Appunti di Sistemi Distribuiti

Matteo Gianello

25 gennaio 2014

Indice

1 Introduzione

A partire dalla metà degli anni '80, grazie a due innovazioni tecnologiche si fecero diversi passi avanti nell'uso dei calcolatori. La prima di queste innovazioni fu lo sviluppo di microprocessori potenti; la seconda grande innovazione fu l'invenzione delle reti di computer con l'introduzione delle **LAN** (*Local Area Network*) che consentirono a centinaia di macchine di essere connesse le une alle altre e permisero lo scambio di piccole quantità di informazioni in pochi microsecondi. Il risultato di questa innovazione tecnologica è che oggi mettere insieme una grande quantità di computer tramite una rete ad alta velocità è diventato molto semplice. Questo tipo di sistemi sono solitamente chiamate *reti di computer* o **sistemi distribuiti**.

1.1 Definizione di sistema distribuito

Esistono diverse definizioni di *Sistema distribuito* ma tutte quante sono abbastanza insoddisfacenti. Daremo ora una prima definizione che è sufficiente per i nostri scopi:

Un sistema distribuito è una collezione di computer indipendenti che appare ai propri utenti come un singolo sistema coerente

Da questa definizione possiamo ricavare diversi caratteristiche di un sistema distribuito, la prima è che i sistemi distribuiti sono costituiti da componenti autonomi; la seconda è che gli utenti, siano essi persone o altri programmi, vedono il sistema come un'unica entità. Il che significa che i diversi componenti devono in qualche modo collaborare.
Quello che non viene specificato in questa definizione è il tipo di computer usati per i componenti ne come questi sono interconnessi.
Le caratteristiche più importanti dei sistemi distribuiti sono il fatto che le differenze tra i vari computer e le loro modalità di comunicazione risultano per lo più nascoste agli utenti finali. Inoltre gli utenti possono interagire con un sistema distribuito in modo *consistente* e *uniforme* ovvero indipendentemente da dove e quando avviene l'interazione.
Teoricamente i sistemi distribuiti dovrebbero essere facilmente espandibili e scalabili, inoltre, i sistemi distribuiti sono di norma sempre disponibili anche se alcune sui parti sono momentaneamente fuori uso.
Allo scopo di supportare reti eterogenee e sistemi operativi differenti alle volte si introduce uno strato software tra lo strato di applicazione e i diversi sistemi operativi, questo strato è chiamato **middleware** come mostrato in figura 1.1.

1.2 Obiettivi

La possibilità costruire sistemi distribuiti non implica che tutti i sistemi debbano essere costruiti come sistemi distribuiti. Per far si che sia utile progettare e costruire un sistema distribuito dobbiamo rispettare alcune caratteristiche. un sistema distribuito dovrebbe:

- rendere le risorse facilmente accessibili,
- nascondere il fatto che le risorse sono distribuite sulla rete,
- essere aperto,
- essere scalabile.

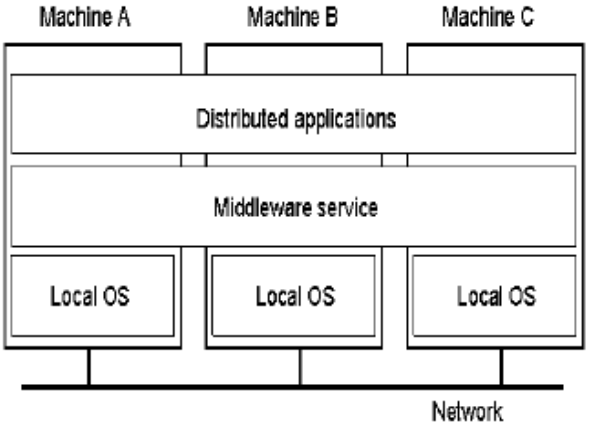

Figura 1: Schema di un middleware

1.2.1 Accessibilità delle risorse

L'obiettivo principale di un sistema distribuito è quello di rendere facile l'accesso alle risorse remote e condividerle in maniera efficiente e controllata.
Ma che cosa intendiamo per risorse? Con il termine *risorse* possiamo indicare qualsiasi cosa, alcuni esempi tipici sono stampanti, computer, dati, file, pagine web o intere reti. Le ragioni che portano a voler condividere le risorse sono molteplici, la prima è sicuramente quella economica, pensiamo ad esempio a ricercatori che condividono un super-computer o ad una stampante condivisa in un ufficio. Inoltre, la connessione di più utenti facilita la collaborazione come avviene nei **groupware** dove gruppi di persone lavorano insieme anche stando in diverse parti del mondo.
Tutto questo incremento di connessione e collaborazione dovrebbe portare però ad una necessaria crescita anche in termini di sicurezza, anche se nella pratica attuale tale incremento nei sistemi di sicurezza non è ancora avvenuto; non è raro trovare sistemi in cui password e altre informazioni sensibili sono inviate come testo in chiaro. Altri problemi legati alla sicurezza sono l'aumento delle *junk mail* o mail di *spam* e l'invio e la raccolta di informazioni riguardanti l'utente per creare un profilo mentre è connesso.

