

PER ALTRI APPUNTI CONSULTARE IL SITO:
https://luigi-v.github.io/Appunti_Universita/

1. SISTEMA DISTRIBUITO

Un **sistema distribuito** consiste di un insieme di macchine, ognuna gestita in maniera autonoma, connesse attraverso una rete. Ogni nodo (computer) del sistema distribuito esegue un insieme di componenti che comunicano e coordinano il proprio lavoro attraverso uno strato software detto **middleware**, in maniera che l'utente (del sistema ma anche programmatore e progettista) percepisca il sistema come un'unica entità integrata.

Le caratteristiche di un sistema distribuito sono:

- **Remoto**: le componenti sono locali o remote, quindi "distribuite" su macchine diverse;
- **Concorrenza**: un sistema distribuito è per natura concorrente, due o più istruzioni possono essere eseguite contemporaneamente su macchine diverse, d'altronde risulta essere complicato tale esecuzione perché non esistono *semafori* o *lock* per gestire la sincronizzazione;
- **Assenza di uno stato globale**: non esiste un punto preciso dove controllare lo stato globale dell'intero sistema distribuito, perché essendo i nodi geograficamente distanti non si conosce con certezza lo stato di ogni nodo;
- **Malfunzionamenti parziali**: alcuni componenti distribuiti possono smettere di funzionare, in maniera indipendente dalle altre componenti, e questo fallimento non deve influire sulle funzionalità dell'intero sistema distribuito;
- **Eterogeneità**: per definizione un sistema distribuito è eterogeneo, ovvero costituito da componenti diversi sia hardware che software.
- **Autonomia**, un sistema distribuito non può essere controllato da un singolo punto, in più la collaborazione dei vari nodi va ottenuta mediante le richieste del sistema distribuito con quelle del sistema che gestisce ciascun nodo;
- **Evoluzione**, un sistema distribuito può cambiare in maniera sostanziale durante la sua vita, sia perché cambia l'ambiente sia perché cambia la tecnologia utilizzata;
- **Mobilità**, adattare al meglio le prestazioni del sistema mobilitando i nodi e le risorse (dati).

1.1 MOTIVAZIONI DEL PERCHÉ UN SISTEMA DISTRIBUITO

In generale, i sistemi distribuiti rispondono a motivazioni sia di tipo economico che di natura tecnologica.

Per quanto riguarda il **contesto economico**, i sistemi distribuiti rispondono alle esigenze del mercato che è caratterizzata da numerose e frequenti acquisizioni, integrazioni e fusioni di aziende. Quindi, la necessità di affrontare in tempi brevi l'integrazione dei sistemi informatici di aziende diverse, che si sono fuse insieme, richiede una infrastruttura versatile e agile, che permetta di poter essere operativi in pochissimo tempo. Allo stesso tempo, spesso sistemi informativi di aziende che vengono separate dalla "casa madre", in un meccanismo cosiddetto "**downsizing**", devono mantenere un certo livello di integrazione con le aziende del gruppo, in una sorta di federazione di sistemi che complica la gestione, e per cui si usano i sistemi distribuiti. Un altro aspetto interessante è la "platea" di utenti di internet, che di volta in volta aumentano, quindi deve essere possibile gestire picchi di carico aggiungendo risorse, ovvero altri nodi al sistema, perché i sistemi centralizzati non "scalano".

Per il **contesto tecnologico**, la tecnologia hardware subisce delle evoluzioni velocissime, ogni anno esce hardware più potente e quindi i software e i sistemi che si appoggiano a tali strutture devono essere in grado di reggere di pari passo tale evoluzione, allo stesso tempo però deve essere anche importanti mantenere la compatibilità con sistemi cosiddetti "**Legacy**" (sistemi datati), quindi la progettazione deve puntare a realizzare sistemi distribuiti che siano **flessibili** per poter evolvere e fare evolvere i sistemi distribuiti in maniera da integrare sistemi legacy al proprio interno.

Diverse "leggi" empiriche elaborate negli anni si sono provate fedeli nel prevedere la velocità di evoluzione. Ad esempio, la **Legge di Moore** che afferma che la densità dei transistor nei processori si raddoppia ogni 18 mesi. In pratica, afferma che la potenza di calcolo raddoppia ogni 18 mesi.

Altre leggi riguardano lo sviluppo delle tecnologie di rete come:

- La "legge" di **Sarnoff** dice che il valore di una rete di broadcast è direttamente proporzionale al numero di utenti: $V=a \cdot N$.
- La "legge" di **Metcalfe** dice che il valore di una rete di comunicazione è direttamente proporzionale al quadrato del numero di utenti: $V=a \cdot N+b \cdot N^2$.
- La "legge" di **Reed** dice che il valore di una rete sociale è direttamente proporzionale ad una funzione esponenziale in N : $a \cdot N+b \cdot N^2+c \cdot 2^N$.

NOTA: La parola "legge" viene messa tra virgolette perché è considerata come qualcosa di empirico, non c'è dimostrazione pratica.

1.2 OPEN DISTRIBUTED PROCESSING (RD-ODP)

L'**open distributed processing** è un modello di riferimento per produttori, sviluppatori e progettisti che serve per facilitare lo sviluppo di sistemi distribuiti e va a dettagliare nello specifico quali saranno le funzionalità che un sistema deve offrire, ignorando la specifica implementazione hardware e software. Questo modello RM-ODP ha come obiettivo quello di gestire i problemi di **comunicazione** in un sistema rispetto ai problemi (più semplici) di **connessione**, che vengono trattati principalmente dal modello ISO/OSI. Il modello RM-ODP non si limita, come ISO/OSI, a trattare con i problemi di comunicazione tra sistemi eterogenei, ma punta ad astrarre e standardizzare anche il concetto di portabilità e di trasparenza all'interno di un sistema distribuito. RM-ODP estende ed ingloba, il modello ISO/OSI, usando quest'ultimo come modalità per la comunicazione tra componenti eterogenee.

1.2.1 REQUISITI NON FUNZIONALI DI UN SISTEMA DISTRIBUITO

La realizzazione di un sistema distribuito non è agevole e comporta la necessità di considerare vari aspetti generali e globali della architettura, standardizzati in maniera tale che possano servire da specifica per i vari fornitori di piattaforme hardware e software allo scopo di fornire strumenti adeguati a rendere più agevole la progettazione, implementazione e manutenzione di un sistema distribuito.

Un **requisito non funzionale** è un aspetto che non è direttamente collegato alle funzionalità, ma indica la qualità del sistema e non sono identificabili in una specifica parte del sistema, sono globali e vanno considerati come fattori che hanno un impatto significativo sull'architettura.

Questi requisiti non funzionali specificano che la progettazione deve puntare a realizzare sistemi distribuiti che:

- ...siano **aperti**: in modo da supportare la portabilità di esecuzione e di interoperabilità (capacità di collaborare insieme tra diverse componenti) attraverso *interfacce* e *servizi* ben documentati e aderenti a standard noti e riconosciuti. Questo aspetto risulta importante per poter far evolvere il sistema (si possono aggiungere nuove componenti al bisogno) ma anche per evitare di rimanere legati ad un singolo fornitore, cioè se si usano standard aperti, si può cambiare fornitore senza particolari rischi per l'intera architettura (che può essere riutilizzata ed integrata).
- ...siano **integrali**: così da incorporare al proprio interno sistemi e risorse differenti senza dover utilizzare strumenti ad-hoc. Questo permette di trattare in maniera efficiente (economica) con il problema della eterogeneità hardware, software e di applicazioni.
- ...siano **flessibili**: per poter evolvere e fare evolvere i sistemi distribuiti in maniera da integrare sistemi legacy al proprio interno. Un sistema distribuito dovrebbe anche poter gestire modifiche durante l'esecuzione in modo da poter accomodare cambi a run-time, riconfigurandosi dinamicamente.

- ...siano **modulari**: in modo da permettere ad ogni componente di poter essere autonoma ma con un grado di interdipendenza verso il resto del sistema.
- ...supportino la **federazione** di sistemi: in modo da unire diversi sistemi, dal punto di vista amministrativo oltre che architettonico, per lavorare e fornire servizi in maniera congiunta.
- ...siano **facilmente gestibili**: in modo da permettere il controllo, la gestione e la manutenzione per configurarne i servizi, la loro qualità (Quality of Service) e le politiche di accesso.
- ...forniscano supporto per la **qualità del servizio** (Quality of Service): ha lo scopo di fornire servizi con vincoli di tempo, disponibilità e affidabilità, anche in presenza di malfunzionamenti parziali. La tolleranza ai malfunzionamenti è una delle principali richieste di qualità del servizio di un sistema distribuito, in quanto i sistemi centralizzati sono particolarmente poco tolleranti ai malfunzionamenti, che possono rendere l'intero sistema inutilizzabile. Un sistema distribuito è potenzialmente in grado di trattare con i malfunzionamenti, utilizzando (dinamicamente) componenti alternativi per fornire funzionalità che alcune componenti non sono in grado temporaneamente di fornire.
- ...siano **scalabili**: perché qualsiasi sistema distribuito accessibile da Internet può essere soggetto a picchi di carico non prevedibili e deve essere in grado di gestirli, ma si deve anche poter gestire (anche attraverso la flessibilità) che il sistema possa evolvere per accomodare evoluzioni del contesto aziendale che può crescere velocemente, aumentando notevolmente la platea di utenti che accedono ai servizi forniti dal sistema.
- ...siano **sicuri**: così che utenti non autorizzati non possano accedere a dati sensibili. La sicurezza è particolarmente complicata dalla natura remota dei sistemi distribuiti e della mobilità degli utenti, nodi e risorse al proprio interno.
- ...offrano **trasparenza**: mascherando i dettagli e le differenze dell'architettura sottostante che assicura la distribuzione dei servizi sulle componenti del sistema. Questa caratteristica risulta importante per poter permettere l'agevole progettazione ed implementazione: il progettista o programmatore deve avere un certo grado di indipendenza dai dettagli della distribuzione della architettura. È però anche vero che questa trasparenza non deve essere completa e che progettisti/programmatori debbano essere coscienti della natura distribuita di alcune caratteristiche.

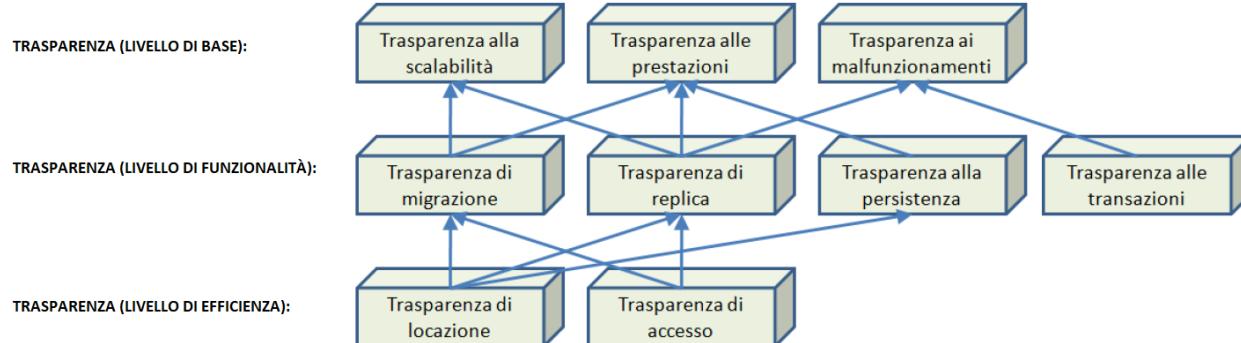
1.2.2 TRASPARENZA DI UN SISTEMA DISTRIBUITO

I dettagli del sistema che offre le funzionalità operative sono nascosti agli utenti, infatti, la **trasparenza** di un sistema non permette di identificare le singole parti ma viene visto come un'unica entità.

Questo permette al progettista/sviluppatore di lavorare in un ambiente che non fornisce informazioni specifiche sulla architettura del sistema, ignorando, ad esempio: l'eterogeneità delle componenti, i fallimenti indipendenti a cui sono soggetti le componenti, l'esistenza e la posizione dei diversi nodi che offrono lo stesso servizio e così via. Il progettista/sviluppatore richiede al sistema di fornire un certo tipo ed un certo livello di trasparenza su una certa parte della sua applicazione, attraverso una richiesta al sistema.

Il primo **vantaggio della trasparenza** è la maggiore produttività del lavoro di sviluppo: il progettista e lo sviluppatore possono concentrare il loro lavoro sull'applicazione, utilizzando un sistema "astratto" di cui ignorano i dettagli. Questo si riflette sulla velocità di prototipizzazione e sull'economia della produzione del software. Ma la trasparenza serve anche a permettere un alto riuso delle applicazioni sviluppate che, proprio perché sviluppate nella totale trasparenza dei dettagli sottostanti, possono essere riutilizzate in contesti diversi e su sistemi diversi.

La trasparenza che viene fornita da un sistema distribuito ricade in diverse tipologie, strettamente collegate e dipendenti l'una dall'altra, come:



- Trasparenza di accesso**: nasconde le differenze nella rappresentazione dei dati e nel meccanismo di invocazione per permettere l'interoperabilità tra oggetti. Questo significa anche che gli oggetti devono essere accessibili attraverso la stessa interfaccia, sia che siano acceduti da locale sia che siano acceduti da remoto. In questa maniera, un oggetto può essere facilmente spostato a run-time da un nodo ad un altro. In generale, questo tipo di trasparenza viene fornito di default dai sistemi, in quanto è il tipo di trasparenza necessario per assicurare l'interoperabilità in un ambiente eterogeneo.
- Trasparenza di locazione**: non permette di utilizzare informazioni circa la localizzazione nel sistema di una particolare componente, che viene identificata ed utilizzata in maniera indipendente dalla sua posizione. Questo tipo di distribuzione fornisce una vista logica del sistema di naming, in modo da disaccoppiare il nome da una posizione all'interno della rete. Anche questo tipo di trasparenza è fondamentale per un sistema distribuito, in quanto senza di esso non si potrebbe spostare componenti da un nodo ad un altro, poiché essi potrebbero essere riferiti secondo la loro posizione e non attraverso un meccanismo di naming logicamente disconnesso dalla locazione.
- Trasparenza di migrazione**: nasconde la possibilità che il sistema faccia migrare un oggetto da un nodo ad un altro, continuando ad essere raggiungibile ed utilizzabile da altri oggetti. Questo viene utilizzato per ottimizzare le prestazioni del sistema (bilanciando il carico tra i nodi oppure riducendo la latenza per accedere ad una componente) o anche per anticipare malfunzionamenti o riconfigurazioni del sistema ma deve essere gestito in maniera automatizzata da parte della infrastruttura del sistema.
La trasparenza di migrazione dipende dalla trasparenza di accesso (che permette di accedere ad un oggetto, anche se locale, solo attraverso la propria interfaccia che viene usata da remoto) e dalla trasparenza di locazione (che nasconde la locazione fisica di un oggetto, permettendone l'accesso attraverso un sistema di naming logico fornito dal sistema).
- Trasparenza di replica**: il sistema maschera il fatto che una singola componente viene replicata da un certo numero di copie (dette repliche) che vengono posizionate su altri nodi del sistema, e che offrono esattamente lo stesso tipo di servizio della componente originale. Ovviamente, il sistema si deve occupare di mantenere assolutamente coerente lo stato di tutte le repliche con la componente originale, in maniera tale da rispettare la semantica delle operazioni che vengono compiute sulla componente e dei servizi offerti.

Anche questo tipo di trasparenza dipende da quella di accesso e di locazione. Le repliche vengono utilizzate per diversi scopi, come per le prestazioni, facendo in modo di replicare componenti laddove (all'interno del sistema) maggiori sono le richieste per quel tipo di servizi, in modo da minimizzare la latenza per accedervi, ma vengono anche utilizzate per poter far scalare il sistema in presenza di aumento del carico di lavoro.

