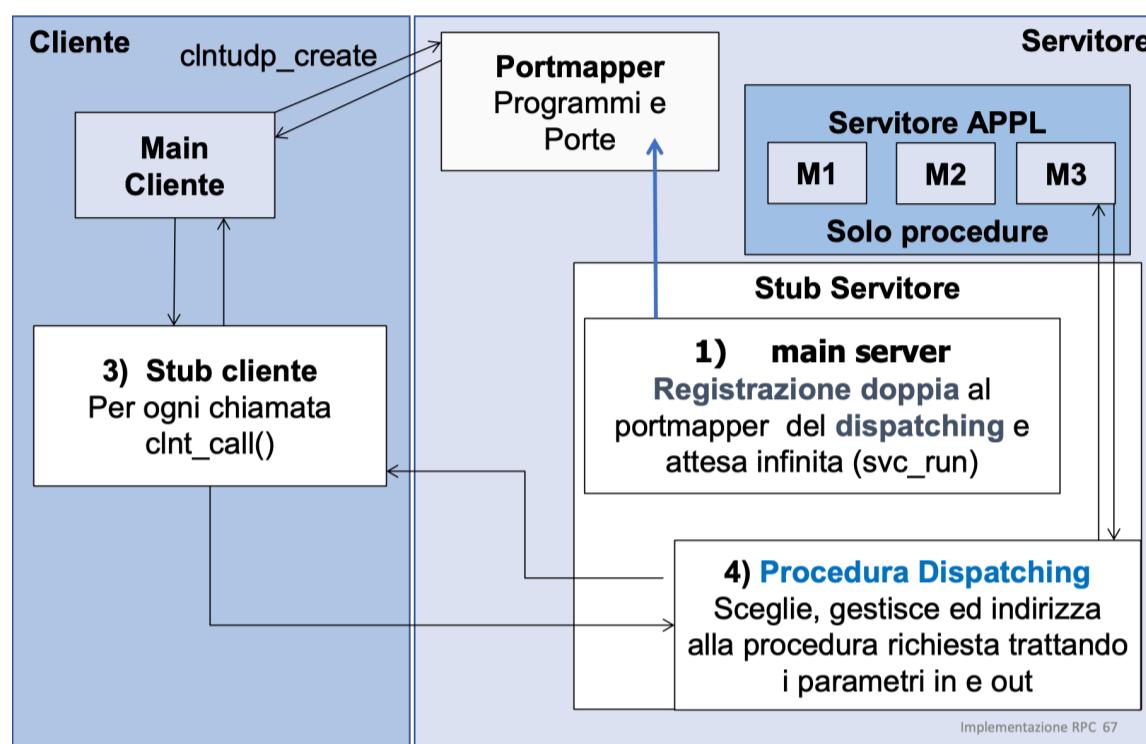
Analisi errori e rilascio delle risorse

Il risultato di `clnt_call()` può essere analizzato e con `void clnt_pcreateerror(char *s)` o `void clnt_perror(CLIENT *clnt, char *s)` si può stampare sullo standard error una stringa contenente un messaggio di errore.

Infine, per deallocare lo spazio associato al gestore `CLIENT` è possibile utilizzare la funzione `void clnt_destroy(CLIENT *clnt)`.

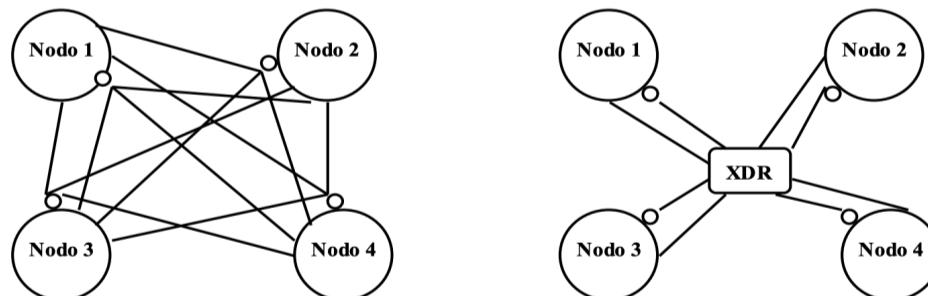
Schema generale

Schema generale del contenuto dei componenti del client e del server:



Funzioni di conversione

Per comunicare tra nodi eterogenei due soluzioni si può agire dotando ogni nodo di tutte le conversioni possibili verso e da tutti gli altri nodi, oppure concordare un formato comune di rappresentazione dei dati, dotando ogni nodo delle uniche funzioni di conversione verso e da questo formato. RCP di SUN utilizza quest'ultimo metodo, concordando come formato comune XDR, al fine di utilizzare un minor numero di funzioni di conversione.



Ogni nodo deve dunque provvedere solamente le proprie funzionalità di trasformazione dal formato locale a quello standard e viceversa. Sulla rete si utilizza il solo formato standard XDR.