1.2.2 Trasparenza

Uno degli obiettivi principali in un sistema distribuito è quello di nascondere che i processi e le risorse sono distribuiti. Un sistema in grado di presentarsi come un singolo computer è detto **trasparente**.
Possiamo catalogare la trasparenza in diversi tipi in quanto questo concetto può riguardare molti aspetti di un sistema distribuito.
La **trasparenza all'accesso** riguarda le differenze nella rappresentazione dei dati e la modalità di accesso alle risorse da parte degli utenti. Ovvero si desidera nascondere le differenze nelle macchine e trovare un accordo nella rappresentazione dei dati. Un altro importante tipo di trasparenza è la **trasparenza di ubicazione** che si prefigge l'obiettivo di nascondere agli utenti la localizzazione di una risorsa. I *nomi* in questa tipo di trasparenza giocano un ruolo importante in quanto è possibile raggiungere tale trasparenza assegnando ad ogni risorsa un nome logico indipendente dalla sua locazione, un esempio di tale tecnica sono gli *URL*.
Alcuni sistemi distribuiti che consentono lo spostamento delle risorse senza compromettere la possibilità di accesso devono fornire la **trasparenza alla migrazione**. Nel caso in cui le risorse

possono essere spostate *durante* l'utilizzo senza che utenti o applicazioni notino tale spostamento si deve garantire anche la **trasparenza al riposizionamento**.
La **trasparenza alla replica** riguarda la possibilità di fornire una o più copie della stessa risorsa per aumentarne la disponibilità e migliorare le prestazioni, tutto questo nascondendo all'utente il fatto che la risorsa è replicata.
Come già detto l'obiettivo principale dei sistemi distribuiti è la condivisione di risorse, ma questa porta in alcuni casi ad avere una condivisione di tipo *competitivo* ovvero, più utenti vorrebbero accedere alle stesse risorse (es. una tabella di un database) tutto ciò deve essere evitato tramite la **trasparenza alla concorrenza** che deve lasciare la risorsa in uno stato consistente. Questa consistenza può essere ottenuta tramite diverse meccanismi tra cui ad esempio il *locking* nel quale gli utenti ottengono a turno l'accesso esclusivo ad una risorsa.
Per introdurre l'ultimo tipo di concorrenza partiamo da un'altra definizione di sistema distribuito

Ne apprendi l'esistenza quando il crash di un computer di cui non hai mai sentito parlare ti impedisce di portare a termine qualunque lavoro

Questa definizione pone un altro aspetto importante della progettazione di un sistema distribuito, quello della gestione dei guasti; rendere un sistema **trasparente ai guasti** significa far si che un utente non si renda conto che una risorsa smette di funzionare correttamente. La difficoltà più grande è distinguere quando una risorsa è morta da quando è semplicemente molto lenta.
Sebbene in generale si preferisce avere sistemi trasparenti ci sono situazioni in cui nascondere completamente agli utenti la distribuzione del sistema non è una buona idea. Come nel caso si voglia contattare un servizio che sta dall'altra parte del mondo e si voglia una risposta in un tempo inferiore ai 35ms; o quando si vuole che due repliche siano sempre consistenti, nel caso di server in due continenti diversi un aggiornamento potrebbe richiedere alcuni secondi.
Il problema principale che limita però la trasparenza è la trasparenza stessa, infatti, ammettendo che la completa trasparenza di un sistema è *impossibile* è *saggio* cerca di ottenerla a tutti i costi? Rendere la distribuzione esplicita può aiutare gli utenti a capire eventuali comportamenti *anomali* del sistema.

1.2.3 Apertura

un altro obiettivo dei sistemi distribuiti è l'apertura. un sistema distribuito **aperto** è un sistema che offre servizi rispettando delle regole standard che descrivono la sintassi e la semantica dei servizi stessi.
Nei sistemi distribuiti i servizi sono descritti tramite **interfacce** per lo più utilizzando un linguaggio denominato *IDL (interface description language)* che però descrive soltanto la sintassi delle interfacce, ovvero, il nome delle funzioni e i tipi di parametri, i valori di ritorno o le possibili eccezioni sollevate. Per la descrizione di che cosa fa il servizio, invece, si usa solitamente il linguaggio naturale.
Se l'interfaccia è ben specificata un processo che ha bisogno di una determinata interfaccia può comunicare con un altro processo che implementa tale interfaccia; inoltre, consente a due processi distinti di implementare tale interfaccia in due modi completamente diversi, il che porta a due sistemi distribuiti che però operano allo stesso modo.
Una specifica però deve essere *completa* e *neutrale*, completa significa che viene specificato tutto ciò che è necessario per realizzare un'implementazione, ma ottenere la completezza è molto difficile e perciò di solito un programmatore deve aggiungere dettagli specifici dell'implementazione. Per neutrale si intende, invece, che la specifica non deve imporre dettagli sull'implementazione. Completezza e neutralità portano ad altre due importanti caratteristiche che i sistemi distribuiti dovrebbero soddisfare, **interoperabilità** che significa che due implementazioni di vendor diversi