5. **Trasparenza alla persistenza:** scherma l'utente dalle operazioni che compie il sistema per rendere persistente (cioè in memoria secondaria) un oggetto durante una fase di non utilizzo. Infatti, per ottimizzare le prestazioni del sistema, gli oggetti di utilizzo raro non vengono mantenuti attivi (nello spazio di indirizzamento della memoria principale) ma vengono de-attivati, e memorizzati (con il loro stato) all'interno della memoria secondaria, mantenendo solamente un handle per la loro riattivazione, quando arrivano richieste di operazioni da eseguire. In questo caso, l'oggetto viene riportato in memoria principale e reso attivo per rispondere alla richiesta. Gli oggetti che invocano servizi su un oggetto de-attivato non avvertono la differenza con le invocazioni su un oggetto attivo. La trasparenza alla persistenza si basa sulla trasparenza di locazione, in quanto l'accesso indipendente dalla posizione fisica dell'oggetto permette una riattivazione dell'oggetto anche su nodi diversi da quelli su cui era stato de-attivato.

6. **Trasparenza alle transazioni:** (anche chiamata *trasparenza alla concorrenza*) nasconde all'utente le attività di coordinamento che vengono svolte per assicurare la consistenza dello stato degli oggetti in presenza della concorrenza. Sia l'utente che il progettista/sviluppatore sono ignari delle attività che vengono svolte per assicurare l'atomicità delle operazioni e possono semplicemente ritenersi gli unici utenti all'interno del sistema. La gestione delle transazioni distribuite è un compito non semplice e la semplificazione che si ottiene lasciandola al sistema è davvero notevole per lo sviluppatore di applicazioni. La possibilità di poter operare in maniera transazionale una operazione è anche cruciale per assicurare che in presenza di malfunzionamenti una risorsa non si trovi in uno stato non coerente.

7. **Trasparenza alla scalabilità:** è uno dei principali motivi a favore di un sistema distribuito rispetto ad uno centralizzato. Un sistema viene detto scalabile quando è in grado di poter servire carichi di lavoro crescenti senza dover modificare la propria architettura e la propria organizzazione. Progettare un sistema scalabile è necessario visto che la platea di utenti ai quali potenzialmente un servizio su Internet viene offerto è smisurata. Un servizio deve potenzialmente poter scalare dalle poche decine alle centinaia di migliaia di utenti senza che questo comporti la riprogettazione dell'intero sistema. Ovviamente, si dovranno acquisire nuove risorse, ma il sistema dovrà essere in grado di poterle utilizzare senza modifiche sostanziali. La trasparenza alla scalabilità assicura che il progettista/sviluppatore non deve curarsi dei dettagli di come il proprio servizio scalera al crescere delle richieste, ma sarà il sistema che provvederà, attraverso il meccanismo di replica e di migrazione, a fare in modo che le nuove risorse aggiunte al sistema vengano utilizzate per fare fronte al carico crescente. Basato su migrazione e replica.

8. **Trasparenza alle prestazioni:** abbastanza simile alla trasparenza alla scalabilità. Il sistema distribuito che assicura questo tipo di trasparenza rende il progettista/sviluppatore ignaro dei meccanismi che vengono utilizzati per ottimizzare le prestazioni del sistema, durante la fornitura di servizi. In particolare, il sistema può provvedere ad implementare politiche di bilanciamento del carico, spostando componenti da nodi carichi di lavoro verso nodi che hanno maggiori disponibilità di calcolo a disposizione, oppure politiche di minimizzazione della latenza, avvicinando (repliche di) componenti su nodi più vicini (in termini di topologia di rete) agli utenti che li usano più frequentemente, oppure politiche di ottimizzazione delle risorse di memoria, che prevedono la inattivazione di oggetti che non vengono usati frequentemente e che possono essere re-attivati se necessario. Per questo motivo, la trasparenza alle prestazioni si appoggia sulla trasparenza alla migrazione, alla replica ed alla persistenza.

9. **Trasparenza ai malfunzionamenti:** nasconde ad un oggetto il malfunzionamento (e, se è il caso, la successiva recovery) di oggetti con i quali sta interoperando. La trasparenza si estende ovviamente sia agli utenti del sistema che non devono avere la sensazione di malfunzionamenti parziali all'interno del sistema, in quanto il sistema deve automaticamente riconfigurare la richiesta e fornire il servizio in maniera alternativa. Questo tipo di trasparenza si poggia sulla trasparenza di replica, in quanto quest'ultima fornisce la possibilità di poter ripetere trasparentemente le operazioni che si erano iniziate su una replica di un oggetto, potendo rieseguirle su un'altra replica. Ma si basa anche sulla trasparenza alle transazioni, in quanto operazioni complesse eseguite come una transazione, se interrotte a causa di un malfunzionamento non vengono confermate (commit) e quindi non alterano lo stato della risorsa e possono essere ripetute su una replica.

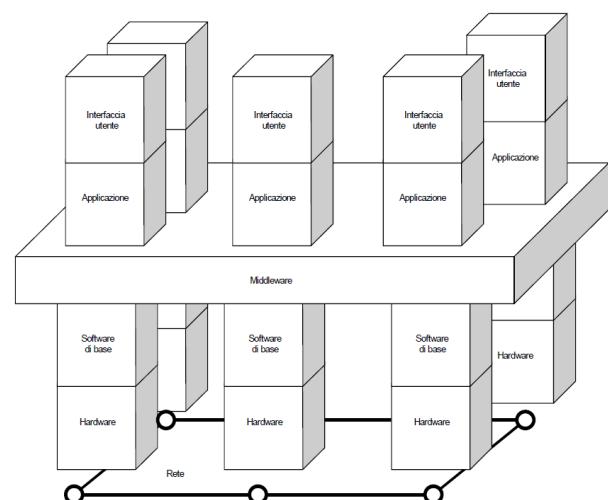
1.3 MIDDLEWARE AD OGGETTI DISTRIBUITI

I sistemi distribuiti basati su Oggetti Distribuiti sono uno degli strumenti utilizzati dai sistemi distribuiti per assicurare **estendibilità, affidabilità e scalabilità, rendendo minimo lo sforzo** per progettare, sviluppare e manutenere sistemi complessi.

Gli **oggetti distribuiti** si trovano nell'unione di due aree della tecnologia software, ovvero tra i **sistemi distribuiti**, che puntano a realizzare un unico sistema integrato basato sulle risorse offerte da diversi calcolatori messi in rete, e lo **sviluppo e programmazione orientata agli oggetti**, che si focalizzano sulle modalità per ridurre la complessità dei sistemi software, creando artefatti software riutilizzabili in diversi contesti.

Gli **oggetti distribuiti**, quindi, hanno come obiettivo quello di realizzare servizi distribuiti riutilizzabili, il tutto basato su una architettura che utilizza come risorse nodi eterogenei, sia per hardware che per software, raggiungibili attraverso una rete. Questa integrazione viene realizzata attraverso il **middleware ad oggetti distribuiti**, che risiede tra le applicazioni e lo strato del sistema operativo, stack di protocolli di rete e hardware, con l'obiettivo di permettere alle componenti del sistema di cooperare e comunicare.

È chiaro che la comunicazione attraverso i nodi della rete potrebbe avvenire attraverso le primitive di comunicazione di rete che vengono offerte dal sistema operativo di ogni singolo nodo, però risulta essere troppo complesso per la programmazione di applicazioni, in quanto i programmati dovrebbero prendersi cura di tutti i dettagli di basso livello (ad esempio, trasformare una struttura dati in stream di byte), risolvendo in prima persona tutti i problemi di rappresentazione dei dati su differenti architetture hardware.



Lo **scopo del middleware**, appunto, è quello di rendere semplici questi compiti e di fornire delle astrazioni appropriate per i programmati, che ben si integrino con i tradizionali strumenti che essi utilizzano per realizzare la applicazione. Il middleware ad oggetti distribuiti viene suddiviso in 3 strati:

1. **Middleware di infrastruttura:** questo layer si occupa delle comunicazioni tra sistemi operativi diversi e della gestione della concorrenza per evitare lo sforzo di utilizzare meccanismi non portabili (dipendendo dalla singola piattaforma hardware/software di ogni nodo) per sviluppare e manutenere applicazioni distribuite. Un esempio di questo tipo di infrastruttura può essere quella della stessa *macchina virtuale Java*.

2. **Middleware di distribuzione:** che basa i suoi servizi sul middleware di infrastruttura per automatizzare operazioni comuni per la comunicazione.

Tra i compiti più importanti ci sono, ad esempio, quelli di:

- richiedere un servizio ad un altro nodo potendo inviare parametri (**marshalling**). Questo compito non è banale in quanto si deve realizzare l'invio di parametri tra diverse piattaforme hardware/software, e, oltre ai problemi di eterogeneità della rappresentazione di tipi primi (tipicamente affrontata dal layer di infrastruttura) si deve poter inviare anche dati complessi (oggetti) la cui definizione (classe) può anche non essere a disposizione della macchina che riceve l'invocazione del servizio;
- utilizzare lo stesso canale di comunicazione (socket, ad esempio) per diverse richieste, oppure utilizzare una sola macro-richiesta che include diverse richieste;
- modificare la semantica delle operazioni di invocazione oltre quella tradizionale di unicast, quale ad esempio l'invocazione in multicast (invocazioni su diversi oggetti contemporaneamente) oppure l'attivazione di oggetti in risposta ad invocazione di servizi;
- riconoscimento e gestione dei malfunzionamenti di rete, attuando strategie per permettere che l'applicazione possa effettuare la recovery di uno stato semanticamente corretto dell'applicazione.

3. **Middleware per servizi comuni di supporto:** un layer che serve a fornire i servizi comuni a tutte le applicazioni distribuite e, quindi, riutilizzabili in tutti i contesti, quali, ad esempio, la persistenza degli oggetti (cioè il loro collegamento automatico ad un sistema di gestione di database), la sicurezza (con gestione di autenticazione e di politiche di accesso), la gestione delle transazioni (assicurando, ad esempio, l'atomicità di operazioni che sono effettuate in un contesto altamente concorrente, quale i sistemi distribuiti).

L'**obiettivo dei 3 livelli di middleware** è quello di assicurare che il programmatore di applicazioni concentri i propri sforzi sullo sviluppo della logica di business dell'applicazione e che non si debba interessare direttamente dei dettagli di comunicazione a livello di rete, il che consuma risorse, tempo e richiede competenze specifiche. In più forniscono astrazioni utili per il programmatore. Infine, proprio perché il middleware ad oggetti astrae la parte di distribuzione e di servizio delle applicazioni distribuite, lo sviluppo avviene ad alto livello, permettendo di poter utilizzare e riutilizzare framework e soluzioni, appoggiandosi a metodologie evolute di ingegneria del software per rendere maggiormente proficua, efficiente ed efficace la soluzione realizzata.

1.3.1 PROGENITORE: REMOTE PROCEDURE CALLS (RPC)

Il **Remote Procedure Calls** fu un modello presentato negli anni '80 che permetteva ad una procedura in esecuzione su una macchina di invocarne un'altra che si trovasse su una macchina diversa. RPC realizzava l'invocazione in maniera agevole per il programmatore, come se fosse stata una tradizionale chiamata di procedura fatta in locale. Infatti, in RPC è stato definito per la prima volta il meccanismo di invocazione remota che richiede la traduzione dei tipi di dato a livello applicazione (usati come parametri e come risultati dell'invocazione di procedura remota) in maniera tale che:

- fosse possibile trasmettere i dati utilizzando stream di byte su socket, in maniera codificata e standardizzata in modo che all'altro capo della comunicazione i dati fossero ricostruiti come erano stati trasmessi (operazione di **Marshalling**);
- si superassero le differenze nella rappresentazione dei dati da trasmettere, ad esempio, la rappresentazione degli interi (con le differenze hardware tra architetture big-endian o little-endian o di diversi linguaggi di programmazione o sistemi operativi) o la rappresentazione delle stringhe di caratteri (che possono essere codificati tramite diversi standard, come ASCII) (**data representation**);

Inoltre, RPC imponeva che fosse rispettata la **sincronia** della invocazione, bloccando il client (processo che invoca il metodo remoto) fino a quando il server (processo che ospita la procedura invocata) non avesse risposto all'invocazione remota di procedura.

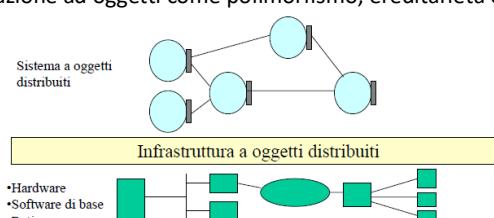
Le operazioni di sincronizzazione, marshalling, data representation e comunicazione tra client e server venivano implementate dai sistemi di RPC attraverso gli **stub**, lato client e lato server che interfacciavano il processo chiamante con il processo che offre la procedura, fornendo al programmatore un'astrazione che facilitava il suo compito di realizzare l'applicazione distribuita. La scrittura degli stub venne poi automatizzata, utilizzando strumenti per creare stub a partire dalla definizione dei servizi offerti attraverso un linguaggio specifico, chiamato **Interface Definition Language (IDL)**, per il quale esistevano le corrispondenze in ciascun linguaggio di programmazione supportato dallo specifico ambiente.

I problemi e difficoltà di RPS sono stati:

- Paradigma procedurale e non come quello ad oggetti;
- I tipi di dati sono solamente elementari (limita i tipi struttura);
- Mancanza gestione eccezioni e malfunzionamenti;
- Manca la concorrenza di più programmi.

1.3.2 DA RPC AL MIDDLEWARE AD OGGETTI DISTRIBUITI

Gli oggetti distribuiti sono un modello presentato negli anni '90 che unisce la tecnologia software della **programmazione ad oggetti** con quella dei **sistemi distribuiti**, il modello **RPC** viene esteso in maniera da permettere l'invocazione di metodi di oggetti remoti. Tale estensione non rappresenta semplicemente un aggiornamento della chiamata a procedure per oggetti, in quanto il modello viene esteso integrando al proprio interno le caratteristiche proprie del modello di programmazione ad oggetti come polimorfismo, ereditarietà e gestione delle eccezioni.



Il passaggio da Remote Procedure Call agli Oggetti Distribuiti è stato motivato dalla evoluzione e dallo sviluppo dei sistemi distribuiti complessi, che, a crescere della disponibilità delle risorse hardware e di comunicazione, diventano ampi, complessi, eterogenei e difficile da gestire e da manutenere.

Storicamente, la prima (1991) proposta significativa di ambiente di programmazione basato su oggetti distribuiti è stato **Common Object Request Broker Architecture (CORBA)**, che è uno standard che permette ad oggetti distribuiti, scritti in diversi linguaggi e quindi eterogenei, di comunicare e collaborare per realizzare una applicazione distribuita. Tutta l'architettura di CORBA si basa sulla invocazione di un servizio (metodo) su un oggetto distribuito. Le invocazioni remote vengono realizzate attraverso il servizio fornito dall'**Object Request Broker (ORB)** che astrae il meccanismo di invocazione in maniera completamente trasparente al client. In seguito, ha riscontrato diversi problemi tra cui la complessità e la mancata interoperabilità tra ORB diversi.

L'effetto è che CORBA ha dovuto lasciare il passo nei sistemi distribuiti a soluzioni basate sul modello a componenti, o **Enterprise computing**, o a soluzioni basate su architetture orientate a servizi (**Service Oriented Architecture**).

Java Remote Method Invocation è stata la proposta di Sun, interna all'ambiente Java, per realizzare applicazioni distribuite basate su oggetti. **RMI** è realizzato in Java ed eredita numerose caratteristiche che ne hanno reso l'utilizzo diffuso, dapprima, per realizzare applicazioni distribuite e, poi, come strumento di comunicazione di base per **Java Enterprise Edition**. Principalmente, Java RMI definisce il Middleware di distribuzione all'interno della piattaforma Java, integrandosi, però, con altre librerie di Java per fornire i servizi comuni di supporto.

Microsoft .NET Framework è la soluzione di Microsoft per la realizzazione di applicazioni ed include un'ampia libreria di soluzioni ed un ambiente di esecuzione chiamato **Common Language Runtime (CLR)** in modo che applicazioni possano essere scritte in uno dei linguaggi supportati da Microsoft.