Esistono funzioni che consentono lo conversione da un certo tipo di dato X a un formato XDR (ciascuna funzione restituiscono valore vero se la conversione ha successo):

Funzioni built-in		Funzioni per tipi composti	
Funzione built-in	Tipo di dato	Funzione	Tipo di dato
<code>xdr_bool()</code>	Logico	<code>xdr_array()</code>	Vettori con elementi di tipo qualsiasi
<code>xdr_char()</code> <code>xdr_u_char()</code>	Carattere	<code>xdr_vector()</code>	Vettori a lunghezza fissa
<code>xdr_short()</code> <code>xdr_u_short()</code>	Intero a 16 bit	<code>xdr_string()</code>	Sequenza di caratteri con terminatore a NULL <i>N.B.: stringa in C</i>
<code>xdr_enum()</code>	Enumerazione	<code>xdr_bytes()</code>	Vettore di byte senza terminatore
<code>xdr_float()</code>	Virgola mobile	<code>xdr_reference()</code>	Riferimento ad un dato
<code>xdr_int(),</code> <code>xdr_u_int</code>	Intero	<code>xdr_pointer()</code>	Riferimento ad un dato, incluso NULL
<code>xdr_long(),</code> <code>xdr_u_long()</code>	Intero a 32 bit	<code>xdr_union()</code>	Unioni
<code>xdr_void()</code>	Nullo		
<code>xdr_opaque()</code>	Opaco (raw byte)		
<code>xdr_double()</code>	Doppia precisione		

Implementazione RPC 43

È possibile anche creare funzioni di conversione con tipi definiti dall'utente. Esempio:

```
struct simple {int a; short b;} simple;

bool_t xdr_simple(xdrsp, simplep)
XDR *xdrsp; // rappresenta il tipo in formato XDR
struct simple *simplep; // rappresenta il formato interno
{
    if (!xdr_int(xdrsp, &simplep->a)) return(FALSE);
    if (!xdr_short(xdrsp, &simplep->b)) return (FALSE);
    return(TRUE);
}
```

Tutte queste funzioni di conversione vengono create automaticamente tramite rpcgen all'interno del file di routine di conversione XDR.

Implementazione del programma RPC

Il programmatore deve sviluppare:

- Il programma cliente: implementazione del main() e della logica necessaria per reperimento e binding del servizio remoto.
- Il programma server: implementazione di tutte le procedure (servizi).

Gli stub cliente e servitore vengono generati automaticamente.

Passi di sviluppo

I passi di sviluppo sono i seguenti:

1. Definire servizi e tipi di dati in un file con estensione `.x`.
2. Generare in modo automatico gli stub del client e del server e, se necessario, le funzioni di conversione XDR.
3. Realizzare i programmi client e server, 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, lato server, i servizi.
 - a. Attivare il Portmapper (se non attivo).
 - b. Registrare i servizi presso il Portmapper (eseguendo il server).
5. Reperire, lato client, l'endpoint del server tramite il Portmapper, creando il gestore di trasporto per l'interazione col server.

Vincoli nella definizione di procedure

Alcuni vincoli per la definizione di procedure RPC di SUN: ogni programma può contenere più procedure remote, si prevedono anche versioni multiple delle procedure, ogni definizione di procedura ha un solo parametro d'ingresso e un solo parametro d'uscita, gli identificatori (di programma, versione e procedura) usano lettere maiuscole e ogni procedura è associata ad un numero di procedura unico all'interno di un programma.

Il numero di un programma viene definito in hex (32 bit):

- 0 - 1fffffff : predefinito Sun (applicazioni d'interesse comune).
- 20000000h - 3fffffff : definibile dall'utente (applicazioni debug dei nuovi servizi).
- 40000000h - 5fffffff : riservato alle applicazioni (per generare dinamicamente numeri di programma).
- Altri gruppi riservati per estensioni.

Per quanto riguarda il numero di procedura invece bisogna ricordare che il numero di procedura 0 è riservato dal protocollo RPC per la NULLPROC, dunque si parte generalmente dalla versione 1.

Definizione servizi e tipi di dati

Un esempio completo di file XDR:

```
const MAXNAMELEN = 256;
const MAXSTRLEN = 255;

struct r_arg {string filename <MAXNAMELEN>; int start; int length;};

struct w_arg {string filename <MAXNAMELEN>; opaque block<>; int start;};

struct r_res {int errno; int reads; opaque block<>;};
struct w_res {int errno; int writes;};

program ESEMPIO {
    version ESEMPIOV {
        int PING(int) = 1;
        r_res READ(r_arg) = 2;
        w_res WRITE(w_arg) = 3;
    } = 1;
} = 0x20000020;
```

Generazione degli stub

Per la generazione degli stub si utilizza `rpcgen` (Remote Procedure Call Generator), un compilatore di protocollo RPC, il quale processa un insieme di costrutti descrittivi per tipi di dati e per le procedure remote tramite linguaggio XDR.

Dato un file `.x` di partenza `rpcgen` produce in automatico un file di testata con estensione `.h`, file stub del client, file stub del server e file di routine di conversione XDR, tutti e 3 con estensione `.c`. Lo sviluppatore deve poi realizzare il programma client e il programma server.

Invocazione della procedura

Lato client, per invocare una procedura remota, si crea innanzitutto il gestore di trasporto tramite la funzione `clnt_create()`, e poi, passando come parametro il gestore appena creato, si richiama la procedura utilizzando il suo nome e aggiungendo il carattere underscore seguito dal numero di versione (tutto in caratteri minuscoli). Gli argomenti della procedura appena descritta sono due, il primo è l'argomento vero e proprio della procedura, mentre l'altro corrisponde al gestore di trasporto `*CLIENT`. Esempio:

```
CLIENT *cl = clnt_create(char *server, RLSPROG, RLSVERS, "tcp");
/* server: hostname del server (fisico o logico) */
result = proc_1(arg, cl); // chiamata RPC
```

I parametri passati alla procedura invocata devono inoltre essere dichiarati `static` al fine di consentire il marshalling.