possono collaborare e coesistere basandosi unicamente su di uno standard comune. **Portabilità** indica la possibilità di eseguire un applicazione scritta per un sistema distribuito A su di un sistema distribuito B senza dover apportare modifiche all'applicazione.
Infine un altro obiettivo che i sistemi distribuiti dovrebbero prefissarsi è che l'aggiunta o la sostituzione di componenti dovrebbe risultare facile e non influire sui componenti già presenti; questa caratteristica sta ad indicare che il sistema distribuito è **ampliabile**. Per ottenere la flessibilità in un sistema aperto è fondamentale che esso sia organizzato come un insieme di componenti piccolo e e facilmente sostituibile e adattabile. Ma questo comporta fornire le interfacce non solo dei componenti che si interfacciano direttamente con gli utenti ma anche dei componenti interni.

1.2.4 Scalabilità

La scalabilità sta diventando uno degli aspetti più importanti dei sistemi distribuiti a causa della grande diffusione di internet. Esistono diversi tipi di scalabilità la prima si ha quando un sistema è scalabile rispetto alla sua dimensione il che significa che possiamo aggiungere utenti e risorse. Un sistema può essere scalabile geograficamente quando utenti e risorse sono situati in luoghi molto lontani. Ed, infine, un sistema può essere scalabile anche dal punto di vista dell'amministrazione ovvero quando comprende molte infrastrutture indipendenti rimane comunque facile da gestire.
La scalabilità richiede di affrontare molti problemi. Prendiamo ad esempio la scalabilità rispetto alla dimensione, alcuni servizi in un sistema distribuito possono essere forniti da un unico server questo fa si che aggiungendo utenti quel particolare server diventa un collo di bottiglia per l'intero sistema. A volte l'uso di un solo server è inevitabile come nel caso della gestione di dati sensibili. Per quanto riguarda la scalabilità a livello geografico anch'essa comporta innumerevoli problemi, infatti, la maggior parte dei sistemi distribuiti che lavorano su LAN si basano sulla comunicazione **sincrona** nella quale un *client* richiede una risorsa e resta in attesa che tale risorsa sia disponibile. Questo meccanismo non è applicabile per sistemi globali nei quali la comunicazione tra due computer può richiedere anche qualche millisecondo. In oltre si deve tener conto che la comunicazione su WAN(*wide area network*) è inaffidabile e praticamente sempre punto a punto (*point-to-point*). Al contrario le reti locali più affidabili permettono anche il *broadcasting* ovvero l'invio simultaneo a tutte le macchine delle rete dello stesso messaggio.
Dopo aver visto i problemi di scalabilità ci chiediamo come risolvere tali problemi nei sistemi distribuiti. I problemi di scaling nei sistemi distribuiti si presentano come problemi nelle prestazioni dovuti alle limitate capacità dei server. Ad oggi esistono soltanto tre tecniche di *scaling*: nascondere le latenze, la distribuzione e la replica.
Nascondere le latenze permette di ottenere la scalabilità geografica; l'idea di base è quella di limitare il più possibile i tempi di attesa delle risposte dai servizi remoti. Alcune possibili soluzioni sono anticipare il più possibili la richiesta al server remoto e durante l'attesa della risposta svolgere qualche altra operazione; questo tipo di comunicazione è detta **comunicazione asincrona**. Quando arriva una risposta l'applicazione si interrompe e viene richiamato un gestore (*handler*) speciale per completare la richiesta sollevata. Inoltre la comunicazione asincrona è spesso utilizzata nei sistemi *batch* e nelle applicazioni *parallele*.
Un'altra soluzione è quella di eseguire un *thread* per la richiesta il quale, anche se si blocca, permette agli altri thread di proseguire.
Esiste una classe di applicazioni, però, che non può utilizzare la comunicazione asincrona, questo tipo di applicazioni sono le applicazioni *interattive* nelle quali il client dopo aver effettuato una richiesta ad un server remoto non ha niente di meglio da fare che aspettare la risposta. In questo caso l'unica cosa da fare è limitare il tempo di attesa e per fare ciò solitamente si sposta il carico

di lavoro computazionale che solitamente è svolto dal server sul client. Come ad esempio nel caso di compilazione di *form* per un accesso alla base di dati. Si può inviare al server ogni campo della form per poi aspettare dal server la conferma della correttezza dei dati, oppure, più efficiente è far controllare direttamente al client la correttezza dei dati ed inviare al server l'intera form completa.
Un'altra soluzione ai problemi di scalabilità è la **distribuzione** che comporta prendere un componente spezzarlo in parti più piccole e distribuire tali parti nel sistema. Un esempio molto noto di questo tipo di tecnica è il DNS (*domain name system.* Lo spazio dei nomi è organizzato in un albero dei *domini* divisi in zone non sovrapposte. I nomi di ogni zona sono gestiti da un unico server.
L'ultima tecnica di scalabilità è la **replicazione** che consiste nel duplicare quelle risorse che sono maggiormente richieste in modo da evitare colli di bottiglia e bilanciare il carico sulle risorse.