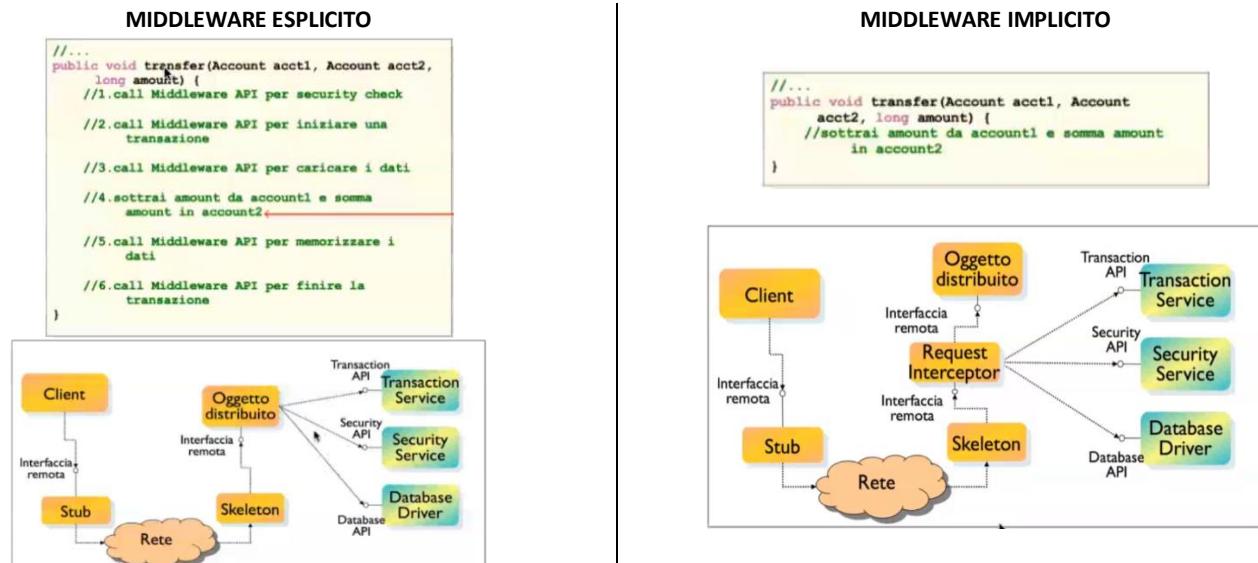
1.4 MIDDLEWARE AD OGGETTI DISTRIBUITI NEL MODELLO A COMPONENTI

Le tecnologie basate su oggetti distribuiti hanno favorito la diffusione dei sistemi distribuiti, queste hanno influenzato lo sviluppo delle tecnologie basate sul **Modello a Componenti Distribuiti** (anche detto *Enterprise Computing*).

Una **Componente Distribuita** è un blocco riutilizzabile di software che può essere combinato in un sistema distribuito per realizzare funzionalità. All'interno di una componente risiedono servizi e applicazioni che espongono tramite un'interfaccia le proprie funzionalità.

Quello che caratterizza e differenzia le componenti da altri moduli software riutilizzabili (come gli oggetti, ad esempio) è che essi possono essere combinati sotto forma di eseguibili binari, piuttosto che sottoforma di azioni da compiere sul codice sorgente.

Inoltre, il modello a componenti si basa sul cosiddetto **middleware implicito** che viene contrapposto alle tecnologie di **middleware esplicito**. Il middleware implicito, attraverso meccanismi di **intervettazione** delle richieste e delle interazioni tra gli oggetti, è in grado di fornire servizi comuni e trasversali ad ogni componente senza che essa debba esplicitamente richiederli all'interno del codice.



Il server che gestisce la componente (detta **application server** o **container**) fornisce questi servizi sulla base delle richieste (codificate non nel codice ma in un file di metadati di descrizione, come XML) specificate quando la componente viene messa a disposizione sul server (fase di **deployment**).

Così i servizi vengono messi a disposizione in maniera completamente trasparente allo sviluppatore di software della componente, realizzando una maggiore interoperabilità tra produttori di software diversi.

Tra i servizi che devono essere messi a disposizione da un sistema a componenti, di particolare importanza sono quelli di fornire un protocollo di comunicazione remota, che permette le interazioni tra le componenti remote, in quanto i meccanismi di comunicazione tra oggetti distribuiti permettono di offrire il supporto per le invocazioni di operazioni tra layer diversi di architetture software.

Ad esempio, uno dei requisiti fondamentali di un sistema distribuito è quello di poter gestire il ciclo di vita di un oggetto distribuito che permette di attivare oggetti su macchine diverse, quando necessario, per assicurare scalabilità e per poter gestire al meglio i malfunzionamenti.

Domande che fa all'orale:

- Cosa è un sistema distribuito?
- Cosa sono i requisiti non funzionali di un sistema?
- Cosa è la trasparenza di accesso? (la fa all'orale, ma più in generale tipo "[Parlami della trasparenza di un sistema distribuito](#)")
- A cosa serve il middleware?
- Cosa significa che gli oggetti distribuiti garantiscono interoperabilità?

2. SOCKET TCP

Nel paradigma di programmazione orientata ad oggetti, la computazione viene effettuata attraverso un insieme di oggetti che contengono uno stato (variabili istanza) ed espongono un comportamento, vale a dire permettono ad altri oggetti di usare i loro metodi. Per poter estendere questo modello di programmazione nell'ambito distribuito, è necessario permettere di invocare un metodo da parte di un oggetto remoto, questo lo si fa mediante i cosiddetti **Socket**.

La comunicazione tra programmi su internet avviene utilizzando la suite di protocolli TCP/IP. La maniera in cui questi protocolli vengono usati è quello di fornire un'astrazione (mediante il software di rete) chiamata socket che permette di ricevere e trasmettere dati.

I **socket TCP** sono degli endpoint di una comunicazione bidirezionale sulla rete che unisce due programmi, ad ogni socket viene assegnato un numero di porta che serve a identificare l'applicazione che è incaricata di dover trattare i dati, che è in esecuzione sul computer che li riceve.

Quindi un socket viene univocamente definito dalla combinazione di **indirizzo IP** e **num di porta**.

Normalmente, si identificano i due computer coinvolti in un socket, col nome di **client** e **server**.

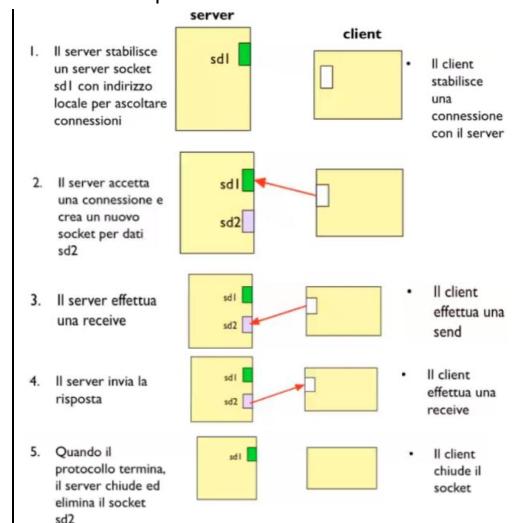
Il **server** è in esecuzione ed attende che qualche client richieda la connessione. Dal lato client, il programma conosce l'indirizzo della macchina su cui è in esecuzione il server ed il numero di porta, in più il client deve anche comunicare al server il numero di porta locale sul quale riceverà i dati (di solito questo viene assegnato dal sistema).

Il procedimento di connessione prevede che il server debba accettare la connessione e che assegna un **nuovo socket** per la comunicazione bidirezionale tra client e server (*socket di comunicazione*). In questa maniera, il server può tornare ad accettare connessioni da altri client.

In questo caso, tipicamente il server lancia un thread per ogni socket stabilito con un client, in modo da permettere la gestione concorrente delle comunicazioni con tutti i client.

Il package **java.net** offre due classi per i socket, proprio per gestire la fase di accettazione da parte dei server e la comunicazione bidirezionale tra client e server:

La **classe Socket** (crea il canale di comunicazione) e la **classe ServerSocket** (accetta connessioni, richiamata dal server), quest'ultima implementa un socket di connessione che attende richieste da parte del client, quando ne riceve una, assegna un socket alla connessione bidirezionale, restituendo l'oggetto **Socket** che viene usato per la comunicazione tra client e server.



2.1 STREAM

La comunicazione tra client e server avviene attraverso la scrittura e la lettura di **stream** (flussi) associati con il socket e che permettono una facile interazione (gestita dal linguaggio) per poter trasmettere istanze di classi Java (oggetti) tra client e server, attraverso un meccanismo di **serializzazione**.

Gli stream di I/O sono una utile astrazione che Java fornisce al programmatore per trattare con una sequenza di dati che può essere "diretta a" / "proveniente da" diverse entità, quali file, periferiche, memoria e ovviamente socket.

Gli stream sono presenti sotto numerose forme: la gerarchia delle classi di stream nel package `java.io` offre un notevole numero.

La sottoclasse più importante che utilizzeremo è **ObjectInputStream** che fornisce il meccanismo di **deserializzazione** quando si riceve un oggetto precedentemente serializzato con **ObjectOutputStream**. Gli oggetti che possono essere trasmessi su questo tipo di stream devono implementare l'interfaccia **Serializable**.

La sequenza di istruzioni classicamente utilizzata per accedere agli stream di un socket lato server è:

```
ServerSocket serverSocket = new ServerSocket(9000);
socket = serverSocket.accept();
System.out.println("Accettata una connessione... attendo comandi");
ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
```

Metodi di Socket:

- **ServerSocket(int port)**: crea un server socket su una specifica porta;
- **Socket accept() throws IOException**: aspetta connessioni sul ServerSocket e lo accetta (metodo bloccante fino a nuova connessione);
- **public void close() throws IOException**: chiude il socket;
- **void setSoTimeout(int timeout) throws SocketException**: setta un timeout per call ad accept, se il tempo passa viene lanciata l'eccezione.

Metodi di InputStream, dati che da una sorgente esterna possono essere usati dal programma:

int	available()	Restituisce una stima del numero di byte che possono essere letti (o ignorati) da questo flusso di input senza essere bloccati dalla successiva chiamata di un metodo per questo flusso di input.
void	close()	Chiude questo flusso di input e rilascia tutte le risorse di sistema associate al flusso.
void	mark(int readlimit)	Contrassegna la posizione corrente in questo flusso di input.
boolean	markSupported()	Verifica se questo flusso di input supporta i metodi mark e reset .
abstract int	read()	Legge il byte di dati successivo dal flusso di input.
int	read(byte[] b)	Legge un certo numero di byte dal flusso di input e li memorizza nella matrice di buffer b.
int	read(byte[] b, int off, int len)	Legge fino a lenbyte di dati dal flusso di input in una matrice di byte.
void	reset()	Riposiziona questo flusso nella posizione in cui il mark metodo è stato chiamato l'ultima volta su questo flusso di input.
long	skip(long n)	Salta e scarta nbyte di dati da questo flusso di input.

Metodi di OutputStream, dati che il programma può inviare:

void	close()	Chiude questo flusso di output e rilascia tutte le risorse di sistema associate a questo flusso.
void	flush()	Svuota questo flusso di output e forza la scrittura dei byte di output memorizzati nel buffer.
void	write(byte[] b)	Scrive i b.length byte dalla matrice di byte specificata in questo flusso di output.
void	write(byte[] b, int off, int len)	Scrive i len byte dalla matrice di byte specificata a partire dall'offset off in questo flusso di output.
abstract void	write(int b)	Scrive il byte specificato in questo flusso di output.

Esempio con i Socket:

Server.java

```
import java.io.*;
import java.net.*;
import java.util.logging.Logger;

public class Server {
    static Logger logger = Logger.getLogger("global");

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(9000);
            logger.info("Socket istanziato, accetto connessioni...");
            Socket socket = serverSocket.accept();
            logger.info("Accettata una connessione... attendo comandi.");
            ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
            String nome= (String) inStream.readObject();
            logger.info("Ricevuto:" + nome);
            outStream.writeObject("Hello " + nome);
            socket.close();
        } catch (EOFException e) {
            logger.severe("Problemi con la connessione:" + e.getMessage());
            e.printStackTrace();
        } catch (Throwable t) {
            logger.severe("Lanciata Throwable:" + t.getMessage());
            t.printStackTrace();
        }
    }
}
```

Client.java

```
import java.io.*;
import java.net.*;
import java.util.logging.Logger;

public class Client {
    static Logger logger = Logger.getLogger("global");

    public static void main(String args[]) {
        try {
            Socket socket = new Socket ("localhost", 9000);
            ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
            out.writeObject("Giovanni");
            System.out.println(in.readObject());
            socket.close();
        } catch (EOFException e) {
            logger.severe("Problemi con la connessione:" + e.getMessage());
            e.printStackTrace();
        } catch (Throwable t) {
            logger.severe("Lanciata Throwable:" + t.getMessage());
            t.printStackTrace();
        }
    }
}
```

Esempio Client-Server coi Socket:

RecordRegistro.java

```
import java.io.Serializable;

public class RecordRegistro implements Serializable {
    private static final long serialVersionUID = -4147133786465982122L;

    // Costruttore
    public RecordRegistro(String n, String i) {
        nome = n;
        indirizzo = i;
    }

    // Metodi accessori
    public String getNome() {
        return nome;
    }
    public String getIndirizzo() {
        return indirizzo;
    }

    // Variabili istanza
    private String nome;
    private String indirizzo;
}
```

Il client si connette al server su *localhost* passando *Giovanni*.

Il server risponde salutando *Giovanni* con *Hello Giovanni*. Nel try instanziamo su socket di connessione sulla porta 9000 e chiamiamo il metodo bloccante *accept()*. Da questo metodo si esce solamente quando un client effettua una richiesta di connessione verso il server, ed il metodo restituisce un socket che è stato stabilito tra il server ed il client.

È su questo socket che usiamo gli stream in input e output che servono, prima, per chiamare il metodo (anche esso bloccante) *readObject()* che restituisce l'oggetto trasmesso (e deserializzato dallo stream) che viene utilizzato per realizzare la risposta.

Ed infine si chiude il socket con *close()*.

Il client apre il socket verso l'host locale dove abbiamo lanciato il server (prima riga del try), preleva gli stream dal socket e scrive il suo nome sullo stream di output (oggetti *out* e *in*).

Il client termina stampando a video quello che ha ricevuto dalla lettura dello stream di input.

NOTA: Particolare attenzione va data all'ordine con il quale si devono aprire gli stream: prima lo stream di output e poi quello di input.

Per concludere, il server può essere facilmente reso iterativo, inserendo la *accept()* e la risposta verso il client all'interno di un *while(true)*. Poi, si può rendere il server multi-thread, in modo che ad ogni *accept*, si faccia partire un thread (che gestisce l'invio della risposta al client) permettendo al server di tornare subito alla *accept()* successiva.

L'esempio implementa sul server un registro che contiene record (composti da due campi stringa, nome e indirizzo) che vengono inseriti e possono essere reperiti specificando solamente il nome.

Realizzeremo un server che, in attesa su una porta, riceve le richieste di inserimento (codificate attraverso un oggetto *RecordRegistro* con tutti i campi riempiti) o le richieste di ricerca (codificate attraverso un oggetto *RecordRegistro* che ha solo il campo nome riempito).

Iniziamo dalla definizione dei record da memorizzare, un semplice wrapper di due campi stringa, con i relativi metodi di accesso.

Le instance sono serializzabili (implementa l'interfaccia *Serializable*), in quanto ci aspettiamo che debbano essere trasmesse su socket TCP in formato binario. In pratica, l'algoritmo di deserializzazione usa questo numero per assicurarsi che la classe appena caricata corrisponda ad un oggetto serializzato. Se questo non corrisponde ai numeri che ha calcolato, allora viene lanciata una eccezione *InvalidClassException*.