Argomenti e risultato

Come abbiamo già detto una procedura ha un singolo argomento e un singolo risultato. Nonostante ciò è possibile passare più informazioni definendo una struttura oppure consentire ad un parametro di poter essere di un tipo o di un altro a seconda del contesto tramite l' `union switch`. Tale struttura cambia il proprio tipo a seconda del valore in ingresso ed è possibile accedere ai propri campi nel codice utilizzando nella variabile la proprietà che come nome ha il nome della struttura al quale viene aggiunto `_u`. Esempio

```
// file .x
union readdir_res switch (int errno) {
    case 0: namelist list;
    default: void;
};
// sviluppato dal programmatore
readdir_res *result;
result->readdir_res_u.list
```

Il valore di ritorno di una procedura, inoltre, deve essere dichiarato come `static`, in quanto in questo modo permarrà oltre la singola invocazione della procedura e lo stub avrà la possibilità di effettuare il marshalling e la spedizione al client. Utilizzando la keyword `static` bisogna inoltre ricordarsi che ogni invocazione della stessa procedura utilizza il valore precedente della variabile.

▼ Esempio RPC che fornisce la lista dei file in un direttorio

File .x:

```
const MAXNAMELEN = 255;

typedef string nametype <MAXNAMELEN>;
struct namenode {nametype name; namelist next;};
typedef struct namenode *namelist;

/* risultato del servizio RLS */
union readdir_res switch (int errno) {
    case 0: namelist list;
    default: void;
};

program RLSPROG {
    version RLSVERS {
        readdir_res DIR(nametype) = 1;
    } = 1;
} = 0x20000013;
```

Client:

```
#include <rpc/rpc.h>
#include "rls.h"

int main(int argc, char *argv[]) {
    CLIENT *cl;
    namelist nl;
    char *server;
    readdir_res *result;

    static char *dir;

    if (argc != 3) {
        fprintf(stderr, "uso: %s <host> <dir>\n", argv[0]);
        exit(1);
```

```

}

server = argv[1];
dir = argv[2];
cl = clnt_create(server, RLSPROG, RLSVERS, "tcp");

if (cl == NULL) {
    clnt_pcreateerror(server);
    exit(1);
}

result = readdir_1(&dir, cl); // chiamata RPC

if (result == NULL) {
    clnt_perror(cl, server);
    exit(1);
}
if (result->remoteErrno != 0) {
    perror (dir);
    exit(1);
}

for (nl = result->readdir_res_u.list; nl != NULL; nl = nl->next)
    printf("%s\n", nl->name);

// libero le risorse distruggendo il gestore di trasporto
clnt_destroy(cl);
}

```

Server:

```

#include <rpc/rpc.h>
#include <sys/dir.h>

readdir_res *readdir_1_svc(nametype *dirname, struct svc_req *rd) {
    DIR *dirp;
    struct direct *d;
    namelist nl;
    namelist * nlp;
    static readdir_res res;

    /* si libera la memoria della chiamata precedente */
    xdr_free((xdrproc_t) xdr_readdir_res, (caddr_t) &res);

    dirp = opendir(*dirname);
    if(dirp == NULL) {
        res.remoteErrno = errno;
        return &res;
    }
    nlp = &res.readdir_res_u.list;
    while (d = readdir(dirp)) {
        nl = *nlp = (namenode*) malloc(sizeof(namenode));
        nl->name = malloc(strlen(d->d_name)+1);
        strcpy(nl->name, d->d_name);
        nlp = &nl->next;
    }
    nlp = NULL; // chiusura lista
    res.remoteErrno = 0;
}

```

```

closedir(dirp);

return &res;
}

```

▼ Stub

Client:

```

#include <memory.h> /* for memset */
#include "rls.h"
static struct timeval TIMEOUT = {25, 0};

readdir_res *readdir_1(nametype *argp, CLIENT *clnt) {
    static readdir_res clnt_res;
    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call(clnt, REaddir,(xdrproc_t) xdr_nametype,
        (caddr_t) argp, (xdrproc_t) xdr_readdir_res,
        (caddr_t) &clnt_res, TIMEOUT) != RPC_SUCCESS)
        return (NULL);
    return (&clnt_res);
}

```

Server:

```

#include "rls.h"
#include <stdio.h>

static void rlsprog_1(); /* procedura di dispatching */

int main (int argc, char **argv) {
    register SVCXPRT *transp;
    /* deregistrazione di eventuale programma con stesso nome */
    pmap_unset(RLSPROG, RLSVERS);
    /* creazione gestore trasp. e registrazione servizio con UDP */
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf(stderr, "%s", "cannot create udp service");
        exit(1);
    }
    if (!svc_register(transp, RLSPROG, RLSVERS,
        rlsprog_1, IPPROTO_UDP)) {
        fprintf (stderr, "%s", "unable to register..., udp.");
        exit(1);
    }
    /* creazione gestore trasp. e registrazione servizio con TCP */
    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        fprintf(stderr, "%s", "...");
        exit(1);
    }
    if (!svc_register(transp, RLSPROG, RLSVERS,
        rlsprog_1, IPPROTO_TCP)) {
        fprintf (stderr, "%s", "unable to register ..., tcp.");
        exit(1);
    }
    svc_run(); /* attivazione dispatcher */
    /* qui non si arriva */
}