1.2.5 Tranelli

I sistemi distribuiti si differenziano dal software tradizionale in quanto sono i componenti sono sparsi per la rete. Il fatto di non tenere conto di questa dispersione in fase progettuale rende i sistemi inutilmente complessi. Questi errori sono dovuti a delle ipotesi (false) che ognuno fa quando progetta un'applicazione distribuita:

1. La rete è affidabile
2. La rete è sicura
3. La rete è omogenea
4. La topologia non cambia
5. La latenza è zero
6. L'ampiezza di banda è infinita
7. Il costo di trasporto è zero
8. C'è un solo amministratore

1.3 Tipi di sistemi distribuiti

Esistono diverse categorie di sistemi distribuiti

1.3.1 Sistemi di calcolo distribuiti

Una classe di sistemi distribuiti è quella utilizzata per calcoli ad alte prestazioni; questa categoria può essere divisa in due sottogruppi. Nei **sistemi di calcolo a cluster** l'hardware è composto da una serie di workstation connessi ad una rete locale ad alta velocità e in cui ogni nodo ha lo stesso sistema operativo. Nei **sistemi con tecnologia grid** invece, si intendono sistemi distribuiti costruiti come un gruppo di computer risiedenti in domini di amministrazione diversi con hardware e software differenti.

Sistemi di calcolo a cluster I sistemi di calcolo a *cluster* divennero popolari quando il rapporto prezzo/prestazioni delle *workstation* divenne vantaggioso. In quasi tutti i casi i sistemi di calcolo a cluster sono utilizzati per per la programmazione parallela.
Un esempio di sistema a cluster molto diffuso è il **Beowolf** un sistema basato su linux in cui l'accesso al sistema avviene tramite un singolo nodo principale, che gestisce l'allocazione dei programmi sui nodi. In realtà il nodo master esegue il *middleware* necessario per l'esecuzione dei programmi e la gestione del cluster. Una parte fondamentale di questo middleware sono le librerie con il quale è sviluppato che forniscono solo funzionalità di comunicazione basate sui messaggi e non sono in grado di gestire errori, sicurezza ecc.
Un altro sistema cluster molto diffuso è il **MOSIX** che fa sembrare il cluster un singolo sistema offrendo così ai programmi la trasparenza di distribuzione.

Grid computing Al contrario dei sistemi di calcolo a cluster in cui i componenti hardware sono omogenei nei sistemi *Grid Computing* vi è un alto livello di eterogeneità sia hardware sia software.
Solitamente nella tecnologia grid le risorse di diverse aziende vengono unite per consentire la collaborazione che viene anche detta **organizzazione virtuale**.

1.3.2 Sistemi informativi basati sulle imprese

Un'altra classe di sistemi distribuiti si trova in strutture aziendali che si sono confrontate con una gran abbondanza di applicazione in rete ma per le quali l'interoperabilità non è stata facile. In alcuni casi l'integrazione riguardava solamente un server che veniva contattato da diversi client i quali confezionavano una richiesta e la inviavano a tale server. Un'integrazione più approfondita avrebbe permesso ai client di confezionare una richiesta diretta a molteplici server che avrebbero eseguito un **applicazione distribuita**.

Sistemi transazionali Sono sistemi incentrati su applicazioni relative a basi di dati. In pratica le operazioni sulle basi di dati sono portate a termine sotto forma di **transazioni**. In un sistema distribuito transazionale abbiamo la particolarità che all'interno di una transazione possiamo avere delle chiamate a procedura remote ottenendo così un sistema **RPC transazionale**.

1.3.3 Sistemi distribuiti pervasivi

I sistemi distribuiti visti fin qui hanno la particolarità di essere caratterizzati dalla stabilità: i nodi sono fissi ed hanno a disposizione una connessione di alta qualità. Con l'introduzione di dispositivi mobili ed embedded però abbiamo a che fare con sistemi distribuiti in cui la caratteristica principale è l'instabilità; tali sistemi sono detti **sistemi distribuiti pervasivi**.
I sistemi pervasivi hanno delle caratteristiche particolari, intanto non esiste un amministratore umano ma dopo una prima configurazione da parte del proprietario tali sistemi devono adattarsi al meglio all'ambiente circostante. Inoltre tali sistemi devono avere tre requisiti fondamentali:

- accettare cambi di contesto
- incoraggiare la composizione *ad-hoc*
- riconoscere la condivisione di default

Ovvero un dispositivo deve essere a conoscenza che il suo ambiente è in costante cambiamento e che i dispositivi che compongono il sistema saranno utilizzati in modo diverso da utenti diversi.