RegistroServer.java

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.logging.Logger;

public class RegistroServer {
    static Logger logger = Logger.getLogger("global");

    public static void main(String[] args) {
        HashMap<String, RecordRegistro> hash = new HashMap<String, RecordRegistro>();
        Socket socket = null;
        System.out.println ("In attesa di connessioni...");
        try {
            // Creazione ed accept su socket
            ServerSocket serverSocket = new ServerSocket(7000);
            while (true) {
                socket = serverSocket.accept();
                ObjectInputStream inStream = new ObjectInputStream (socket.getInputStream());
                RecordRegistro record = (RecordRegistro) inStream.readObject();
                if (record.getIndirizzo() !=null) { // si tratta di una scrittura
                    logger.info ("Inserisco:" + record.getNome() + ", " + record.getIndirizzo());
                    hash.put(record.getNome(), record);
                } else { // è una ricerca
                    logger.info ("Cerco:" + record.getNome());
                    RecordRegistro res = hash.get(record.getNome());
                    ObjectOutputStream outStream=new ObjectOutputStream(socket.getOutputStream());
                    outStream.writeObject(res); // se non c'è il record, res è null
                    outStream.flush();
                }
                socket.close();
            } // fine while
        } catch (EOFException e) {
            logger.severe("Problemi con la connessione:" + e.getMessage());
            e.printStackTrace();
        } catch (Throwable t) {
            logger.severe("Lanciata Throwable:" + t.getMessage());
            t.printStackTrace();
        }
        finally { // chiusura del socket e terminazione programma
            try { socket.close();
            } catch (IOException e) {
                e.printStackTrace();
                System.exit(0);
            }
        }
    }
}
```

ShellClient.java

```
import java.io.*;
import java.net.*;
import java.util.logging.Logger;
public class ShellClient {
    static Logger logger = Logger.getLogger("global");

    public static void main(String args[]) {
        String host = args[0];
        String cmd;
        in = new BufferedReader(new InputStreamReader(System.in));
        try {
            while (!(cmd = ask ("Comandi>")).equals("quit")) {
                if (cmd.equals ("inserisci")) {
                    System.out.println ("Inserire i dati.");
                    String nome = ask("Nome: ");
                    String indirizzo = ask ("Indirizzo: ");
                    RecordRegistro r = new RecordRegistro(nome, indirizzo);
                    Socket socket = new Socket (host, 7000);
                    ObjectOutputStream sock_out = new ObjectOutputStream(socket.getOutputStream());
                    sock_out.writeObject(r);
                    sock_out.flush();
                    socket.close();
                } else if (cmd.equals ("cerca")) {
                    System.out.println ("Inserire il nome per la ricerca.");
                    String nome = ask("Nome: ");
                    RecordRegistro r = new RecordRegistro(nome, null);
                    // si invia un oggetto con indirizzo vuoto
                    Socket socket = new Socket (host, 7000);
                    ObjectOutputStream sock_out = new ObjectOutputStream(socket.getOutputStream());
                    sock_out.writeObject(r);
                    sock_out.flush();
                    ObjectInputStream sock_in = new ObjectInputStream(socket.getInputStream());
                    RecordRegistro result = (RecordRegistro) sock_in.readObject();
                    // se viene ottenuto un risultato, allora si stampa l'indirizzo
                    if (result != null)
                        System.out.println ("Indirizzo :" + result.getIndirizzo());
                    else // altrimenti non esiste un record con quel nome
                        System.out.println ("Record non presente");
                    socket.close();
                } else System.out.println (ERRORMSG);
            } // end while
        } catch (Throwable t) {
            logger.severe("Lanciata Throwable:" + t.getMessage());
            t.printStackTrace();
        }
        System.out.println ("Bye bye");
    }

    private static String ask(String prompt) throws IOException {
        System.out.print(prompt+" ");
        return (in.readLine());
    }

    static final String ERRORMSG = "Cosa?";
    static BufferedReader in = null;
}
```

Questo server non fa altro che attendere sulla porta 7000 che ci siano delle richieste di connessione. Il server è iterativo e serve le richieste così come arrivano, quindi senza multi-thread.

Tutti i record che arrivano vanno memorizzati in una HashMap che, come chiave di accesso, utilizza il nome presente nel record.

Nel ciclo infinito il programma accetta connessioni, riceve un oggetto dallo stream in input.

Se l'oggetto ha il campo indirizzo non vuoto, allora è una richiesta di inserimento, che viene effettuata nell'if, altrimenti si tratta di una ricerca, che viene effettuata e restituita sullo stesso socket, ai client.

Se la ricerca è stata infruttuosa, il metodo get() restituisce null che viene restituito ai client.

Per ogni richiesta dell'utente, apre il socket, fa la richiesta al server e scrive la risposta (se necessario), chiudendo il socket. Questo avviene sia per l'inserimento che per la ricerca.

Per l'inserimento, dopo aver chiesto all'utente di digitare le stringhe per comporre l'oggetto RecordRegistro da inviare, apre il socket, usa lo stream di output, invia l'oggetto e chiude il socket.

Per La ricerca, dopo aver chiesto all'utente il nome dei record da ricercare, il client invia un record con il campo indirizzo a null e si attende un oggetto in risposta. Se l'oggetto non è null, allora la ricerca è andata a buon fine e si stampa l'indirizzo dell'oggetto ricevuto, altrimenti si stampa un messaggio, e si chiude il socket.

Il metodo ask() serve per fare input da tastiera in maniera semplice, scrivendo un prompt definito nel programma.

3. PROGRAMMAZIONE CONCORRENTE, PROCESSI E THREAD

Nella programmazione concorrente, ci sono due unità di base di esecuzione: **processi** e **thread**, un sistema informatico ha molti processi e thread attivi. Esistono 3 tipi di programmazione concorrente:

1. Programmazione concorrente eseguita *su calcolatori diversi*;
2. Processi concorrenti *sulla stessa macchina (multitasking)*;
3. Processo padre che *genera processi figli per fork()*.

Un **processo** ha un ambiente di esecuzione autonomo, ovvero uno spazio di memoria privato di risorse di runtime di base.

Un singolo programma, in effetti può essere un insieme finito di processi cooperanti. Per facilitare la comunicazione tra i processi, la maggior parte dei sistemi operativi supporta le risorse **IPC (Inter Process Communication)**, come pipe e socket.

I **thread** (anche chiamati *processi leggeri*) esistono all'interno di un processo (ne ha almeno uno, chiamato **main thread**) ed è un'unità di programma che viene eseguita indipendentemente, in più condividono le risorse del processo stesso, inclusa la memoria e file aperti, attraverso **memoria condivisa**, ciò rende la comunicazione efficiente, ma potenzialmente problematica.

Una **differenza sostanziale tra thread e processi** consiste nel modo con cui essi condividono le risorse, mentre i processi sono di solito fra loro indipendenti, utilizzando diverse aree di memoria ed interagendo soltanto mediante appositi meccanismi di comunicazione messi a disposizione dal sistema, al contrario i thread di un processo tipicamente condividono le medesime informazioni di stato, la memoria ed altre risorse di sistema.

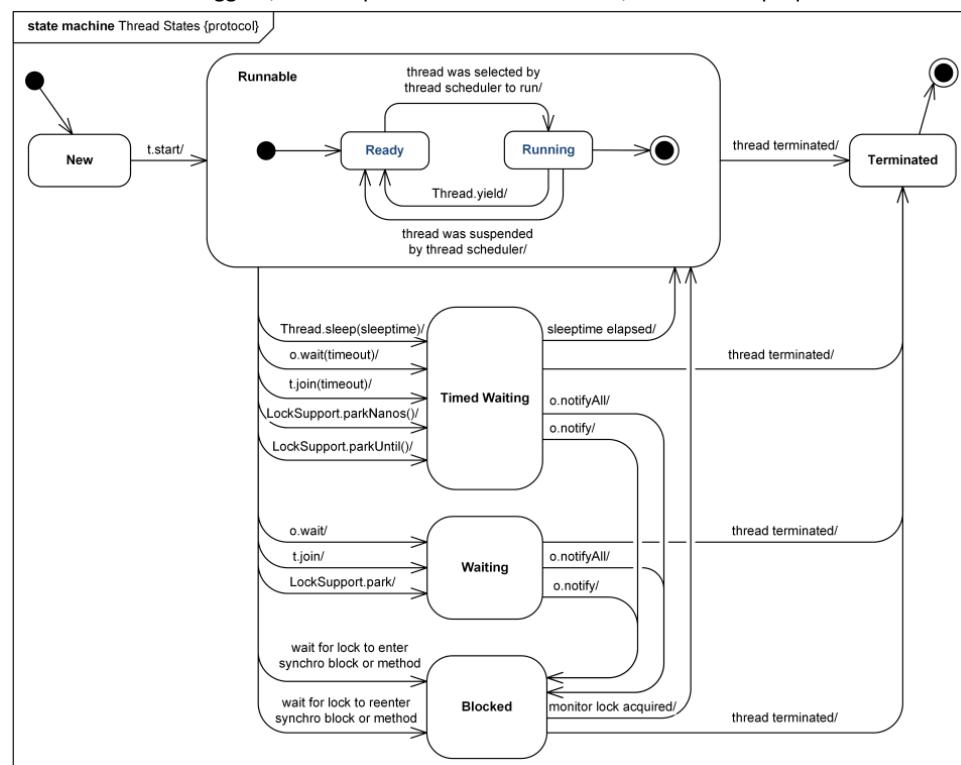
L'altra differenza è che la creazione di un nuovo processo è sempre onerosa per il sistema, in quanto devono essere allocate risorse necessarie alla sua esecuzione (memoria, periferiche, e così via), il thread invece è parte di un processo e quindi una sua nuova attivazione viene effettuata in tempi ridotti e a costi minimi.

Il **multitasking** è la capacità di un sistema operativo di eseguire più compiti (processi) simultaneamente. Il **multithread** è l'estensione del multitask riferita ad un singolo programma in grado di eseguire più thread "contemporaneamente", principalmente esiste un "**Main Thread**" che a sua volta è capace di creare altri thread.

Alcune applicazioni che usano il **multithread** sono i browser che devono caricare dal server diverse immagini, visualizzare la pagina e reagire all'eventuale pulsante premuto dall'utente, oppure un'applicazione di rete che chiede dati ad un'altra applicazione, fornisce dati a chi li richiede e tenere informato l'utente dell'andamento delle operazioni.

3.1 THREAD IN JAVA

I thread in Java sono oggetti, istanze quindi di una classe **Thread**, ed hanno un proprio ciclo di vita:



New - Il thread viene creato.

Ready - Il thread è pronto *per essere eseguito*.

Running - Il thread (le sue istruzioni) *è in esecuzione*.

Timed waiting - Il thread è in attesa di un dato evento.

Waiting - Il thread entra in un'attesa indefinita

Blocked - Il thread è in stato blocked quando sta aspettando di acquisire il lock da un monitor.

Terminated - Il thread ha completato la sua esecuzione.

Un thread passa nello stato *runnable* quando viene chiamato il metodo *start()*.

Un thread passa nello stato **TimeWaiting**, **Waiting** o **Blocked** quando vengono chiamate le funzioni scritte sulle frecce.

Dopo che il thread ha completato l'esecuzione del metodo *run()*, esso viene trasferito nello stato **Terminated**.

Esistono due modalità di **gestione dei thread**:

1. Istanziare un oggetto thread ogni volta che serve un task asincrono (creazione e gestione a cura del programmatore);
2. Astrarre la gestione, passando un task ad un executor.

Noi ci focalizziamo sulla prima modalità, di base:

1. Estendere la classe **java.lang.Thread**;
2. Riscrivere (ridefinire, override) il **metodo run()** nella sottoclasse di Thread;
3. Creare un'istanza di questa classe derivata;
4. Richiamare il **metodo start()** su questa istanza.

Con questo approccio non è possibile estendere tale classe.

Per quanto riguarda la seconda modalità:

1. Implementare l'**interfaccia Runnable**;
2. Riscrivere il **metodo run()** che viene eseguito ogni volta che si lancia il Thread;
3. L'oggetto istanziato è passato al **costruttore di Thread** e lanciato.

Tale approccio è utilizzabile anche per l'approccio con executors.

```
public class HelloThread extends Thread {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}  
  
public class HelloRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

Alcuni metodi utili:

- **Thread.sleep** fa sì che il thread corrente sospenda l'esecuzione per un periodo specificato. Tuttavia, non è garantito che questi tempi di sospensione siano precisi, perché sono limitati dalle funzionalità fornite dal sistema operativo sottostante. Inoltre, il periodo di **sleep** può essere interrotto da un **interrupt**. L'esempio utilizza sleep per stampare i messaggi a intervalli di quattro secondi, in più, il main può lanciare una eccezione che può generare la sleep quando un altro thread interrompe il thread corrente mentre sleep è attivo.

Un **interrupt** indica ad un thread che dovrebbe fermare quello che sta facendo e fare qualcosa altro. Spetta al programmatore decidere esattamente come un thread risponde a un interrupt, ma è molto comune che il thread termini. Un thread invia un interrupt richiamando interrupt sull'oggetto Thread. Affinché il meccanismo di interruzione funzioni correttamente, il thread interrotto deve supportare la propria interruzione. Se il thread chiama frequentemente metodi che generano InterruptedException, vengono restituiti dal metodo run dopo aver rilevato l'eccezione.

Ad esempio, nella sleep() si intercetta l'eccezione e in tal caso si esce altrimenti si stampa.

- **Thread.interrupted()**, un thread che non invoca un metodo che lancia l'eccezione InterruptedException può controllare se è stato interrotto. Nell'esempio, se è stato ricevuto un interrupt allora si lancia l'eccezione gestita in una unica catch() centralizzata.
- **t.join()** consente a un thread di attendere il completamento di un altro thread. Se t è un oggetto Thread il cui thread è attualmente in esecuzione, allora t.join() fa sì che il thread corrente sospenda l'esecuzione fino a quando il thread t non termina.

Esempio SimpleThread:

```
public class SimpleThreads {  
    static void threadMessage(String message) {  
        String threadName = Thread.currentThread().getName();  
        System.out.format("%s: %s%n", threadName, message);  
    }  
    private static class MessageLoop implements Runnable {  
        public void run() {  
            String importantInfo[] = {  
                "Mares", "eat", "Little", "kid"  
            };  
            try {  
                for (int i=0; i<importantInfo.length; i++) {  
                    Thread.sleep(4000);  
                    threadMessage(importantInfo[i]);  
                }  
            } catch (InterruptedException e) {  
                threadMessage("I wasn't done!");  
            }  
        }  
    }  
    public static void main(String args[]) throws InterruptedException {  
        long patience = 1000 * 60 * 60;  
        if (args.length > 0) {  
            try {  
                patience = Long.parseLong(args[0]) * 1000;  
            } catch (NumberFormatException e) {  
                System.err.println("Argument must be an integer.");  
                System.exit(1);  
            }  
        }  
        threadMessage("Starting MessageLoop thread");  
        long startTime = System.currentTimeMillis();  
        Thread t = new Thread(new MessageLoop());  
        t.start();  
        threadMessage("Waiting for MessageLoop thread to finish");  
        while(t.isAlive()) {  
            threadMessage("Still waiting...");  
            t.join(1000);  
            if (((System.currentTimeMillis()-startTime)>patience)  
                && t.isAlive()) {  
                threadMessage("Tired of waiting!");  
                t.interrupt();  
                t.join();  
            }  
        }  
        threadMessage("Finally!");  
    }  
}
```

```
public class SleepMessages {  
    public static void main(String args[])  
        throws InterruptedException {  
        String importantInfo[] = {  
            "Mares", "eat", "Little", "kid"  
        };  
        for (int i=0; i<importantInfo.length; i++) {  
            Thread.sleep(4000);  
            System.out.println(importantInfo[i]);  
        }  
    }  
  
    for (int i = 0; i < importantInfo.length; i++) {  
        // Pause for 4 seconds  
        try {  
            Thread.sleep(4000);  
        } catch (InterruptedException e) {  
            // We've been interrupted: no more messages.  
            return;  
        }  
        // Print a message  
        System.out.println(importantInfo[i]);  
    }  
  
    if (Thread.interrupted()) {  
        throw new InterruptedException();  
    }  
  
    ...  
    t.join();  
    ...
```

SimpleThreads consiste di due thread, il primo è il thread principale di ogni applicazione Java che crea un nuovo thread dall'oggetto *Runnable*, *MessageLoop*, e attende che finisca.

Se il thread *MessageLoop* impiega troppo tempo per terminare, il thread principale lo interrompe.

Tale programma mostra un messaggio col nome del thread, grazie a format, eseguiamo una stampa formattata.

Implementiamo un'interfaccia e riscriviamo run() che viene eseguito allo start del thread.

Al suo interno abbiamo 4 stringhe e poi una sleep() interna ad un blocco try/catch.

Tale blocco stamperà le stringhe con un ritardo di 4 secondi.

Se, nel mentre, si verifica un interrupt, e quindi un'eccezione, il try la cattura e stamperà "I wasn't done!".

Osservando il main, questo può lanciare eccezioni.

All'interno viene calcolata la variabile *patience* che conserva il ritardo della stampa.

All'interno del catch si effettua il controllo del formato.