```

```

        fprintf (stderr, "%s", "svc_run returned");
        exit (1);
    }

/* procedura di dispatching */
static void rlsprog_1
(struct svc_req *rqstp, register SVCXPRT *transp) {
    /* configurazione */
    union {nametype readdir_1_arg;} argument;
    char *result;
    xdrproc_t _xdr_argument,
    _xdr_result;
    char *(*local)(char *, struct svc_req *);
    /* sono diventati generici: local, procedura da invocare, argument
     e result i parametri di ingresso e uscita, le funzioni xdr
     xdr_argument e xdr_result */
    switch (rqstp->rq_proc) {
        case NULLPROC:
            (void)svc_sendreply(transp,(xdrproc_t)xdr_void,(char *)NULL);
            return;
        case REaddir:
            _xdr_argument = (xdrproc_t) xdr_nametype;
            _xdr_result = (xdrproc_t) xdr_readdir_res;
            local = (char *(*)(char *, struct svc_req *)) readdir_1_svc;
            break;
        default:
            svcerr_noproc (transp);
            return;
    }
    /* invocazione chiamata locale*/
    memset ((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs(transp, (xdrproc_t) _xdr_argument,
        (caddr_t) &argument)) {
        svcerr_decode (transp);
        return;
    }
    result = (*local) ((char *) &argument, rqstp);
    if (result != NULL &&
        !svc_sendreply(transp, (xdrproc_t)_xdr_result, result))
        svcerr_systemerr (transp);
    if (!svc_freeargs(transp, (xdrproc_t)_xdr_argument,
        (caddr_t) &argument)) {
        ...
        exit (1);
    }
    return;
}

```

Generalità XDR

Dichiarazione di tipi atomici del linguaggio C

- `bool` con due valori: TRUE e FALSE
- `string` con due utilizzi:
 - con specifica del numero massimo di caratteri fra `<>`.
 - lunghezza arbitraria con `<>` vuoti.

Tradotto da rpcgen in `char *name`.

- `opaque`: sequenza di byte senza un tipo di appartenenza (con o senza la massima lunghezza tra `<>`).

Tradotto da rpcgen in `struct {u_int name_len; char *name_val;} name;`

- `void`: non si associa il nome della variabile di seguito.

Nota: rpcgen non esegue sempre il controllo di tipo, dunque se il tipo indicato non appartiene ai tipi riconosciuti, il compilatore assume che sia definito a parte e non dà errore.

Dichiarazione di tipi semplici

```
typedef colortype color;
```

Dichiarazione di vettori a lunghezza fissa

```
typedef colortype palette[8];
```

Dichiarazione di matrici

In XDR non è possibile definire direttamente strutture innestate, ma bisogna sempre passare per definizioni di strutture intermedie.

```
struct RigaMatrice {char riga[20];};
struct MatriceCaratteri {RigaMatrice riga [10];};
```

Dichiarazione di vettori a lunghezza variabile

```
typedef int heights <12>; // con lunghezza massima
typedef int widths <>; // lunghezza arbitraria
```

Dichiarazione di tipi puntatori

```
typedef int *punt;
```

Dichiarazione di tipi struttura

```
struct coordinate {
    int x;
    int y;
};
```

Dichiarazione di tipi unione

```
union read_result switch (int errno) {
    case 0: opaque data[1024];
    default: void;
};
```

Definizione di tipi enumerazione

```
enum colortype {
    RED = 0,
    GREEN = 1,
    BLUE = 2
};
```

Definizione di tipi costante

```
const MAXLEN = 12;
```

Definizione di tipi non standard

```
typedef string fname_type <255>;
```

RPC: modalità asincrona

A default RPC utilizza un client sincrono con il server.

È possibile intervenire nel client per renderlo asincrono. Per fare ciò occorre utilizzare il protocollo TCP e specificare un timeout nullo quando viene invocata

`clnt_call()`. Il servitore, inoltre, non deve prevedere risposta, dunque nella procedura di risposta deve dichiarare `xdr-`
`void` come funzione XDR e 0 come argomento. Infine, non viene effettuata la chiamata `svc_sendreply()` al termine del servizio asincrono.

▼ Esempio RPC che stampa stringhe in maniera asincrona

Client:

```
#define PROG (unsigned long) 0x20000020
#define VERS (unsigned long) 1
#define PRINTSTRING_BATCHED (unsigned long) 2

#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <time.h>
#include <netdb.h>

main(int argc, char *argv) {
    struct hostent *hp;
    struct timeval total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    char buf[BUFSIZ], *s=buf;

    if (argc < 2) {
        fprintf(stderr,"uso: %s hostname\n",argv[0]);
        exit(1);
    }

    if ((hp = gethostbyname(argv[1])) == NULL) {
        fprintf(stderr,"non ho informazioni su %s\n", argv[1]);
        exit(1);
    }

    memcpy((caddr_t) &server_addr.sin_addr, hp->h_addr, hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = 0;

    /* gestore trasporto TCP */
    if((client = clnttcp_create(&server_addr, PROG, VERS, &sock, 0, 0))
       == NULL) {
        clnt_pcreateerror ("clnttcp_create\n");
        exit(1);
    }

    /* timeout sulla risposta RPC è posto a zero */
    total_timeout.tv_sec = 0;
    total_timeout.tv_usec = 0;
```

```

/* ciclo di lettura di stringhe da tastiera e spedizioni
asincrone al nodo remoto, fino a EOF */
while (scanf("%s", s) != EOF) {
    /* risultato e funzione XDR a NULL */
    clnt_stat = clnt_call(client, PRINTSTRING_BATCHED,
        xdr_wrapstring, &s, xdr_void, NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "RPC asincrona");
        exit(1);
    }
}

/* Chiamata sincrona di NULLPROC per
svuotare completamente il buffer TCP */
total_timeout.tv_sec = 20;
/* Chiamata sincrona di NULLPROC */
clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
    xdr_void, NULL, total_timeout);
if (clnt_stat != RPC_SUCCESS) {
    clnt_perror(client, "rpc");
    exit(1);
}

/* libero le risorse usate */
clnt_destroy(client);
}