In questo tipo di sistemi non vi è alcun tipo di trasparenza, bensì la distribuzione dei dati dei processi e del controllo è intrinseca nei sistemi ed è quindi più efficiente esporla che nasconderla. Alcuni esempi di sistemi pervasivi sono:

- Sistemi domestici
- Sistemi elettronici per l'assistenza sanitaria
- Reti di sensori

2 Architetture

I sistemi distribuiti sono spesso definibili come pezzi di software sparsi su molte macchine. Al fine di dominare la loro complessità è necessario che questi sistemi siano organizzati. Un modo semplice per distinguere l'organizzazione di un sistema distribuito, è quello di distinguere l'organizzazione logica dei componenti software e la relativa realizzazione fisica.
Per *architetture software* intendiamo l'organizzazione e l'interazione dei vari componenti software; mentre l'effettiva realizzazione di un sistema distribuito richiede che i componenti software siano instanziati su macchine reali, l'architettura risultante viene detta *architettura di sistema.*Analizzeremo per prime le architetture centralizzate in cui un server implementa la maggior parte delle funzioni mentre client remoti accedono al server tramite semplici mezzi di comunicazione.

2.1 Stili architetturali

Iniziamo l'analisi dalle diverse tipologie di architetture software in quanto progettazione e adozione di una adeguata architettura sono fondamentali per la riuscita e la manutenibilità del sistema.
Introduciamo ora la nozione di *stile architetturale* che esprime in termini di componenti mezzi di comunicazione e messaggi scambiati. Un *componente* è un'unità modulare con interfacce ben definite e sostituibile nel suo ambiente.
La comunicazione tra i diversi componenti avviene tramite quello che è definito *connettore* ovvero un sistema che implementa le chiamate a procedure remote piuttosto che il passaggio di messaggi o lo streaming dei dati.
Usando componenti e connettori possiamo ottenere diverse configurazioni che a loro volta sono classificati in diversi *stili architetturali.* I più importanti stili architetturali ad oggi identificati sono:

- architettura a livelli (*layer*)
- architetture basate sugli oggetti
- architetture centrate sui dati
- architetture basate sugli eventi

Per quanto riguarda lo stile a livelli è quello più semplice nel quale i componenti sono organizzati a strati in cui un componente del livello L_i può chiamare un componente del livello L_{i-1} ma non può contattare i componenti dello stesso livello. Questo modello è uno dei più utilizzati nelle applicazioni di rete, le richieste scendono lungo la catena gerarchica mentre le risposte risalgono.
Le *architetture basate sugli oggetti* hanno un organizzazione meno rigida, ogni oggetto corrisponde ad un componente e tutti gli oggetti sono connessi tramite *chiamate a procedura remota*; questo tipo di architettura corrisponde esattamente al caso client-server ed insieme a quella a livelli costituiscono il 90% delle architetture dei sistemi distribuiti presenti oggi.
Le *architetture basate sui dati* si sviluppano attorno all'idea che i processi comunicano attraverso un *repository* comune.
Nelle *architetture basate sugli eventi* i processi comunicano essenzialmente attraverso la propagazione degli eventi, i più noti tipi di sistemi distribuiti che utilizzano la propagazione degli eventi sono i sistemi *publish/subscribe* nei quali alcuni processi pubblicano degli eventi ed è il middleware ad assicurarsi che questi eventi siano ricevuti soltanto da qui processi che si sono

iscritti a quel determinato evento.
Nel caso in cui si combinino le architetture basate sugli eventi con quelle basate sui dati si ottiene una architettura chiamata *spazio di dati condivisi* che permette ai processi di essere disaccoppiati anche nel tempo in quanto non è necessario che i processi siano attivi nell'istante in cui avviene la comunicazione.

2.2 Architetture di sistema

Abbiamo visto fino ad ora alcune scelte architetturali, vediamo ora come sono effettivamente organizzati la maggior parte dei sistemi distribuiti considerando dove sono posizionati i componenti software. La scelta di quali componenti usare e di come posizionarli porta alla realizzazione della cosiddetta *architettura di sistema.*

2.3 Architetture centralizzate

La prima architettura che analizzeremo è quella *centralizzata*, in quanto pensare ad un sistema in termini di *client* che richiedono dei servizi a dei *server* facilita la comprensione e la gestione dei sistemi distribuiti.
Nel modello client-server i processi sono suddivisi in due gruppi; un **server** è un processo che implementa uno specifico servizio. Un **client** è invece un processo che richiede un servizio ad un serve inviandogli una richiesta e quindi attendendo una risposta.
La comunicazione tra client e server può essere implementata per mezzo di un semplice protocollo senza connessione quando la rete sottostante è molto affidabile. In questo caso quando un client richiede un servizio invia un messaggio al server indicando il servizio richiesto e i dati di input. Quest'ultimo all'arrivo della richiesta la processa e confeziona i risultati in un altro messaggio.
L'utilizzo di un protocollo senza connessione ha il vantaggio di essere efficiente fino a quando non abbiamo perdita di pacchetti. Si potrebbe pensare di impostare il client affinché rinvii il messaggio quando non riceve alcuna risposta, ma non si può stabilire se è stato perso il messaggio o la risposta e quindi il server ha compiuto oppure no l'operazione richiesta. In alcuni casi le richieste sono *idem-potenti* ovvero possono essere ripetute senza danni.
Per risolvere il problema della perdita di messaggi, molte architetture client-server utilizzano dei protocolli affidabili orientati alla connessione.