La variabile *startTime* prende il tempo di inizio.

Infine, viene creato l'oggetto Thread da *MessageLoop* e lo si fa partire richiamando *start()*.

Mentre *MessageLoop* è in vita, grazie al while che lo controlla con *isAlive()*, aspetta al più 1 secondo, specificato dal metodo *join()*.

Se la "pazienza" è scaduta (controllato dall'if) e il thread è ancora vivo, allora lo si chiude con *t.interrupt()*, e se ne attende la "fine" tramite *t.join()*, ed infine si esce stampando "Finally!".

3.2 PROBLEMATICHE COI THREAD

I thread comunicano principalmente condividendo accesso a campi (tipi primitivi) e campi che contengono riferimenti a oggetti.

Questa forma di comunicazione è estremamente efficiente, ma rende possibili due tipi di errori:

1. **Interferenza** di thread;
2. **Inconsistenza** della memoria.

La **prima** problematica (l'**interferenza**) si verifica quando due operazioni, eseguite in thread diversi che agiscono sugli stessi dati, si *interfogliono*.

Avendo questo blocco di codice, se *Counter* fa riferimento a un oggetto da più thread, l'interferenza tra i thread può impedire che ciò accada come previsto.

Supponiamo che il thread A invoca *increment* circa nello stesso momento in cui invoca *decrement*:

1. Thread A: Recupera c.
2. Thread B: Recupera c.
3. Thread A: incremento del valore recuperato; il risultato è 1.
4. Thread B: decrementa del valore recuperato; il risultato è -1.
5. Thread A: Memorizza il risultato in c; c ora è 1.
6. Thread B: Memorizza il risultato in c; c è ora -1.

Il risultato del thread A viene perso, sovrascritto dal thread B, ma può succedere anche il contrario.

Questo scenario porta al cosiddetto **race condition**, cioè quando il risultato di una operazione dipende dall'ordine di esecuzione di diversi thread.

La **seconda** problematica (l'**inconsistenza**) avviene quando thread diversi hanno visioni diverse degli stessi dati.

Fortunatamente, il programmatore non ha bisogno di una comprensione dettagliata di queste cause, tutto ciò che serve è una strategia per evitarli.

Una di queste è la relazione **happens-before**, che è una garanzia che la memoria scritta da un thread è visibile da un altro thread.

Un esempio →

Il campo *counter* è condiviso tra due thread, A e B, se le due istruzioni vengono eseguite in thread separati, il valore stampato *potrebbe* essere "0", perché non c'è garanzia che la modifica del thread A sarà visibile al thread B, e quindi bisogna stabilire una relazione happens-before.

```
class Counter {  
    private int c = 0;  
    public void increment() {  
        c++;  
    }  
    public void decrement() {  
        c--;  
    }  
    public int value() {  
        return c;  
    }  
}
```

```
int counter = 0;  
//...  
counter++;  
//...  
System.out.println(counter);
```

Esempio Inconsistenza della memoria:

```
class Foo {  
    int bar = 0;  
    public static void main(String args[]){  
        new Foo().unsafeCall();  
    }  
    void unsafeCall () {  
        final Foo thisObj = this;  
        Runnable r = new Runnable () {  
            public void run (){  
                thisObj.bar = 1;  
            }  
        };  
        Thread t = new Thread(r);  
        t.start( );  
        try{  
            Thread.sleep(1000);  
        }catch(InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("bar="+ bar );  
    }  
}
```

Dichiariamo una variabile *bar*, definiamo *unsafeCall* nel main.

unsafeCall definisce un thread che mette la variabile *bar* a "1".

Dopo si fa partire il thread, ed il main thread attende 1 sec.

Se si lancia questo programma, parte il main thread che lancia il thread interno che imposta la variabile *bar* ad "1", e il main thread attende 1 sec e poi termina.

La stampa non è precisa, non è detto che il thread fa in tempo a stampare il valore corretto della variabile, potrebbe essere 0 o 1.

Per realizzare **happens-before** lo si fa tramite la **sincronizzazione**, più precisamente, tramite il metodo **start()** e **join()**.

Un'altra soluzione è rendere la **variabile volatile**, cioè modificarne il valore influisce immediatamente sulla memoria. Ciò garantisce che quando un thread modifica la variabile, tutti gli altri thread vedano immediatamente il nuovo valore. La **keyword volatile** è di solito associata ad una variabile il cui valore viene salvato e ricaricato in memoria ad ogni accesso senza utilizzare i meccanismi di **caching**.

Esempio variabile volatile:

```
public class VolatileTest {  
    volatile boolean running = true; // notare la keyword volatile  
    public void test() {  
        //lancio primo Thread  
        new Thread(new Runnable() {  
            public void run() {  
                int counter = 0;  
                while (running)  
                    counter++;  
                System.out.println("Thread 1 concluso.Contatore "+counter);  
            }  
        }).start();  
        new Thread(new Runnable() { //lancio secondo Thread  
            public void run() {  
                try {  
                    Thread.sleep(100);  
                } catch (InterruptedException ignored) { }  
                System.out.println("Thread 2 concluso");  
                running = false;  
            }  
        }).start();  
    }  
    public static void main(String[] args) {  
        new VolatileTest().test();  
    }  
}
```

All'interno di *test()*, lanciamo il primo thread che mette il contatore a "0", poi finché *running* è "true", incrementa il contatore e dopo stampa il suo valore.

Lo *sleep()* nel secondo thread è necessario per dare al primo thread la possibilità di partire. Questo secondo thread mette *running* a "false".

Se la variabile *running* non era volatile, il primo thread non termina mai, perché viene letto sempre il valore della cache, ovvero "true", cioè le modifiche che fa il secondo thread non vengono viste dal primo thread. Dichiarendo volatile la variabile *running* invece si costringe il Thread (o chi per esso) ad aggiornare di volta in volta il valore senza memorizzarlo in cache.

3.3 SINCRONIZZAZIONE DEI THREAD IN JAVA

Il linguaggio di programmazione Java fornisce due idiomati di sincronizzazione di base: **metodi sincronizzati** e **synchronized statements**.

METODI SINCRONIZZATI:

I **metodi sincronizzati** (**synchronized**) sono un costrutto del linguaggio Java, che permette di risolvere gli errori di concorrenza, al costo di inefficienza.

Per rendere un metodo sincronizzato, basta aggiungere **synchronized** alla sua dichiarazione →

Grazie a questa tecnica non è possibile che due esecuzioni dello stesso metodo sullo stesso oggetto siano *interfogliate*. Quando un thread esegue un metodo sincronizzato per un oggetto, gli altri thread che invocano metodi sincronizzati dello stesso oggetto sono sospesi fino a quando il primo thread non ha finito. Quando un thread esce da un metodo sincronizzato, allora si stabilisce una relazione **happens-before** con tutte le successive invocazioni dello stesso metodo sullo stesso oggetto, cioè *i cambi di stato effettuati dal thread appena uscito sono visibili a tutti i thread*.

```
public class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void increment(){  
        c++;  
    }  
    public synchronized void decrement(){  
        c--;  
    }  
    public synchronized int value() {  
        return c;  
    }  
}
```

SYNCHRONIZED STATEMENT:

Per realizzare i **synchronized statements** si utilizza il **lock intrinseco**, che è una entità associata ad ogni oggetto e garantisce sia accesso esclusivo sia accesso consistente (relazione **happens-before**).

Per convenzione, un thread che necessita di un accesso esclusivo deve acquisire il lock intrinseco dell'oggetto prima di accedervi e quindi rilasciare il lock quando ha terminato. Finché un thread possiede un **lock intrinseco**, nessun altro thread può acquisire lo stesso blocco (o statement), l'altro thread si bloccherà quando tenterà di acquisire il blocco.

Quando un thread esegue un metodo sincronizzato di un oggetto ne acquisisce il lock, e lo rilascia al termine (anche se c'è una eccezione).

A differenza dei metodi sincronizzati, gli statement sincronizzati devono specificare l'oggetto che fornisce il lock intrinseco.

In questa maniera, si sincronizzano gli accessi solo durante la modifica, ma poi si provvede in maniera concorrente all'inserimento in lista.

```
public void addName(String name){  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}  
  
public class MsLunch {  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
    public void inc1() {  
        synchronized(lock1) {  
            c1++;  
        }  
    }  
    public void inc2() {  
        synchronized(lock2) {  
            c2++;  
        }  
    }  
}
```

Gli statement sincronizzati sono utili anche per migliorare la concorrenza con la sincronizzazione.

Nelle funzioni inc1() e inc2(), se c'era synchronized prima del nome, si sequenzializza tutto, mentre, se c'era synchronized(this), non sarebbero stati indipendenti.

In questo modo, i lock vengono acquisiti sugli oggetti.

L'ultimo meccanismo per garantire la sincronizzazione, sono le **azioni atomiche**.

Nella programmazione, un'**azione atomica** è un'azione che effettivamente avviene tutta in una volta e non sono interrompibili. Un'azione atomica non può fermarsi nel mezzo: o viene completamente, o non avviene affatto. Nessun effetto collaterale di un'azione atomica è visibile fino al suo completamento. Ad esempio, le operazioni di read e write.

Tuttavia, ciò non elimina la necessità di sincronizzare le azioni atomiche, poiché sono ancora possibili errori di coerenza della memoria. L'utilizzo delle variabili volatili riduce il rischio di errori di consistenza della memoria, poiché qualsiasi scrittura su una di essa stabilisce una relazione happens-before con le letture successive della variabile stessa. Ciò significa che le modifiche a una variabile volatile sono sempre visibili agli altri thread.

Stesso codice di prima, ma col package **java.util.concurrent.atomic**, questo ci garantisce azioni atomiche, e quindi la sincronizzazione.

```
import java.util.concurrent.atomic.AtomicInteger;  
class AtomicCounter {  
    private AtomicInteger c = new AtomicInteger(0);  
    public void increment() {  
        c.incrementAndGet();  
    }  
    public void decrement() {  
        c.decrementAndGet();  
    }  
    public int value() {  
        return c.get();  
    }  
}
```

3.4 PROBLEMATICA DELLA SICRONIZZAZIONE DI THREAD

Possono avvenire alcune problematiche durante la sincronizzazione di thread, una di queste è il **deadlock**, che descrive una situazione in cui due o più thread sono bloccati per sempre, in attesa l'uno dell'altro.

Un esempio →

Alphonse e Gaston sono due amici molto cortesi ed hanno una rigida regola di cortesia: quando ti inchini ad un amico, devi rimanere inchinato finché il tuo amico non ha una possibilità di restituire l'inchino. Ma se i metodi *bow* (inchino) vengono invocati assieme, entrambi i thread si bloccheranno invocando *bowBack*.

Quando Deadlock viene eseguito, è estremamente probabile che entrambi i thread si blocchino quando tentano di invocare *bowBack*. Nessuno dei due blocchi finirà mai, perché ogni thread aspetta che l'altro esca da *bow*.

Riscriviamo i metodi synchronized in questa maniera →

Così che il lock degli oggetti sono esplicativi. A questo punto Alphonse acquisisce il suo lock e cerca di acquisire quello di Gaston, questo fa lo stesso, ovvero prima il suo e poi quello di Alphonse.

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name){ this.name = name; }
        public String getName(){ return this.name; }

        public synchronized void bow(Friend bower) {
            System.out.format("%s:%s "+" has bowed to me!%n",this.name,bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s:%s "+" has bowed back to me!%n",this.name,bower.getName());
        }
    }
    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");

        new Thread(new Runnable(){ public void run() { alphonse.bow(gaston); }}).start();
        new Thread(new Runnable(){ public void run() { gaston.bow(alphonse); }}).start();
    }
}

public void bow(Friend bower){
    synchronized(this) {
        System.out.format("%s:%s has bowed to me!%n", this.name, bower.getName());
        bower.bowBack(this);
    }
}
public void bowBack(Friend bower){
    synchronized(this) {
        System.out.format("%s:%s has bowed back to me!%n", this.name, bower.getName());
    }
}
```

Altre problematiche, meno comuni, sono:

- **Starvation (ingordo)**, descrive una situazione in cui un thread non è in grado di ottenere l'accesso regolare alle risorse condivise e non è in grado di fare progressi. Ciò accade quando le risorse condivise vengono rese non disponibili per lunghi periodi da thread "avid". Ad esempio, si supponga che un oggetto fornisca un metodo sincronizzato che spesso impiega molto tempo per essere restituito. Se un thread richiama questo metodo frequentemente, altri thread che richiedono anche un accesso sincronizzato frequente allo stesso oggetto verranno spesso bloccati.
- **Livelock**, un thread A può reagire ad azioni di un altro thread B che reagisce con una risposta verso A. I due thread non sono bloccati (non è un deadlock!) ma sono occupati a rispondere alle azioni dell'altro, anche se sono in esecuzione, non c'è progresso. Un esempio è due persone che si incontrano in un corridoio stretto, sullo stesso lato, se le due persone hanno un "*attitudine belligerante*", entrambi aspettano che l'altro si sposti, ma questo provoca un deadlock, mentre se hanno un "*attitudine garbata*", entrambi si possono spostare sullo stesso lato, bloccandosi ancora e continuare a spostarsi all'infinito, e questo è un livelock.

3.5 EFFICIENZA DEI THREAD

Una non efficienza dei thread è il cosiddetto **race condition**, ovvero che il risultato finale dell'esecuzione dei processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguiti.

È necessario comprendere bene in che maniera i metodi synchronized sono eseguiti in mutua esclusione. Per questo motivo presentiamo un esempio, semplice, che al variare di alcuni semplici keyword si comporta in maniera diversa.

Nell'esempio →

Abbiamo un oggetto *Example*, creiamo 2 *SimpleThread* e li lanciamo col metodo *start()*. Questo metodo farà partire l'esecuzione dei 2 thread, ovvero richiameranno il metodo *run()*.

Nella classe *Example* abbiamo due metodi sincronizzati (*firstWaitingRoom* e *secondWaitingRoom*), che non fanno altro che stampare l'ingresso all'interno dei metodi, si attende qualche secondo specificato ed infine stampa quando si esce.

```
import java.util.logging.Logger;
public class Example {
    private static Logger log = Logger.getLogger("Example");
    public static void main(String[] args) {
        Example ob = new Example();
        log.info("Creating threads");
        SimpleThread one = new SimpleThread("one", ob,2);
        SimpleThread two = new SimpleThread("two", ob,2);
        one.start();
        two.start();
    }
    public synchronized void firstWaitingRoom(int sec, String name){
        log.info(name+"..entering.");
        try{
            Thread.sleep(sec*1000);
        }catch(InterruptedException e) {
            e.printStackTrace();
        }
        log.info(name+"..exiting.");
    }
    public synchronized void secondWaitingRoom(int sec, String name){
        log.info(name+"..entering.");
        try{
            Thread.sleep(sec*1000);
        }catch(InterruptedException e) {
            e.printStackTrace();
        }
        log.info(name+"..exiting.");
    }
}
```

La classe *SimpleThread* prevede, oltre a variabili di istanza e costruttore, il metodo *run()* che calcola un valore a caso e se questo valore è "0" allora viene avviato il metodo *firstWaitingRoom* altrimenti *secondWaitingRoom*.