```

Server:

```

#define PROG (unsigned long) 0x20000020
#define VERS (unsigned long) 1
#define PRINTSTRING (unsigned long) 1
#define PRINTSTRING_BATCHED (unsigned long) 2

#include <stdio.h>
#include <rpc/rpc.h>

void printdispatch();

main() {
    SVCXPRT *transp;
    transp = svctcp_create(RPC_ANYSOCK, 0, 0);

    if (transp == NULL) {
        fprintf(stderr,"cannot create an RPC server\n");
        exit(1);
    }
    pmap_unset(PROG, VERS);
    if (!svc_register(transp,PROG, VERS, printdispatch, IPPROTO_TCP)) {
        fprintf(stderr,"cannot register PRINT service\n");
        exit(1);
    }
    svc_run();
    fprintf(stderr,"uscita dal ciclo di attesa di richieste!\n");
}

void printdispatch(struct svc_req *rqstp, SVCXPRT *transp) {

```

```

char *s = NULL;
switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "non posso rispondere.\n");
            exit(1);
        }
        fprintf(stderr, "Fine!\n");
        return;
    case PRINTSTRING_BATCHED:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "problemi decodifica argomenti.\n");
            break;
        }
        fprintf(stderr, "%s\n", s);
        svc_freeargs(transp, xdr_wrapstring, &s);
        /* questo è un servizio asincrono: non c'è svc_sendreply() */
        break;
    default:
        svcerr_noproc(transp);
        return;
}

/* funzione di basso livello per deallocare gli
argomenti acquisiti con la chiamata svc_getargs() */
svc_freeargs(transp, xdr_wrapstring, &s);
}

```

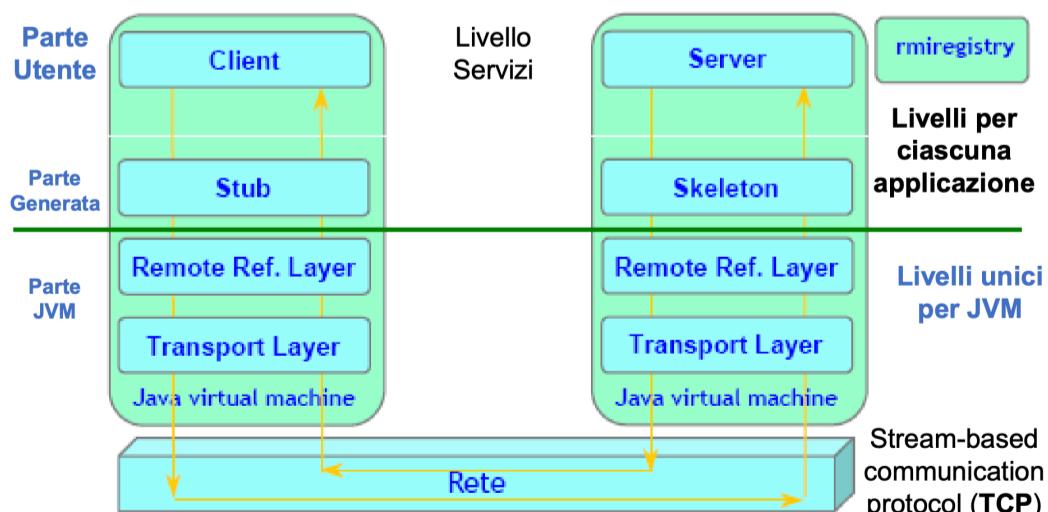
▼ 5.0 - Java RMI

▼ 5.1 - Java RMI base

Introduzione

La architettura RMI introduce la possibilità di richiedere esecuzione di metodi remoti in JAVA (RPC in JAVA) integrando il tutto con il paradigma object-oriented. RMI come insieme di strumenti, politiche e meccanismi che permettono ad un'applicazione Java in esecuzione su una macchina di invocare i metodi di un oggetto di una applicazione Java in esecuzione su una macchina remota.

Per l'RMI vengono utilizzati due proxy, stub dalla parte cliente e skeleton dalla parte servitore, i quali nascondono al livello applicativo la natura distribuita dell'applicazione. Si usa una variabile interfaccia per contenere un riferimento a un proxy che permette di controllare e preparare il passaggio da un ambiente cliente ad un ambiente servitore.



Il livello Remote Reference Layer (RRL) è responsabile della gestione dei riferimenti agli oggetti remoti, dei parametri e delle astrazioni di una connessione stream-oriented.

Il livello Transport Layer (TL) è invece responsabile della gestione delle connessioni fra gli spazi di indirizzamento delle

due JVM e gestisce il ciclo di vita delle connessioni.

I due livelli, RRL e Transport, sono parte della macchina virtuale JVM.

La semantica utilizzata nelle RMI è at-most-once con utilizzo del protocollo TCP.

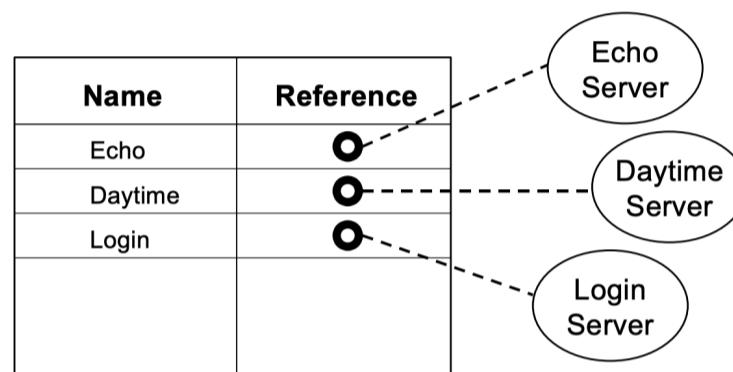
RMI Registry

Un client in esecuzione su una macchina ha bisogno di localizzare un server a cui connettersi, che è in esecuzione in una JVM su una macchina diversa.