Stratificazioni delle applicazioni Il problema principale dell'architettura client server è che non vi è una netta distinzione tra quali sono le funzionalità del client e quelle del server; è stata così introdotta una un nuovo tipo di suddivisione delle diverse funzionalità in base che esse siano più vicine all'utente o ai dati. I tre livelli identificati sono:

- il livello dell'interfaccia utente
- il livello applicativo
- il livello dei dati

Il livello dell'interfaccia utente contiene tutto ciò che è necessario per interfacciarsi con l'utente come la gestione della grafica. Il livello applicativo di solito contiene le applicazioni. Il livello dei dati gestisce tutto ciò che concerne i dati da gestire.
Esistono diversi modi in cui questi tre livelli possono essere istanziati sull'hardware come possiamo vedere in Figura 2. La configurazione più utilizzata è quella nel quale il client implementa solo il livello dell'interfaccia utente (*thin client*) ma esistono altri sistemi una parte dal livello

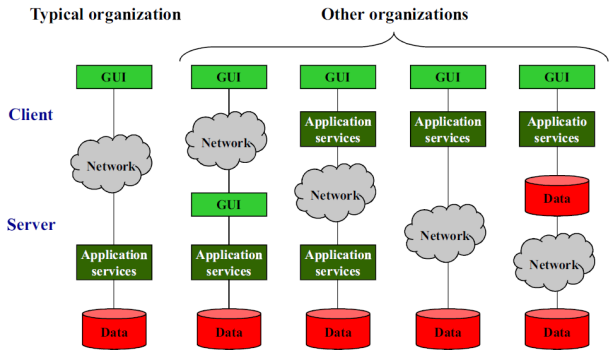

Figura 2: Esempio di suddivisione dei layer

applicativo si trova implementata nel client, in questo caso si parla di *fat client*. Per quanto riguarda il livello dei dati molte volte è gestito da meccanismi che rendono i dati **persistenti** anche quando non vi sono altre applicazioni in esecuzione. In alcuni casi molto semplici il livello dei dati consiste in un *filesystem* ma nella maggioranza dei casi si tratta di una *base di dati*

Architetture multi livello La suddivisione delle applicazioni in tre livelli logici suggerisce anche una distribuzione fisica delle applicazioni client server su molte macchine, la distribuzione più semplice è quella su due macchine:

1. una macchina client che contiene solo i programmi dell'interfaccia utente
2. una macchina server contenete il resto dei programmi e i dati

Si possono suddividere le applicazioni anche in altri modi come visto in Figura 2.
La tendenza degli ultimi anni è quella di suddividere i diversi livelli su più macchine in modo da creare una architetture multi livello. Un esempio pratico ad esempio è quello mostrato in Figura 3 nella quale si mostra un architettura a 3 livelli.

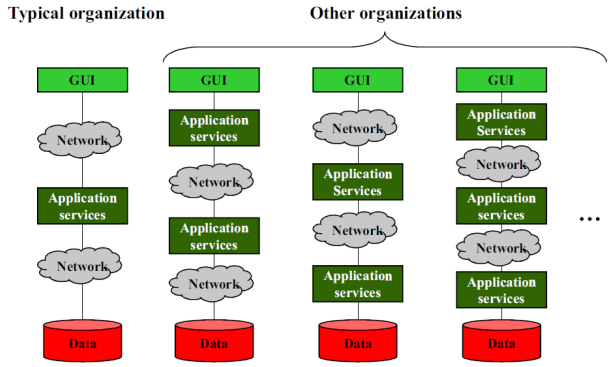

Figura 3: Esempio di suddivisione dei layer su 3 livelli

2.3.1 Architetture decentralizzate

La suddivisione delle architetture client server in livelli denota un tipo di distribuzione detta *verticale* in quanto i componenti sono divisi su più macchine seguendo un criterio *logico*.

Questo tipo di distribuzione è utile con le architetture client-server ma nel caso in cui la distribuzione che conta è quella dei client e dei server e non delle funzioni allora si parla di *frammentazione orizzontale*. Un esempio molto conosciuto di architettura con distribuzione orizzontale sono i **sistemi peer-to-peer**.
I processi che costituiscono un sistema peer-to-peer sono tutti uguali, di conseguenza l'interazione tra processi è quasi del tutto simmetrica e i processi agiscono sia da client che da server e per questo sono anche detti **servent**.
Dato questo tipo di comportamento il problema principale delle reti *p2p* è quello di organizzare i processi in una rete *overlay* ovvero una rete nella quale i processi costituiscono i nodi e i collegamenti rappresentano i canali di comunicazione. Con questa struttura un processo non può comunicare direttamente con un altro processo arbitrario ma deve seguire i canali di comunicazione disponibili.
Esistono due tipi principali di reti *overlay* quelle strutturate e quelle non strutturate.