```
public class SimpleThread extends Thread {
    private String name;
    private Example object;
    private int delay;
    public SimpleThread(String n, Example obj, int del) {
        name = n;
        object = obj;
        delay = del;
    }
    public void run() {
        double y = Math.random();
        int x = (int) (y * 2);
        if(x==0)
            object.firstWaitingRoom(delay, name);
        else//x==1
            object.secondWaitingRoom(delay, name);
    }
}
```

In questo esempio, abbiamo vari casi possibili di esecuzione, in quanto ci sono 2 thread che cercano di accedere a due metodi(*firstWaitingRoom* e *secondWaitingRoom*) oppure entrambi allo stesso.

In generale, si possono provare strategie differenti per *provare* a risolvere l'interfogliamento:

1. DUE METODI DI INSTANZA SINCRONIZZATI:

Supponiamo che una volta debbano accedere alla stessa sala d'attesa e un'altra a diverse sale, ed entrambi i metodi sono sincronizzati.

Nel momento in cui entrambi sono sincronizzati, succede che è come diventassero sequenziali.

Output di accesso alla stessa sala

```
6-ott-2013 17.14.51 Example main
INFO: Creating threads
6-ott-2013 17.14.51 Example secondWaitingRoom
INFO: one.. entering.
6-ott-2013 17.14.53 Example secondWaitingRoom
INFO: one.. exiting.
6-ott-2013 17.14.53 Example secondWaitingRoom
INFO: two.. entering.
6-ott-2013 17.14.55 Example secondWaitingRoom
INFO: two.. exiting.
```

Output di accesso a sale diverse

```
6-ott-2013 17.15.08 Example main
INFO: Creating threads
6-ott-2013 17.15.08 Example firstWaitingRoom
INFO: one.. entering.
6-ott-2013 17.15.10 Example firstWaitingRoom
INFO: one.. exiting.
6-ott-2013 17.15.10 Example secondWaitingRoom
INFO: two.. entering.
6-ott-2013 17.15.12 Example secondWaitingRoom
INFO: two.. exiting.
```

2. UN SOLO METODO SINCRONIZZATO:

Supponiamo che un solo metodo sia sincronizzato e che si accede a due sale di attesa diverse, non ci sarà mutua esclusione.

Output:

```
6-ott-2013 17.14.04 Example main
INFO: Creating threads
6-ott-2013 17.14.04 Example secondWaitingRoom
INFO: one.. entering.
6-ott-2013 17.14.04 Example firstWaitingRoom
INFO: two.. entering.
6-ott-2013 17.14.06 Example firstWaitingRoom
INFO: two.. exiting.
6-ott-2013 17.14.06 Example secondWaitingRoom
INFO: one.. exiting.
```

```
public synchronized void firstWaitingRoom(int sec, String name){
    log.info(name+"..entering.");
    try{
        Thread.sleep(sec*1000);
    }catch(InterruptedException e) {
        e.printStackTrace();
    }
    log.info(name+"..exiting.");
}
public synchronized void secondWaitingRoom(int sec, String name){
    log.info(name+"..entering.");
    try{
        Thread.sleep(sec*1000);
    }catch(InterruptedException e) {
        e.printStackTrace();
    }
    log.info(name+"..exiting.");
}
public synchronized void firstWaitingRoom(int sec, String name){
    log.info(name+"..entering.");
    try{
        Thread.sleep(sec*1000);
    }catch(InterruptedException e) {
        e.printStackTrace();
    }
    log.info(name+"..exiting.");
}
public void secondWaitingRoom(int sec, String name){
    log.info(name+"..entering.");
    try{
        Thread.sleep(sec*1000);
    }catch(InterruptedException e) {
        e.printStackTrace();
    }
    log.info(name+"..exiting.");
}
```

Esempio efficienza Thread:

```
public class ArrayInit extends Thread{
    public static int[] data;
    public static final int SIZE = 10000000;
    public static void main(String[] args){
        data = new int[SIZE];
        long begin, end;
        int j;
        begin = System.currentTimeMillis();
        for(int i = 0; i < SIZE; i++){
            for(j=0; j < 10000; j++)
                data[i] = i;
        }
        end = System.currentTimeMillis();
        System.out.println("Time:" + (end - begin) + "ms");
    } //end main
} //end class
public class EffInit extends Thread {
    private int start;
    private int dim;
    public static int[] data;
    public static final int SIZE = 10000000;
    public static final int MAX_THR = 16;
    public static void main(String[] args){
        data = new int[SIZE];
        long begin, end;
        int start, j;
        EffInit[] threads;
        for(int numThread = 1; numThread <= MAX_THR; numThread++) {
            begin = System.currentTimeMillis();
            start = 0;
            threads = new EffInit[numThread];
            for (j = 0; j < numThread; j++) {
                threads[j] = new EffInit(start, SIZE/numThread);
                threads[j].start();
                start += SIZE / numThread;
            }
        }
    }
}
```

Tale esempio non fa altro che inizializzare 10 mila volte 10 milioni di elementi, e permette di calcolare il tempo impiegato.

Possible output:

```
Time 20 ms
```

Ora vogliamo ottimizzare questa operazione, e quindi dividiamo l'array in blocchi e si fanno partire diversi thread, così che ognuno di questi thread si occuperà di inizializzare un sottoinsieme dell'array, ottimizzando così il tempo.

Possible output:

```
1 Thread(s): 65ms
2 Thread(s): 11ms
3 Thread(s): 6ms
4 Thread(s): 7ms
5 Thread(s): 6ms
6 Thread(s): 4ms
7 Thread(s): 7ms
8 Thread(s): 5ms
```

```

for (int numThread = 1; numThread <= MAX_THREAD; numThread++) {
    begin = System.currentTimeMillis();
    start = 0;
    threads = new EffInit[numThread];
    for (j = 0; j < numThread; j++) {
        threads[j] = new EffInit(start, SIZE / numThread);
        threads[j].start();
        start += SIZE / numThread;
    }//end for
    for (j = 0; j < numThread; j++) {
        try {
            threads[j].join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }//end for
    end = System.currentTimeMillis();
    System.out.println(numThread+"Thread(s): "+(end-begin)+"ms");
}
}//end main
public EffInit(int start,int size){
    this.start = start;
    this.dim = size;
}//end costruttore
public void run(){
    int j;
    for(int i = 0; i <this.dim; i++){
        for(j=0; j < 10000; j++)
            data[this.start + i] = i;
    }//end for
}
}

```

3.6 SINGLETON E DOUBLE-CHECKED LOCKING

Il **Singleton**, noto design pattern della programmazione ad oggetti, permette di restringere l'istanziazione da parte di una classe ad un solo oggetto.

Dei possibili usi sono le memorizzazioni di stato, come Printer, File, Resource Manager,

Questo pattern, però, soffre di **lazy allocation**, ovvero l'allocazione avviene solo quando utilizzato per la prima volta.

Ci sono diverse possibili implementazioni della classe **Singleton**:

1.

C'è una variabile di classe che mantiene l'istanza, un metodo di classe che restituisce un Singleton, se non è stato ancora creato una istanza allora la crea, alla fine, restituisci la nuova istanza a disposizione.

Con questa implementazione, se ci sono più thread, il loro *interfogliamento* può creare errori, perché si potrebbero creare più Singleton, cosa inaccettabile.

2.

Simile al precedente, ma il metodo è sincronizzato.

Questo dovrebbe risolvere il problema, ma non è così, perché la sincronizzazione serve solo la prima volta che viene invocato il metodo.

3.

Sempre simile alla prima, ma col lock esplicito interno al metodo.

Ma anche questo non funziona..., perché se due thread A e B arrivano ad eseguire l'if, assumiamo che A entra nella sezione critica, crea l'istanza ed esce restituendo il "primo" Singleton, ma a questo punto B entra e crea un nuovo Singleton che sovrascrive il primo.

DOUBLE-CHECKING LOCKING:

Con questa strategia effettuiamo due volte il controllo.

Nonostante pubblicato e ampiamente utilizzato, non è garantito che funzioni.

La chiamata al costruttore "sembra" essere prima dell'assegnazione di *instance*, ma questo non è detto che accada. Un Singleton non inizializzato potrebbe essere assegnato a *instance* e se un altro thread controlla il valore poi lo può usare (in maniera scorretta).

Per risolvere questi problemi è possibile rendere la variabile *instance* **volatile**, altrimenti si possono usare le classi statiche "Initialization-on-demand holder".



```

class Singleton {
    private static Singleton instance;
    private Singleton() { /*...*/ }
    public static synchronized Singleton getInstance(){
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

```

```

class Singleton {
    private static Singleton instance;
    private Singleton() { /*...*/ }
    public static Singleton getInstance(){
        if(instance == null) {
            synchronized(Singleton.class) {
                instance = new Singleton();
            }
        }
        return instance;
    }
}

```