Per fare ciò ci sono varie possibilità:

- Il client conosce in anticipo dov'è allocato il server.
- L'utente dice all'applicazione client dov'è allocato il server.
- Un servizio standard (naming service) in una locazione ben nota, che il client conosce, funziona come punto di indirezione e fornisce la locazione in risposta.

Java RMI utilizza un naming service detto RMI Registry che mantiene un insieme di coppie `{name, reference}`. `name` ha solitamente la seguente struttura: `//nomehost:porta/servizio`.



Per attivare il registry sull'host server occorre usare il programma rmiregistry di Sun lanciato in una shell a parte specificando o meno la porta, la quale a default è 1099 (es. `rmiregistry` oppure `rmiregistry 10345`).

Per interagire con l'RMI Registry si possono utilizzare i seguenti metodi della classe `java.rmi.Naming`:

- `public static void bind(String name, Remote obj)`
- `public static void rebind(String name, Remote obj)`
- `public static void unbind(String name)`
- `public static String[] list()`
- `public static Remote lookup(String name)`

Stub e Skeleton

Lo Stub e lo Skeleton rendono possibile la chiamata di un servizio remoto come se fosse locale agendo da proxy. La procedura di comunicazione nello specifico è la seguente:

1. Lo stub:
 - Effettua la serializzazione delle informazioni per la chiamata (id del metodo e argomenti).
 - Invia le informazioni allo skeleton utilizzando le astrazioni messe a disposizione dal RRL.
2. Lo skeleton:
 - Effettua la de-serializzazione 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.
3. Lo stub:
 - Effettua la de-serializzazione del valore di ritorno.
 - Restituisce il risultato al client.

Implementazione

Per ogni componente RMI occorre:

1. Definire il comportamento tramite un'interfaccia che deve estendere `java.rmi.Remote`. Ogni metodo di tale interfaccia deve propagare `java.rmi.RemoteException`.
2. Implementare il comportamento tramite una classe che implementa l'interfaccia definita ed estende `java.rmi.UnicastRemoteObject`.

Passi per l'utilizzo di Java RMI

1. Definire interfaccia e implementazione del componente server utilizzabile in remoto
2. Compilare l'interfaccia e la classe dette sopra con `javac`, poi generare stub e skeleton (con `rmi`) della classe utilizzabile in remoto.
3. Pubblicare il servizio nel sistema di nomi attivando il registry e registrando il servizio tramite `bind`.
4. Ottenerne lato cliente il riferimento all'oggetto remoto facendo una `lookup` sul registry, e compilare il cliente.

Implementazione interfaccia

L'interfaccia deve estendere `java.rmi.Remote` e ogni metodo deve propagare `java.rmi.RemoteException`, che verrà lanciata in caso di problemi. Inoltre ogni metodo ha un solo risultato di uscita e nessuno, uno o più parametri di ingresso.

Esempio:

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

Implementazione server

La classe che implementa il server deve estendere la classe `UnicastRemoteObject` e deve implementare tutti i metodi definiti nell'interfaccia.

Inoltre viene solitamente definito un processo in esecuzione sull'host del servitore che regista tramite `bind / rebind` tutti gli oggetti server da registrare, ciascuno con un nome logico.

Esempio:

```
public class EchoRMIServer extends java.rmi.server.UnicastRemoteObject  
    implements EchoInterface{  
  
    public EchoRMIServer() throws java.rmi.RemoteException {  
        super();  
    }  
  
    // implementazione metodo remoto  
    public String getEcho(String echo) throws java.rmi.RemoteException {  
        return echo;  
    }  
  
    // registrazione servizio  
    public static void main(String[] args) {  
        try {  
            EchoRMIServer serverRMI = new EchoRMIServer();  
            Naming.rebind("//localhost:1099/EchoService", serverRMI);  
        } catch (Exception e) {  
            e.printStackTrace();  
            System.exit(1);  
        }  
    }  
}
```

Implementazione client

Nel processo del client si può accedere al servizio ricavandosi una variabile interfaccia del server tramite una richiesta `lookup` al registry. Successivamente è possibile richiamare i metodi remoti attraverso tale variabile. Ogni chiamata RMI è sincrona bloccante.