Architetture peer-to-peer strutturate In una rete p2p strutturata la rete overlay è costruita usando una procedura deterministica, quella più comune è l'uso di una **hash tabel distribuita** (DHT). In questo tipo di struttura ai dati viene assegnato un identificatore univoco in uno spazio molto grande (129:160 bit). Anche ai nodi del sistema viene assegnato un identificatore nello stesso spazio dei dati. Il punto cruciale di questa architettura è quello di creare uno schema efficiente e deterministico che associ univocamente la chiave di un dato con l'identificativo di un nodo basandosi su un'opportuna metrica di distanza. Inoltre, è importante che quando si effettua la ricerca di un dato sia restituito l'indirizzo del nodo al quale questo dato è associato, e ciò si ottiene *instradando* la richiesta al nodo associato.
Ad esempio nel sistema *Chord* i nodi sono organizzati logicamente ad anello, ed i dati sono organizzati assegnando i dati con identificativo k al nodo con più piccolo identificativo $id > k$; questo nodo è chiamato *successore* della chiave k ed è identificato come $succ(k)$ come mostrato in Figura 4.
La parte più importante della gestione di una rete overlay strutturata è la **gestione dell'appartenenza** da parte dei nodi. Quando un nodo vuole unirsi alla rete genera un id casuale ed effettua una ricerca di id a questo punto il sistema restituirà $succ(id)$. Alla fine per inserirsi il nuovo nodo contatterà il successore individuato dalla ricerca e il suo predecessore e si inserirà nell'anello. Inoltre, l'inserimento causa la migrazione dei dati associati ad id da $succ(id)$. L'uscita è molto semplice il nodo informa della sua dipartita il $succ(id)$ e il suo predecessore e trasferisce i dati a $succ(id)$.

Architetture peer-to-peer non strutturate I sistemi peer-to-peer non strutturati si basano su algoritmi casuali per costruire la rete *overlay*. L'idea è quella che ogni nodo mantenga una lista dei suoi vicini. Inoltre, anche i dati sono posizionati sui nodi in modo casuale, di conseguenza l'unico modo di effettuare una ricerca è inoltrare la richiesta a tutta la rete.
L'obiettivo principale di un sistema p2p non strutturato è la creazione di un **grafo disordinato**. Per fare ciò ogni nodo mantiene una lista di c vicini scelti a caso dall'insieme dei nodi *vivi*; questa vista è detta **vista parziale**. Ora supponiamo che un nodo voglia aggiungersi alla rete, esso contatta in altro nodo arbitrario dalla lista dei punti di accesso; questo punto di accesso è un normale nodo della rete che però ha un alta probabilità di essere disponibile. I meccanismi che creano la rete overlay sono detti *push* e *pull* e permettono lo scambio delle informazioni per la costruzione della rete tra i nodi. Questi meccanismi usati singolarmente possono portare alla creazione di reti overlay disconnesse, per questo si usano entrambe le tecniche.

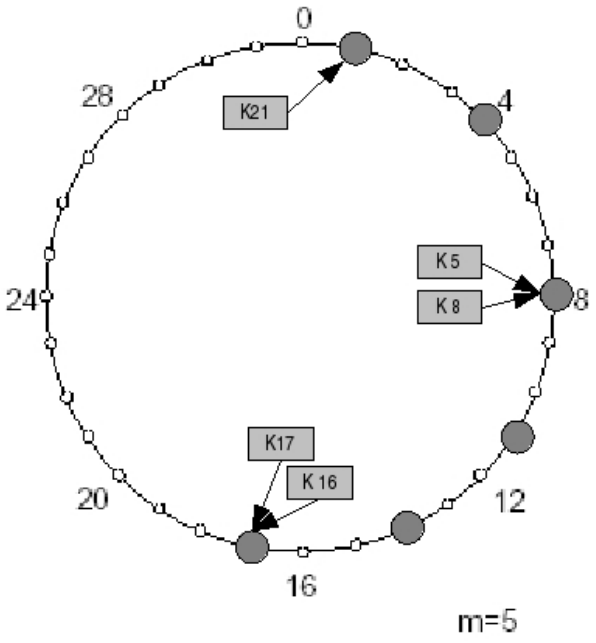

Figura 4: Esempio di sistema Chord

L'uscita dalla rete è molto semplice, infatti, visto che i nodi si scambiano periodicamente le viste parziali basta solo che il nodo lasci la rete, gli altri nodi con il passare del tempo si accorgono dell'assenza del nodo che ha lasciato la rete e lo elimina dalla sua vista parziale.

Gestione della topologia di una rete overlay Anche se i sistemi p2p strutturati e non strutturati sembrano costruire due classi completamente diverse, in realtà selezionando attentamente gli elementi scambiati nelle viste parziali è possibile costruire reti overlay con tipologie specifiche. Un esempio molto interessante sono quelle funzioni che cercano di cogliere la **vicinanza semantica** dei dati che creano reti **overlay semantiche** che permettono ricerche efficienti nei sistemi p2p non strutturati.

Superpeer Specialmente nelle reti non strutturate al crescere delle dimensioni potrebbe diventare problematico localizzare i dati. Questo è dovuto al fatto non esiste un modo deterministico per instradare una richiesta di ricerca.
Per evitare questo problema molte reti p2p hanno proposto l'utilizzo di nodi speciali che mantengono un indice dei dati. Questo tipo di nodi sono detti **superpeer**. I superpeer sono organizzati tra loro tramite una rete p2p; si forma così una struttura ad albero. L'accesso ad un normale peer avviene attraverso l'accesso al superpeer.