```

public static Singleton getInstance(){
    if(instance == null) {
        synchronized(Singleton.class) {
            if(instance == null)
                instance = new Singleton();
        }
    }
    return instance;
}

```

Domande che fa all'orale:

- Ciclo di vita di un thread?
- Cosa significa interferenza e inconsistenza della memoria?
- Cos'è la relazione happens-before?

4. JAVA REMOTE METHOD INVOCATION (RMI)

Java Remote Method Invocation (Java RMI) è una libreria di Java che permette lo sviluppo di applicazioni distribuite, fornendo la possibilità di effettuare comunicazione remota tra programmi scritti in Java. Infatti, **Java RMI** offre ad un oggetto in esecuzione su una Java Virtual Machine la possibilità di invocare metodi di un oggetto in esecuzione in una altra JVM, anche se essa si trova su una macchina differente.

Il ruolo che viene ricoperto da Java RMI all'interno della Java Platform è quello di *integration library* (libreria per l'integrazione).

Le applicazioni RMI seguono un'architettura client-server dove il server crea un certo numero di *oggetti server* accessibili da remoto e attende che gli *oggetti client* ne utilizzino i servizi, compiendo invocazioni remote sui metodi che espongono.

Gli obiettivi principali che si poneva la realizzazione di Java RMI sono:

1. Invocazione trasparente di metodi remoti:

Java RMI offre al programmatore un meccanismo semplice per l'invocazione di metodi che sono offerti da un oggetto remoto, e deve avvenire fornendo l'"illusione" che essa avvenga su un oggetto che risiede all'interno dello stesso spazio di indirizzamento utilizzato dall'oggetto che compie l'invocazione.

2. Integrazione in Java:

Il modello distribuito si integra all'interno del linguaggio Java standard, che permette di offrire un ambiente familiare allo sviluppatore e può usare gli stessi strumenti, modelli e astrazioni che vengono utilizzati per oggetti locali. Java RMI fornisce un **garbage-collector** distribuito in modo da preservare la modalità di gestione della memoria di Java che solleva il programmatore dal doversi occupare esplicitamente della memoria.

3. Non-trasparenza della natura locale/remota di un oggetto:

Nonostante l'obiettivo di assicurare la semplicità di uso per il programmatore, esistono diverse caratteristiche del linguaggio che non devono essere nascoste al programmatore. Quindi, il fatto che un oggetto sia remoto oppure locale deve essere chiaro ed evidente.

4. Rendere minima la complessità di client e server:

Si deve assicurare la minima complessità all'applicazione distribuita basata su Java RMI. In particolare, il livello di complicazione che viene introdotto da un oggetto distribuito per l'oggetto clienti (cioè l'oggetto che compie l'invocazione remota) e per l'oggetto server (cioè l'oggetto che riceve ed esegue l'invocazione) deve essere limitato.

5. Preservare la sicurezza fornita da Java:

Il modello a oggetti distribuito fornito da Java RMI non deve alterare il livello di sicurezza che viene offerto dalla piattaforma Java. Infatti, sin dalla presentazione del linguaggio, la "sicurezza" e la "robustezza" del linguaggio sono state al centro delle attenzioni dei progettisti, principalmente a causa della natura distribuita del linguaggio, che prevede la esecuzione in locale di programmi che vengono scaricati dalla rete.

6. Modalità di invocazione:

Java RMI deve prevedere la possibilità che esistano diversi tipi di invocazione, fornendo quella di tipo **unicast** da un client verso un server ma permettendo (in futuro) anche l'estensione verso invocazioni di tipo **multicast** vale a dire verso diversi server replicati. Inoltre, deve essere possibile che l'oggetto server sia attivato solo al momento dell'invocazione e che i riferimenti ad oggetti persistenti siano persistenti.

7. Livelli di trasporto multipli:

Infine, Java RMI deve essere aperto verso future espansioni che prevedano che il protocollo di trasporto (basato su **socket**) possa essere modificato.

GARBAGE COLLECTION:

Essendo la memoria una risorsa finita, bisogna gestirla al meglio. Esistono, in generale, due filosofie di gestione della memoria, la prima è gestita completamente dal programmatore mediante funzioni, come free() e malloc() in C, e quindi è più facile commettere errori, però il programmatore ha il completo controllo, mentre la seconda è fornire un servizio per la allocazione/deallocazione assolutamente trasparente al programmatore, la cosiddetta **garbage collection**, dove tutti gli oggetti vengono automaticamente allocati/deallocati quando il sistema lo ritiene necessario, in quanto esso mantiene traccia dei riferimenti attivi ad ogni oggetto e quindi si ha maggiore produttività in quanto la progettazione e l'implementazione possono ignorare la gestione della memoria.

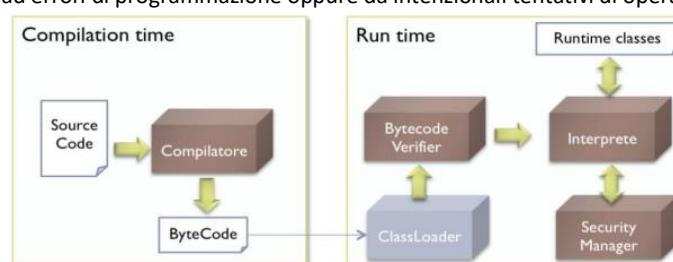
La **Garbage Collection** in locale funziona mantenendo e calcolando il numero di riferimenti (tramite tecnica **reference counting**) che fanno riferimento ad un oggetto, se un oggetto non è più riferito allora è candidato ad essere eliminato dall'heap alla prossima esecuzione del **Garbage Collector**.

La proposta di **Java Remote Method Invocation** fornisce un sistema di **Garbage Collection** per gli oggetti remoti. Questo meccanismo si basa su una estensione della **garbage collection** locale, la JVM tiene traccia di tutti i riferimenti all'oggetto remoto che risultano essere attivi (*live*). Alla prima invocazione di un oggetto, la JVM ritiene quell'oggetto referenziato e quindi da non eliminare. Al termine delle invocazioni, il client fa in modo di inviare un messaggio di dereferenziazione dell'oggetto e, la JVM server quindi considera quel riferimento debole (*weak*) e quindi passibile di eliminazione alla prossima invocazione del **garbage collector**.

Questa tecnica però ha alcuni problemi, ad esempio, il client potrebbe chiudersi per qualche malfunzionamento, oppure potrebbe perdere la connessione verso il server, ed il server si troverebbe con un oggetto remoto che risulta essere referenziato (e quindi da non passare al garbage collector) ma in effetti non sarà mai utilizzato dal client, e quindi rappresenta un potenziale problema di **memory leak**. A questo scopo si introduce il meccanismo di **lease**, ovvero ogni riferimento che viene assegnato al client ha un tempo di vita specificato. Al termine del periodo, se non vengono effettuate altre invocazioni, il server considera quel riferimento non più valido e quindi l'oggetto diventa *weak* e collezionabile dal garbage collector. Questo significa che, in generale, il programmatore che scrive l'oggetto client deve prevedere che il **lease** possa scadere e, per evitare che l'oggetto server sia eliminato, fornire dei metodi che (a intervalli di tempo prefissati) provveda a "rinnovare" il lease, effettuando delle chiamate fittizie a metodi che non hanno nessun effetto.

SICUREZZA IN JAVA:

La sandbox fornita dal linguaggio permette di eseguire applicazioni (e applet) in maniera tale che le operazioni che esse compiono siano controllate e ristretta così da prevenire danni dovuti ad errori di programmazione oppure da intenzionali tentativi di operazioni illegali.



Java fornisce la **sandbox** basandosi su **4 livelli di sicurezza**:

1. Sicurezza del linguaggio, Java è fortemente tipizzato quindi tutte le variabili hanno un tipo definito a tempo di compilazione e solamente (poche) implicite conversioni (casting) vengono effettuate dal compilatore e dalla macchina virtuale a run-time, tutte le altre operazioni di casting devono essere esplicitate dal programmatore. Poi, Java offre la gestione automatica della memoria attraverso un meccanismo di **garbage collection**, che impedisce di esaurire lo spazio di indirizzamento del processo. L'assenza di puntatori e l'impossibilità di poter fare aritmetica o assegnamenti con i riferimenti rende impossibile effettuare accessi illegali in memoria. Inoltre, l'accesso alla memoria reale non viene determinato a tempo di compilazione ma a tempo di esecuzione, non si può conoscere in anticipo in che zona di memoria verranno memorizzati gli oggetti e non si può scrivere codice per alterarli. Infine, a tempo di esecuzione vengono controllati i limiti di array per prevenire accessi a elementi non esistenti.

2. Classloader, si occupa di caricare la classe a tempo di esecuzione, anche da locazioni remote. Il suo compito principale è quello di caricare la classe in un **namespace** separato rispetto a quello delle classi locali, in modo che classi del linguaggio built-in, locali, non possono essere rimpiazzate da altre. Infatti, quando si fa riferimento ad una classe, viene prima cercata tra le classi del sistema locale (built-in) e, solo successivamente, nel **namespace** della classe dalla quale viene riferita.

3. Bytecode Verifier, dopo che il **Classloader** ha caricato una classe per l'esecuzione, il **Bytecode verifier** controlla che essa non sia "volontariamente" ostile, che non ci siano stack underflow/overflow e violazioni alla regole specificate dai modificatori di accesso.

4. Security Manager, si occupa di definire i confini della sandbox. Il **Security Manager** viene interpellato dalla macchina virtuale per ciascuna operazione potenzialmente pericolosa, e fornisce le autorizzazioni sulla base della policy che ha stabilito l'utente lanciando la macchina virtuale.

4.1 MODELLO AD OGGETTI DISTRIBUITI DI JAVA RMI

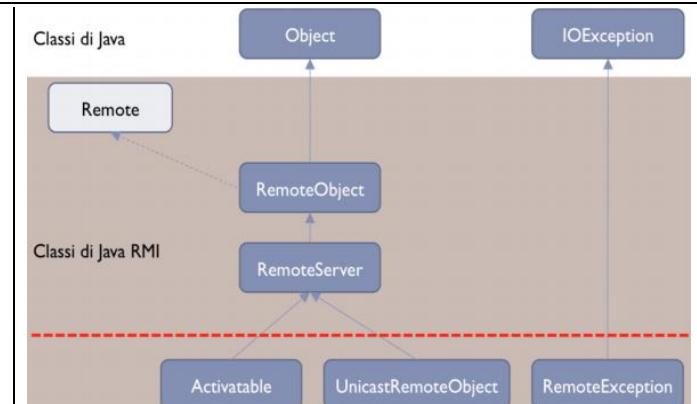
Un **oggetto remoto** è un oggetto i cui metodi possono essere acceduti da un altro spazio di indirizzamento, e potenzialmente da un'altra macchina. La descrizione dei servizi offerti da remoto da un oggetto remoto è contenuta all'interno di una **interfaccia remota** che è una interfaccia Java che dichiara i metodi remoti. Una **invocazione di metodi remoti (Remote Method Invocation)** rappresenta l'invocazione di un metodo su un oggetto remoto (specificato nell'interfaccia remota) e ha la stessa sintassi di un'invocazione di un metodo locale.

L'oggetto client di *oggetti remoti server* utilizza esclusivamente l'interfaccia remota dell'oggetto, non la sua implementazione, questo garantisce che le funzionalità remote risultino astratte verso il client e disaccoppia le due implementazioni, permettendo, ad esempio, evoluzioni dell'oggetto server (cioè della sua implementazione) senza che il client debba essere modificato.

4.1.1 STRUTTURA DELLE CLASSI JAVA RMI

Java RMI è contenuto in 5 package:

- **java.rmi** e **java.rmi.server** che contengono il meccanismo basilare di funzionamento delle invocazioni remote;
- **java.rmi.activation** per gli oggetti attivabili;
- **java.rmi.dgc** per la Distributed Garbage Collection;
- **java.rmi.registry** per il servizio di localizzazione.



INTERFACCE ED ECCEZIONI REMOTE:

Prima di definire un oggetto remoto, si deve definire un'interfaccia remota per l'oggetto, in modo che vengano esposti i servizi che l'oggetto remoto intende mettere a disposizione per un utilizzo da parte dei client.

Un'**interfaccia remota per Java RMI** deve estendere (implementare) l'interfaccia **java.rmi.Remote** che è un'interfaccia cosiddetta **marker**, cioè un'interfaccia vuota che serve solamente per poter segnalare che essa definisce dei metodi accessibili da remoto.

Ogni metodo descritto in un'interfaccia remota deve essere un **metodo remoto**, cioè deve soddisfare entrambe le seguenti condizioni:

1. Un metodo remoto deve dichiarare esplicitamente l'**eccezione java.rmi.RemoteException**, poiché la semantica dei malfunzionamenti di un oggetto remoto è diverso da quella dei malfunzionamenti di un oggetto locale. In questa maniera si forza lo sviluppo successivo, che comporterà l'invocazione di questi metodi, a gestire l'eccezione remota in maniera esplicita, in quanto il compilatore controlla che l'eccezione sia gestita.
2. I **parametri remoti** di un metodo remoto devono essere dichiarati tramite la propria interfaccia remota, non utilizzando la classe dell'implementazione remota. Questo permetterà di poter passare riferimenti remoti sia come parametri che come valori restituiti.

Il meccanismo dell'interfaccia remota aggiunge un livello ulteriore di accessibilità ai *modificatori di accesso dei metodi (public, protected, etc.)*.

I metodi remoti, dichiarati in un'interfaccia remota sono più accessibili dei metodi public, che risultano accessibili ma solamente da invocazioni all'interno della stessa macchina virtuale. Infine, come nelle interfacce (anche remote) sia possibile definire delle costanti.

IMPLEMENTAZIONI REMOTE:

Per realizzare l'implementazione remota che deriva (*implements*) da un'interfaccia remota per offrire verso l'esterno i metodi remoti in essa definiti, si può procedere in due modi:

1. **Riuso dell'implementazione remota**, prevede che la classe che contiene l'implementazione dell'oggetto derivi esplicitamente da **java.rmi.server.UnicastRemoteObject** ereditando di conseguenza il comportamento definito dalle classi **java.rmi.server.RemoteObject** e **java.rmi.server.RemoteServer**;
2. **Classe di implementazione locale**, permette che la classe derivi il comportamento da qualche altra classe (non remota) e che si debba quindi occupare esplicitamente di esportare l'oggetto (tramite il metodo statico **exportObject()** di **java.rmi.server.UnicastRemoteObject**) e di implementare la semantica di alcune operazioni di **Object** per oggetti remoti che sono ridefiniti in **java.rmi.server.RemoteObject** e **java.rmi.server.RemoteServer**.

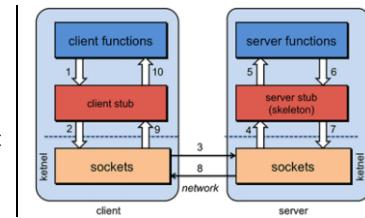
Va sottolineato come l'implementazione remota può implementare anche altri metodi, oltre a quelli remoti definiti nell'interfaccia remota ma che questi saranno accessibili esclusivamente in locale, secondo le direttive di accesso fornite dai modificatori di accesso.

RIFERIMENTI REMOTI:

Nel modello distribuito, gli oggetti client interagiscono con un oggetto **stub** che espone localmente le stesse interfacce remote definite dall'oggetto remoto.

In questa maniera, lo **stub** rappresenta l'interfaccia remota dell'oggetto remoto in locale, sulla macchina dove l'oggetto client è in esecuzione.

Dal punto di vista della JVM del client, il tipo dello **stub** è lo stesso di quello dell'oggetto server, quindi, il client può accedere tradizionalmente al tipo di un oggetto remoto, controllando quale interfaccia remota implementa, attraverso *instanceof*.



CARICAMENTO DINAMICO DELLE CLASSI:

Java Remote Method Invocation permette il passaggio di oggetti come parametri, valori restituiti o eccezioni, attraverso la **serializzazione**. Questo però crea problemi, in quanto, il caricamento delle classi a tempo di esecuzione risulta essere più complesso nel momento in cui stiamo passando ad un metodo offerto da un server remoto (ad esempio) un parametro che è una istanza di una sottoclasse della classe dichiarata nella firma del metodo. In questo caso, l'oggetto remoto può trovarsi nella situazione in cui non conosce esattamente come è strutturata la classe di cui l'oggetto passato è istanza. Questo viene risolto da Java RMI attraverso il caricamento dinamico delle classi. Quando si fa il marshalling degli oggetti per la trasmissione (ad esempio, come parametri nella invocazione da client a server), essi vengono anche annotati con il **codebase**, cioè con la Uniform Resource Locator (URL) di un server WWW da dove è possibile trovare la definizione della classe (cioè il file .class). Quando viene effettuato l'unmarshalling dell'oggetto, il **Classloader** cerca di risolvere il nome della classe nel suo contesto, poi, in caso non sia possibile, viene acceduta la definizione della classe per poter ricreare l'oggetto all'altro capo della comunicazione (nel nostro esempio, sul server).

LOCALIZZAZIONE E INVOCAZIONE DI OGGETTI REMOTI:

Per poter invocare il metodo remoto di un oggetto remoto, l'oggetto client deve avere a disposizione il riferimento remoto, può essere:

1. **Ottenuto come risultato di altre invocazioni (locali o remote) di metodi**, questa è standard per quanto riguarda le invocazioni locali;
2. **Attraverso un servizio di directory**, Java RMI fornisce un semplice meccanismo di *name server* nella classe **java.rmi.Naming**, che permette di gestire riferimenti a oggetti remoti accessibili specificando un ID (stringa). Tale classe fornisce metodi per ricercare (*lookup()*), registrare (*bind()*, *unbind()* e *rebind()*) ed elencare (*list()*) gli identificativi registrati, accedendo al servizio attraverso una specifica che segue lo standard Uniform Resource Locator (URL).

L'invocazione di un metodo remoto ha la stessa sintassi di un'invocazione locale. Poiché i metodi remoti devono includere **RemoteException** nella propria firma, il codice dell'oggetto client viene forzato dal compilatore a gestire questo possibile malfunzionamento della chiamata remota, in aggiunta ad altre eccezioni che dipendono dalla semantica dell'applicazione.

PASSAGGIO DI PARAMETRI:

Un metodo remoto può dichiarare solo parametri o valori restituiti che siano serializzabili, vale a dire che implementino l'interfaccia **Serializable**. Le classi dei parametri o dei valori restituiti che non siano disponibili localmente, vengono scaricate dinamicamente.

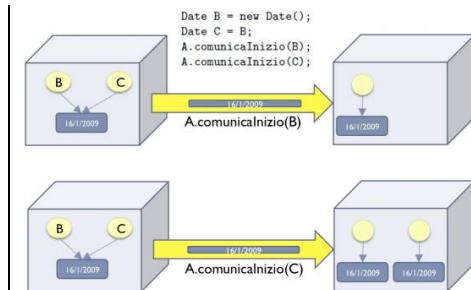
Un oggetto locale passato come parametro o restituito come valore da un'invocazione remota viene **passato per copia**, vale a dire che il contenuto dell'oggetto viene copiato prima di essere serializzato dal meccanismo di serializzazione di Java. Quindi la semantica per il passaggio di parametri locali è diversa da quella di Java che, ricordiamo, passa il riferimento di un oggetto. Quindi, quando vengono passati come parametri più riferimenti allo stesso oggetto ad un'altra macchina virtuale utilizzando invocazioni diverse allora i riferimenti sulla macchina server saranno ad oggetti distinti.

Ad esempio:

Se A è un riferimento ad un oggetto remoto che offre il metodo remoto **comunicalnizio(Date x)**, e se B e C sono riferimenti a oggetti locali di tipo **Date**, dopo avere eseguito il codice:

```
Date B = new Date();
Date C = B;
A.comunicalnizio(B);
A.comunicalnizio(C);
```

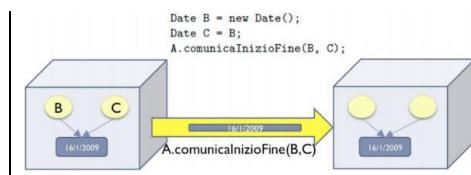
Avremo che sulla macchina remota, ci saranno due oggetti diversi di tipo **Date()** a cui punteranno le due variabili, mentre sulla macchina locale, dalla quale provenivano, essi facevano riferimento allo stesso oggetto.



Il meccanismo implementato da Java RMI assicura la cosiddetta **integrità referenziale**, ovvero quando vengono passati più riferimenti allo stesso oggetto nella stessa invocazione, allora viene garantito che anche sulla macchina remota alla quale sono stati passati i riferimenti punteranno allo stesso oggetto.

Continuando l'esempio precedente, se l'oggetto A fornisce anche il metodo remoto **comunicalnizioFine(Date x, Date y)**, allora dopo avere eseguito il codice:

