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

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

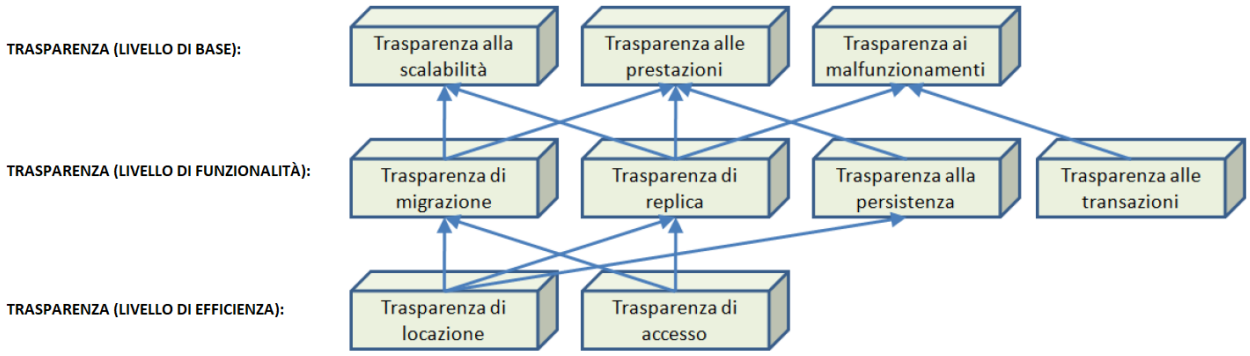
La realizzazione di un sistema distribuito non è agevole e comporta la necessità di considerare vari aspetti generali e globali della architettura, standardizzati in maniera tale che possano servire da specifica per i vari fornitori di piattaforme hardware e software allo scopo di fornire strumenti adeguati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



1. **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.
2. **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.
3. **Trasparenza di migrazione**: nascondere 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).
4. **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.

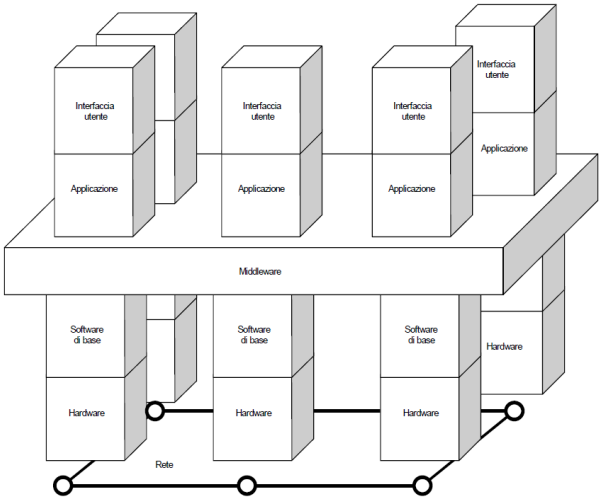
5. **Trasparenza alla persistenza**: scherma l'utente dalle operazioni che compie il sistema per rendere persistente (cioè in memoria secondaria) un oggetto durante una fase di non utilizzo. Infatti, per ottimizzare le prestazioni del sistema, gli oggetti di utilizzo raro non vengono mantenuti attivi (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 programmatori 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 programmatori, 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*.

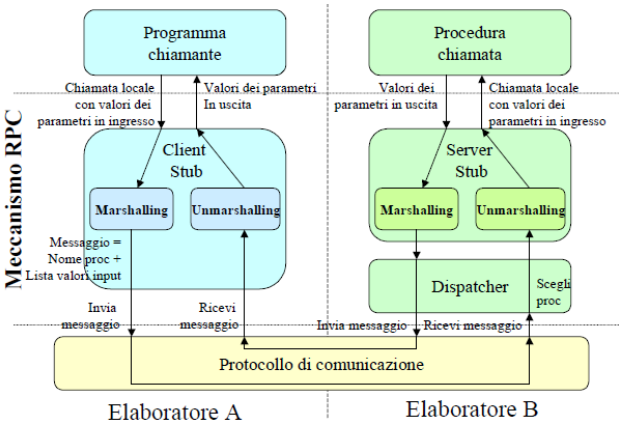
2. **Middleware di distribuzione**: che basa i suoi servizi sul middleware di infrastruttura per automatizzare operazioni comuni per la comunicazione. Tra i compiti più importanti ci sono, ad esempio, quelli di:
   - richiedere un servizio ad un altro nodo potendo inviare parametri (***marshalling***). 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 primini (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 dei 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 e 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 standardizzate in modo che all'altro capo della comunicazione i dati fossero ricostruiti come erano stati trasmessi (operazione di ***Murshalling***);
- 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 rappresentation***);



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.

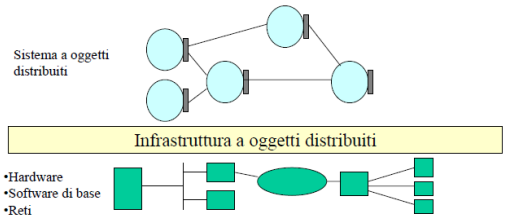
Le operazioni di sincronizzazione, marshalling, data rappresentation e comunicazione tra client e server venivano implementate dai sistemi di RPC attraverso gli **stub**, lato client e lato server che interfacciavano il processo chiamante con il processo che offriva la procedura, fornendo al programmatore 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 della evoluzione e dallo sviluppo dei sistemi distribuiti complessi, che, al crescere della disponibilità delle risorse hardware e di comunicazione, diventano ampi, complessi, eterogenei e difficile da gestire e da manutenere.

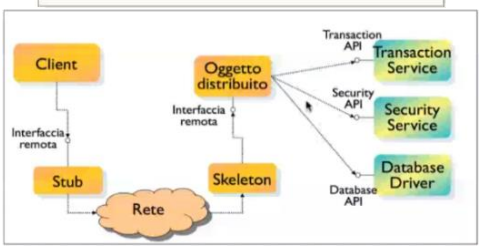
Storicamente, la prima (1991) proposta significativa di ambiente di programmazione basato su oggetti distribuiti e stato **Common Object Request Broker Architecture** (**CORBA**), che è uno standard che permette ad oggetti distribuiti, scritti in diversi linguaggi e quindi eterogenei, di comunicare e collaborare per realizzare una applicazione distribuite. 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) e 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 **intercettazione** 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.

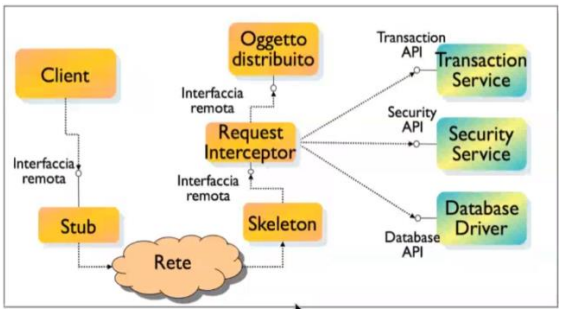
MIDDLEWARE ESPLICITO

```
//...
public void transfer(Account acct1, Account acct2,
      long amount) {
    //1.call Middleware API per security check

    //2.call Middleware API per iniziare una
         transazione

    //3.call Middleware API per caricare i dati

    //4.sottrai amount da account1 e somma
         amount in account2

    //5.call Middleware API per memorizzare i
         dati

    //6.call Middleware API per finire la
         transazione
}
```



MIDDLEWARE IMPLICITO

```
//...
public void transfer(Account acct1, Account
      acct2, long amount) {
    //sottrai amount da account1 e somma amount
         in account2
}
```



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 e quello di poter gestire ii 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?