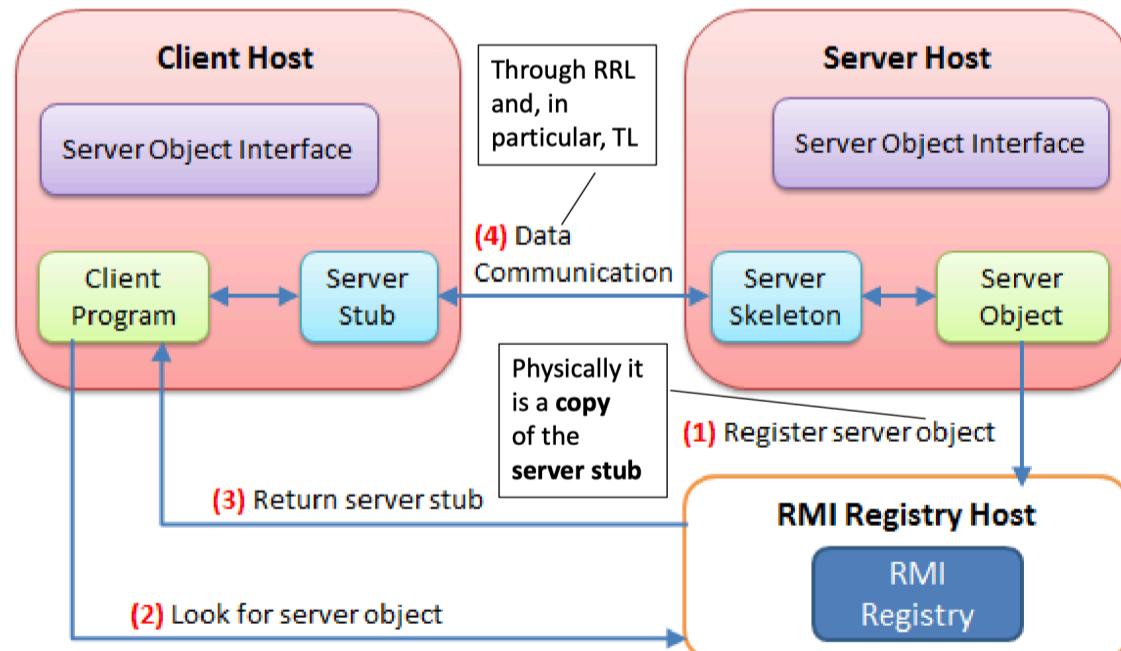
Esempio:

```
public class EchoRMIClient {
    public static void main(String[] args) {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        try {
            // connessione al servizio RMI remoto
            EchoInterface serverRMI =
                (EchoInterface) java.rmi.Naming.lookup("//localhost:1099/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);
        }
    }
}
```

Visione d'insieme



Compilazione ed esecuzione

Lato server:

1. Compilazione dell'interfaccia e dell'implementazione parte server.

```
javac EchoInterface.java EchoRMIServer.java
```

2. Generazione degli eseguibili Stub e Skeleton.

```
rmic -vcompat EchoRMIServer
```

3. Esecuzione del registry e del server.

```
rmiregistry # avviamento del registry  
java EchoRMIServer # avviamento del server
```

Lato client:

1. Compilazione dell'implementazione parte client

```
javac EchoRMIClient.java
```

2. Esecuzione del client

```
java EchoRMIClient
```

▼ 5.2 - Approfondimenti su Java RMI

Serializzazione

In generale, nei sistemi RPC, ossia in cui riferiamo funzionalità remote, i parametri di ingresso e uscita subiscono una duplice trasformazione per risolvere problemi di rappresentazioni eterogenee:

- Marshalling: processo di codifica degli argomenti e dei risultati per la trasmissione.
- Unmarshalling: processo inverso di decodifica di argomenti e risultati ricevuti.

In Java, grazie all'uso del bytecode uniforme e standard, non c'è bisogno di un/marshalling, dunque i dati vengono semplicemente serializzati/deserializzati tramite i metodi `readObject()` e `writeObject()`, utilizzati dallo stub e dallo skeleton.

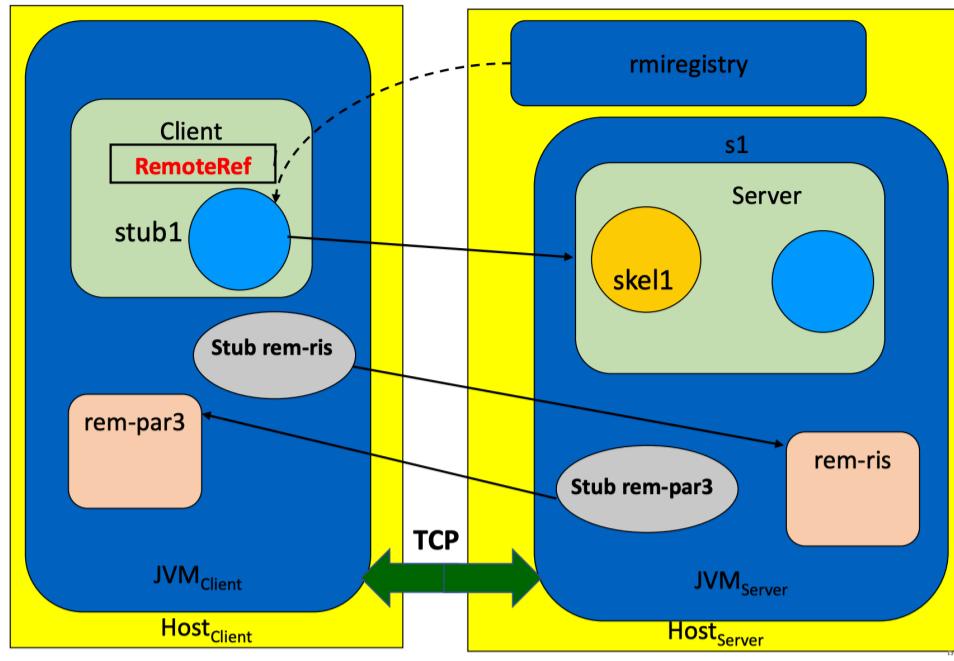
Passaggio dei parametri

In Java il passaggio di parametri ai metodi avviene nei seguenti modi:

Tipo	Metodo Locale	Metodo Remoto
Tipi primitivi	Per valore	Per valore
Oggetti	Per riferimento	Per valore (interfaccia <code>Serializable</code> o deep copy, ossia copia dell'intero grafo a partire dalla istanza e a comprendere tutti gli oggetti riferiti)
Oggetti Remoti		Per riferimento remoto (interfaccia <code>Remote</code>), ossia viene passato lo stub all'oggetto remoto corrispondente

Per il passaggio di oggetti remoti, ogni istanza di stub identifica l'oggetto remoto al quale si riferisce attraverso un identificativo (ObjID) che è univoco nella JVM dove l'oggetto remoto si trova e che viene mantenuto nello stub stesso. L'utilizzo di oggetti remoti ad esempio quando il client richiede l'istanza del server remoto all'RMI Registry. In questo caso l'RMI Registry è a tutti gli effetti un server RMI che contiene al suo interno un oggetto remoto che consente al RRL di raggiungere il server.