```
Date B = new Date();
Date C = B;
A.comunicalnizioFine(B, C);
```



Avremo che sulla macchina remota ci sarà un solo oggetto **Date** a cui punteranno i due riferimenti passati come parametri.

Quindi quando si passa un oggetto remoto come parametro o si ottiene come valore restituito, viene passato il suo stub, e non l'implementazione. Questo funzionamento è realizzato col **meccanismo di Marshalling**.

MECCANISMO DI MARSHALLING:

Fare il marshalling di un oggetto in Java significa effettuare una serializzazione modificando la semantica dei riferimenti remoti (invece di un riferimento remoto viene inserito lo stub dell'oggetto remoto) e aggiungendo informazioni all'oggetto (il **codebase** della classe dell'oggetto).

Il marshalling di Java RMI si basa sulla specializzazione del meccanismo tradizionale di serializzazione effettuata da **ObjectOutputStream**. Infatti, questa classe offre la possibilità di poter modificare il comportamento tramite il quale gli oggetti vengono scritti come stream di byte. Più precisamente, la specializzazione del meccanismo di serializzazione per il marshalling avviene modificando tre metodi della classe **ObjectOutputStream**:

- **replaceObject()**, può definire un metodo alternativo per serializzare un oggetto sullo stream, deve essere richiamato ogni volta invocato **writeObject()**;
- **enableReplaceObject()**, restituisce un booleano e stabilisce se l'istanza deve oppure no specializzare il meccanismo di serializzazione, usando il metodo **replaceObject()**;
- **annotateClass()**, che permette di inserire informazioni addizionali sulla classe, viene usato per specificare il codebase e permettere quindi il caricamento dinamico.

4.2 ARCHITETTURA DI JAVA RMI

Il sistema di Java RMI è strutturato su tre livelli (layer):

- **Stub/Skeleton Layer** che comprende gli stub lato client e gli skeleton lato server;
- **Remote Reference Layer** che specifica il comportamento dell'invocazione e la semantica del riferimento (unicast, multicast, etc.);
- **Transport Layer** che si occupa della connessione e della sua gestione.

L'applicazione dell'utente si trova in cima a questi livelli e interagisce (in maniera moderata) con il livello di stub/skeleton.

Un **client** che invoca un metodo su un oggetto server remoto, fa uso di uno **stub** per portare a termine la sua richiesta in quanto il riferimento remoto che è in possesso del **client** è solamente un riferimento al suo **stub**, presente in locale.

Lo **stub** implementa l'interfaccia remota dell'oggetto remoto (verso il **client**) e inoltra le richieste all'oggetto server attraverso il **remote reference layer** del **client**.

Il **remote reference layer** del **client** si occupa di gestire la semantica delle invocazioni remote lato client (se, ad esempio, si tratta di una invocazione unicast o multicast).

Il **livello di trasporto** si occupa di stabilire la connessione con la macchina remota e della successiva gestione della connessione, curando il **dispatching** delle invocazioni verso gli oggetti remoti. A questo scopo, inoltra la richiesta al livello di reference del server.

Il livello di reference del server si occupa di inoltrare la richiesta allo **skeleton** e di curare la semantica dell'invocazione lato server (gestendo, ad esempio, i riferimenti ad oggetti persistenti per curarne la loro attivazione).

STUB/SKELETON LAYER:

Essendo il livello più alto di Java RMI, esso si occupa di essere l'interfaccia tra l'applicazione (cioè le classi Java scritte dal programmatore) ed il resto dei sistemi. L'interfaccia verso il basso, in direzione del **remote reference layer**, consiste nel fornire uno stream di Marshal di oggetti Java che vengono passati (per copia) al **remote reference layer**.

Lo **stub** è incaricato di:

- Iniziare la connessione con la macchina virtuale remota, chiamando il **remote reference layer**;
- Effettuare il **marshalling** verso uno stream di marshal, fornito dal **remote reference layer**;
- Attendere il risultato della invocazione;
- Effettuare l'**unmarshalling** dei valori restituiti (o delle eccezioni verificatesi);
- Restituire il valore verso l'oggetto client che ha richiesto l'invocazione.

Lo **skeleton** è incaricato di effettuare il dispatching verso l'oggetto remoto, vale a dire, curare che l'invocazione sia effettuata sull'oggetto remoto in attesa di invocazioni. Quando uno **skeleton** riceve un'invocazione in entrata, si occupa di:

- Effettuare l'**unmarshalling** del **remote reference layer** (lato server) dei parametri per l'invocazione;
- Invocare il metodo sull'implementazione che si trova nella sua JVM;
- Effettuare il **marshalling** del valore restituito (compreso di eventuali eccezioni) verso chi ha invocato il metodo.

REMOTE REFERENCE LAYER:

Questo layer si occupa di interfacciare il livello di trasporto con quello di **stub/skeleton** fornendo e supportando la semantica dell'operazione di invocazione di un metodo. In un senso, si occupa della parte di protocollo indipendente dall'invocazioni remote specifiche, che vengono gestite da stub e da skeleton. Infatti, ogni implementazione di oggetto remoto può scegliere un protocollo generale che determina le modalità di invocazione, fissato per la durata di vita dell'oggetto. Ad esempio, diverse sarebbero le modalità permessa alle invocazioni:

- Invocazioni **unicast**, vale a dire da un singolo client verso un singolo server;
- Invocazioni **multicast**, vale a dire un singolo client fa una invocazione ad una "batteria" di server replicati, in maniera da poter garantire la ridondanza: se uno di questi è in esecuzione allora risponderà all'invocazione;
- Invocazioni di **oggetti attivabili**: le invocazioni potrebbero essere effettuate ad un oggetto remoto che è persistente, vale a dire viene attivato se arrivano delle invocazioni;
- Invocazioni con **riconnessione**: le invocazioni potrebbero tentare connessioni alternative se l'oggetto remoto originariamente contattato non risponde all'invocazione.

Il protocollo di invocazione utilizza le due componenti client e server del **remote reference layer**. Il lato client ha informazione circa il server della invocazione (se **unicast**) e comunica attraverso il livello di trasporto verso il lato server dello stesso layer.

Durante l'invocazione, da **stub a skeleton** e ritorno, il **remote reference layer** può intervenire per forzare uno specifico protocollo di invocazione, ad esempio, nel caso di un'invocazione multicast la parte client del livello può inoltrare la richiesta ad un insieme di server e selezionare la prima risposta che arriva, scartando le altre. Il lato server, invece viene utilizzato per l'attivazione di oggetti persistenti in caso d'invocazione remota.

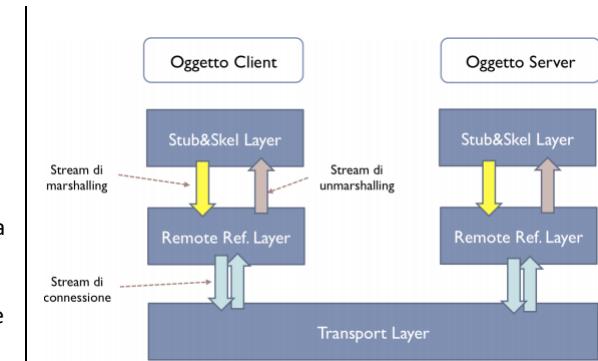
Questo layer fornisce verso l'alto (lo stub/skeleton layer) un riferimento ad un oggetto che implementa l'interfaccia `java.rmi.server.RemoteServer`, che espone un metodo `invoke()` per effettuare l'inoltro dell'invocazione, che viene chiamato dallo stub.

Il **remote reference layer** interagisce verso il basso col **livello di trasporto**, verso il quale utilizza l'astrazione di una connessione orientata ai flussi (stream di connessione). Questa astrazione viene fornita dal livello di trasporto che non è obbligato a realizzare una connessione e potrebbe utilizzare protocolli connectionless senza alterare la modalità con cui il **remote reference layer** comunica i dati.

TRANSPORT LAYER:

Il livello di trasporto ha il compito di:

- Stabilire la connessione verso macchine con indirizzi IP remoti;
- Gestire le connessioni e monitorare il loro stato;
- Rimanere in ascolto per connessioni in arrivo;
- Gestire una tabella degli oggetti remoti che risiedono nello spazio di indirizzamento locale;
- Stabilire una connessione per le chiamate in entrata;
- Identificare l'oggetto dispatcher a cui inoltrare la connessione.



4.3 PROCESSO DI CREAZIONE DI UN PROGRAMMA JAVA RMI

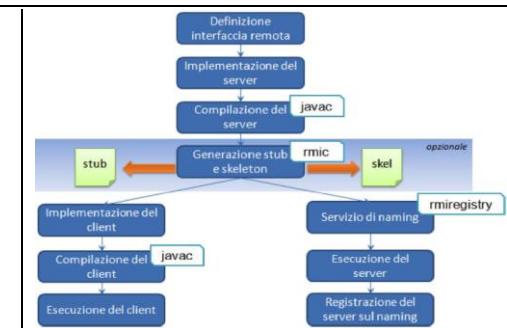
Il processo per la creazione di un programma Java RMI può essere riassunto dalla Figura e si suddivide, dopo alcuni passi preliminari, in due sotto-processi:

- uno che procede verso lo sviluppo ed esecuzione del server;
- l'altro che va in direzione dello sviluppo e della esecuzione dei client.

Alcuni passi sono dipendenti dai precedenti e altri, invece, possono proseguire indipendentemente.

In particolar modo, lo sviluppo del client risulta essere solo minimamente influenzato da quello del server.

Il programmatore lato client ha necessità esclusivamente della interfaccia remota (ed eventualmente dello stub) per potere proseguire nello sviluppo del programma client.



DEFINIZIONE INTERFACCIA REMOTA:

Il primo passo consiste nella definizione di quali sono i servizi che vengono offerti dal nostro server, specificati all'interno di una interfaccia. Questa interfaccia rappresenta il "contratto" che vincola il server ad offrire determinati servizi se il client utilizza l'interfaccia che il server espone.

L'implementazione, ovviamente, sarà a carico dei server e totalmente nascosta al client.

La interfaccia remota per RMI deve avere le seguenti caratteristiche:

- L'interfaccia deve derivare dalla interface mark-up [Remote](#);
- Tutti i metodi remoti devono lanciare l'eccezione [java.rmi.RemoteException](#).

Un classico errore che viene fatto è quello di inserire nell'interfaccia remota "a tappeto" tutti i metodi del server, anche quelli che sono locali.

IMPLEMENTAZIONE DEL SERVER:

Un oggetto remoto in Java deve essere istanza di una classe che:

- Implementi una o più interfacce remote (cioè che estendano [Remote](#));
- Derivi da [java.rmi.UnicastRemoteObject](#).

Il fatto che il server derivi da [UnicastRemoteObject](#) implica che il costruttore del nostro server debba esplicitamente essere scritto (anche vuoto) in quanto è necessario che il costruttore della sottoclasse (il nostro server) lanci esplicitamente la eccezione [UnicastRemoteObject](#) che viene lanciata dal costruttore della superclasse.

COMPILAZIONE DEL SERVER:

In generale, una buona IDE lo fa automaticamente, ma è importante sapere che vengono messi i file `.class` che vengono creati dal compilatore `javac`. Se si accettano i valori di default allora i sorgenti vengono messi in una directory `src` mentre i file in bytecode vengono messi in una directory `bin` che non viene visualizzata. Per poterla visualizzare è necessario utilizzare una *view Navigation*.

COMPILAZIONE CON LO STUB COMPILER rmic:

Avendo specificato interfaccia remota e server (che deriva da [UnicastRemoteObject](#)) si possono generare automaticamente i file che sono necessari (stub e skeleton) senza che debbano essere scritti dal programmatore. A questo scopo, RMI fornisce uno *stub compiler*, chiamato `rmic` che, eseguito sul file `.class` del nostro server, genera lo stub (e lo skeleton). Quindi, dobbiamo eseguire lo stub sul file `.class` del nostro server. A tale scopo possiamo definire una applicazione esterna da chiamare dall'interno di Eclipse in modo da poterlo eseguire selezionando (nel Navigator) il file `.class` e generare lo stub del nostro oggetto server. Una volta fatto ciò fare refresh della cartella.

SERVIZIO DI NAMING rmiregistry:

A questo punto è necessario che il nostro oggetto remoto possa essere accessibile dai client che ne vogliono invocare i servizi. A tale scopo, Java RMI propone un servizio di naming (abbastanza semplice) chiamato [rmiregistry](#). Questo programma deve essere lanciato prima di eseguire l'oggetto server, in quanto il server (tipicamente) tra le prime operazioni farà in modo di registrarsi presso il servizio di naming con una etichetta. I client che vogliono accedere ai servizi di un oggetto remoto, fanno una richiesta (operazione di *lookup*) al registry per ottenere un riferimento remoto da usare per le invocazioni remote. Il servizio di naming deve essere lanciato dalla directory dove si trova il file `.class` dello stub, dell'oggetto remoto e dell'interfaccia remota, quindi (nei nostri esempi) dalla directory `bin` del progetto.

ESECUZIONE SERVER:

Ora si può eseguire il server eseguendo con la macchina virtuale la classe appropriata. L'unica cosa particolare da fare è scegliere una politica di sicurezza per la macchina virtuale, operazione che non compiamo per applicazioni locali, di solito. Infatti, la macchina virtuale, per poter eseguire operazioni potenzialmente pericolose (ed accedere la rete lo è) ha bisogno che venga esplicitamente permesso in una policy che viene fornita alla macchina virtuale su linea di comando. Nel nostro caso adottiamo una politica liberale nel quale tutto è permesso:

```
grant { permission java.security.AllPermission; }
```

Fornire su linea di comando alla macchina virtuale il file di policy da seguire:

```
java -Djava.security.policy=pollicyall xxx
```

Se la macchina virtuale non trova il file di policy non protesta in maniera evidente, ma adotta una policy estremamente restrittiva, impedendoci accesso (dai server) ai servizi di naming.

REGISTRAZIONE DEL SERVER SUL SERVIZIO DI NAMING:

Appena lanciato il server, esso deve (di norma) registrarsi sul servizio di naming. Vengono usati metodi del package [java.rmi.Naming](#) che prevedono semplici modalità di registrazione, richiesta e deregistrazione. Per motivi di sicurezza, non è possibile che il server si registri su un servizio di naming che non sia sul suo stesso host, quindi non è possibile avere un servizio di naming "esterno", il che riduce l'efficacia di soluzioni distribuite.

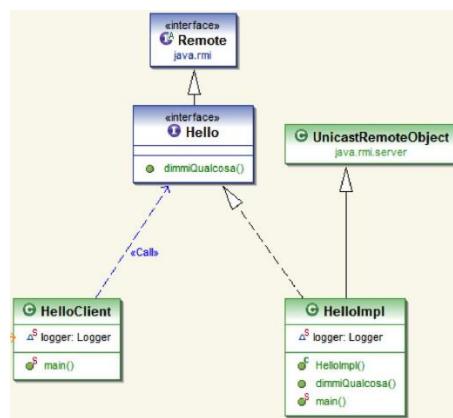
IMPLEMENTAZIONE CLIENT:

Si deve risolvere il problema della localizzazione dell'oggetto remoto, effettuata tramite i servizi di [java.rmi.Naming](#) che permettono di ottenere un riferimento remoto ad un oggetto server. A questo punto, l'invocazione dei metodi remoti procede in maniera tradizionale, con la sola differenza che si deve gestire l'eccezione [RemoteException](#) che viene lanciata da tutti i metodi remoti. L'implementazione dell'invocazione remota risulta essere a carico dello stub.

COMPILAZIONE ED ESECUZIONE DEL CLIENT:

L'unica attenzione che si deve prestare è al fatto che deve essere presente lo stub del server nella directory dove il client viene compilato. La stessa attenzione deve essere compiuta per l'esecuzione.

ESEMPIO HelloWorld in Java RMI:



La nostra applicazione consisterà di un server che offre un metodo remoto che prende come parametro il nome dell'utente che sta invocando il metodo e restituisce una stringa, che verrà stampata a video dal client.

DEFINIZIONE DELL'INTERFACCIA REMOTA:

Implementazione *Hello.java* →

Dobbiamo specificare una interface Java che deriva da `java.rmi.Remote` e indicare il metodo remoto con l'indicazione dei parametri e del tipo di dati restituito. In più, deve lanciare `java.rmi.RemoteException`.

IMPLEMENTAZIONE SERVER:

Implementazione *HelloImpl.java* →

Ora, dobbiamo implementare il server, una convenzione utilizzata per denotare una classe che è l'implementazione di un'interfaccia, è quello di chiamare il programma come l'interfaccia, aggiungendo `Impl` al nome dell'interfaccia.

Notiamo la definizione di un costruttore vuoto, necessario perché (per fare il match con il costruttore della superclasse) deve lanciare la eccezione `RemoteException`.

Poi implementiamo il metodo remoto dell'interface remota col classico comportamento che ci aspettiamo

Nel main istanziamo il Security Manager di RMI che serve a poter caricare classi dinamicamente dalla rete (non serve in questo caso) se esse non sono presenti nei CLASSPHH della macchina virtuale e che implementa la politica data nel file *policyall*.

I blocchi try catch servono per le eccezioni che possono essere lanciate dalla istanziazione dell'oggetto remoto e dall'utilizzo del servizio di Naming attraverso il metodo `rebind()` che registra l'oggetto, con nome `HelloServer` sul registry attivato sullo stesso host dove eseguiremo il programma server.

COMPILAZIONE DEL SERVER: easy.

COMPILAZIONE CON STUB COMPILER rmic:

Dopo aver compilato, si ottiene la classe `HelloImpl.class`, sulla quale possiamo eseguire `rmic` configurato.

SERVIZIO DI NAMING rmiregistry:

A questo punto si può lanciare il registry di RMI.

ESECUZIONE DEL SERVER:

Possiamo lanciare il server ed ottenere dalla console le info che il server è stato creato come oggetto remoto.

REGISTRAZIONE SERVER SU SERVIZIO DI NAMING:

Con `Naming.rebind()`, porta alla registrazione dell'oggetto sul registry, indica che il server è pronto.

IMPLEMENTAZIONE CLIENT:

Tranandosi di invocazioni remote (sia l'utilizzo del servizio di naming che la vera e propria invocazione remota del metodo) questa parte del programma va raccolta in un try catch.

Viene effettuato il `lookup()` del server sul servizio di Naming. Questo viene effettuato passando come parametro una URL che specifica rmi: come protocollo, poi indica il server localhost ed infine indica l'id `HelloServer` con il quale il server si è registrato.

In questa maniera, il riferimento `obj` è un riferimento remoto ad un oggetto server di cui si ignora l'implementazione e si conosce esclusivamente i servizi che vengono offerti tramite la interfaccia remota `Hello`.

A questo punto, viene invocato il metodo remoto, passando come parametro il nome dell'utente ("Pippo") e poi si stampa quello che è stato restituito dalla invocazione remota del metodo `dimmiqualcosa()`.

COMPILAZIONE ED ESECUZIONE DEL CLIENT: easy.

```

public interface Hello extends java.rmi.Remote{
    String dimmiQualcosa(String daChi) throws java.rmi.RemoteException;
}
  
```

```

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;
public class HelloImpl extends UnicastRemoteObject implements Hello{
    private static final long serialVersionUID = -4469091140865645865L;
    static Logger logger= Logger.getLogger("getLogger");
    //Costruttore
    public HelloImpl() throws RemoteException{ }
    //Metodo remoto dimmiQualcosa:
    public String dimmiQualcosa(String daChi) throws RemoteException {
        logger.info("Sto salutando "+daChi);
        return "Ciao!";
    }
    public static void main(String args[]){
        System.setSecurityManager( new RMISecurityManager());
        try{
            logger.info("Creo l'oggetto remoto...");
            HelloImpl obj = new HelloImpl();
            logger.info("...ora effettuo il rebind...");
            Naming.rebind("HelloServer", obj);
            logger.info("... Pronto!");
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
  
```

```

import java.rmi.*;
import java.util.logging.Logger;
public class HelloClient {
    static Logger logger = Logger.getLogger("getLogger");
    public static void main(String args[]) {
        try {
            logger.info("Sto cercando l'oggetto");
            Hello obj=(Hello)Naming.lookup("rmi://localhost/HelloServer");
            logger.info("...Trovato! Invoco metodo...");
            String risultato = obj.dimmiQualcosa("Pippo");
            System.out.println("Ricevuto: " + risultato);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
  
```