L'utilizzo di ogni oggetto remoto crea uno stub di tale oggetto che viene inserito nella JVM del chiamante.



È anche possibile creare all'interno del codice del server un proprio registry utilizzando il seguente metodo della classe LocateRegistry:

```
public static Registry createRegistry(int port)
```

In questo caso, il registry viene creato nella stessa istanza della JVM del server.

Livello di trasporto

Concorrenza

Java gestisce la concorrenza delle chiamate generando un thread per ogni invocazione sull'oggetto remoto in esecuzione sulla JVM.

Connessione

Tra le varie possibilità nell'utilizzo della connessione Java RMI utilizza una sola connessione per servire diverse richieste, e su quell'unica viene utilizzato un meccanismo di demultiplexing per distinguere tra più richieste e risposte. La connessione permane fino alla chiusura di una delle due JVM.

Class loading

Java è fortemente integrato col Web e consente di associare alle pagine HTML anche componenti Java compilati che possono essere scaricati allo scaricamento della pagina Web. Queste vengono chiamate applet e sono componenti class associati a pagine Web che vengono portati al cliente e lì messi in esecuzione automaticamente. Lo scaricamento è attuato per la presenza di una JVM con un ClassLoader associato al browser.

Siccome il codice che viene scaricato da remoto non è mai fidato, questo viene associato ad un controllore delle operazioni, chiamato Security Manager, che può bloccare quelle non consentite durante l'esecuzione prima di ogni operazione.

Nel caso di Java RMI il codice remoto che viene scaricato è quello dell'istanza, dello stub e degli oggetti restituiti come valore di ritorno. Il caricamento di tale codice tramite l'RMI Registry avviene dunque tramite un ClassLoader sul quale può essere attivato un RMISecurityManager. Quest'ultimo effettua il controllo degli accessi bloccando gli accessi che non sono autorizzati da un file di policy che contiene le autorizzazioni.

Per invocare dunque il client e il server specificando il file di policy occorre utilizzare la seguente opzione:

```
// client
java -Djava.security.policy=echo.policy EchoRMIClient
// server
java -Djava.security.policy=echo.policy ddEchoRMIServer
```

Un esempio di file di policy è il seguente:

```
grant {
    /* consente al client e al server di instaurare le
       connessioni necessarie all'interazione remota
       nelle porte specificate */
    permission java.net.SocketPermission "*:1024-65535",
```

```
        "connect, accept";
    /* consente di prelevare il codice da un server http e
       relativa porta 80 */
    permission java.net.SocketPermission "*:80",
        "connect";
    /* consente di prelevare codice a partire dalla radice dei
       direttori consentiti */
    permission java.io.FilePermission "c:\\\\home\\\\RMIDir\\\\-",
        "read";
}
```

Localizzazione del codice

Il cliente, per interagire con l'RMI Registry, utilizza una connessione non persistente, dunque ogni volta che viene effettuato un lookup si crea una nuova connessione e la si chiude al termine.

Inoltre, quando viene invocato qualcosa tramite RMI Registry, prima lo si ricerca all'interno del CLASSPATH locale, e poi nel codebase specificato tramite l'url inserito come parametro nel metodo lookup.

Memorizzazione del codice remoto

Le istanze e le classi scaricate da remoto vengono inserite all'interno dell'heap della JVM.

In un heap gli oggetti richiedono una strategia di allocazione e di deallocazione, che può essere svolta tramite due strategie:

- Reference counting: ogni oggetto mantiene un contatore interno che il numero di riferimenti ad esso, quando diventa 0 viene deallocato.
In RMI si utilizza una strategia distribuita basata sul reference counting.
- Garbage collection: un processo concorrente all'applicazione procede a deallocare gli oggetti/classi che non sono riferiti da nessun altro oggetto.

In Java si lavora con Garbage Collector.

Per quanto riguarda gli oggetti remotizzabili controlla, per ogni connessione su tali oggetti, se i clienti hanno ancora riferimenti a questi, altrimenti li dealloca (soluzione molto costosa, dunque l'RMI Registry viene usato solo per applicazioni in-the-small).

▼ Debug app distribuite

Test della connettività di rete

```
nc -zv <hostname> <porta>
```

Verifica se un'applicazione o un servizio è accessibile su una determinata porta.

Simulazione di un client o un server

Server:

```
nc -l <porta>
```

Mette nc in modalità "ascolto" su una porta specifica.

Client:

```
echo "Test" | nc <hostname> <porta>
```

Invia il messaggio "Test" al server in ascolto.

Redirezione di file per il test

Server:

```
nc -l <porta> > file.txt
```

Client:

```
nc <hostname> <porta> < file.txt
```

Todo

Debug app distribuite con comando nc unix

Fare restante teoria per orale:

OSI

TCP/IP

Servizi applicativi standard internet

Sistemi di nomi

Convertire exams utils e examples in pdf

Come non restituire nulla in RPC

Exercises

Examples

Exams utils