2.3.2 Architetture ibride

Fino ad ora abbiamo visto le architetture centralizzate client-server e alcune architetture peer-to-peer ora vediamo come queste due tipologie di architetture possono combinarsi per dare vita ad altre architetture dette *ibride*

Sistemi edge-server I sistemi *edge-server* sono una classe molto diffusa di architetture ibride. In questa classe i server sono posti ai bordi della rete Internet ovvero tra la rete Internet e quelle

aziendali come nel caso degli **Internet service provider**. I client si connettono alla rete internet tramite un edge-server il quale fornisce i contenuti dopo aver applicato dei filtri e funzioni di transcodifica.

Sistemi distribuiti collaborativi Le strutture ibride sono largamente utilizzate nei sistemi distribuiti collaborativi dove sono necessarie velocità nell'entrare nel sistema e per questo viene utilizzato uno schema di tipo client-server. Dopo di che si usa uno schema completamente decentralizzato.
Un sistema concreto che utilizza questo meccanismo è il sistema *BitTorrent*

2.4 Architetture e middleware a confronto

Considerando le questioni architetturali viste fino ad ora ci chiediamo quale ruolo giochi il middleware visto nei capitoli precedenti.
Il middleware in realtà costituisce uno strato tra le applicazioni e le piattaforme distribuite ed il suo obiettivo è quello di fornire un certo grado di trasparenza alla distribuzione. Ciò che accade in realtà è che i middleware seguono uno specifico stile architetturale (ad oggetti come *CORBA* o ad eventi come *TIB/Rendezvous*). Avere un middleware basato su di un certo stile architetturale rende più semplice la progettazione e la realizzazione delle applicazioni.
Lo svantaggio più grande è però il fatto che un middleware può non essere ottimale per una determinata applicazione ciò porta ad avere o middleware molto grandi o a diverse versioni per una specifica classe di applicazioni.

2.4.1 Interceptor

Concettualmente un *interceptor* non è altro che un costrutto software che interrompe il normale flusso di controllo e consente ad altro codice di essere eseguito. Rendere però un interceptor generico è molto arduo e a volte averne uno con funzionalità limitate migliorerà sia la gestione del software che che il sistema distribuito nel suo complesso.
Il concetto è che un oggetto A può richiamare un metodo appartenete all'oggetto B anche se quest'ultimo risiede su una macchina diversa da A. I passi per eseguire questa chiamata remota sono:

1. All'oggetto A viene fornita un interfaccia locale esattamente uguale a quella fornita dall'oggetto B; a questo punto A richiama il metodo disponibile nell'interfaccia

2. La chiamata di A è trasformata in un'invocazione a un oggetto generico disponibile tramite un'interfaccia generale messa a disposizione dal middleware.

3. L'invocazione a questo oggetto generico viene trasformata in un messaggio inviato attraverso l'interfaccia di rete.

3 Modelling

Nel precedente capitolo abbiamo visto il perché di alcune scelte architetturali nei sistemi distribuiti; in questo capitolo invece analizzeremo alcuni dei modelli esistenti e di come questi siano realmente utilizzati.

3.1 Architettura service oriented

Partendo dal concetto di architettura client-server si può pensare di costruire una architettura costruita interamente attorno al concetto di servizio (*service provider, service consumer, service brokers*). Un servizio rappresenta un insieme di funzionalità vagamente legate tra loro che sono messe a disposizione di una unita chiamata **fornitore di servizi** (*service provider*). Il **brokers** mantiene la descrizione dei servizi disponibili che possono essere cercati dai **consumer** che poi li richiamano quando ne hanno bisogno.
Con il termine *orchestration* si indicano l'insieme di invocazioni di determinati servizi in un flusso di lavoro per soddisfare un determinato obiettivo.

3.2 REST style

Lo stile REST (*REpresentational State Transfert*) è allo stesso tempo un buon modo di descrivere il web e un insieme di principi che definiscono come gli standard del Web dovrebbero essere utilizzati.
Gli obiettivi principali del REST includono :

- La scalabilità delle interazioni tra componenti
- Generalità delle interfacce
- Sviluppo indipendente dei componenti
- Componenti intermedi per ridurre le latenze aumentare la sicurezza ed incapsulare i componenti legacy.

Le principali caratteristiche del sistema REST sono che anch'esso è basato su un architettura client-server; le interazioni sono di tipo *stateless*, gli stati devono essere trasferiti di volta in volta dal client al server;. I dati che giungono come risposta ad una richiesta devono essere etichettati come cacheable oppure non-cacheable; ogni componente non può comunicare con se non con i layer più vicini. I client devono supportare il *code-on-demand*; ed infine, i componenti devono esporre un interfaccia uniforme.
Per quanto riguarda l'ultimo punto le interfacce dei componenti devono soddisfare quattro vincoli principali:

- Tutte le risorse devono essere identificate da un id (solitamente un *URI*) ed ogni risorsa con un id è una risorsa valida
- Manipolazione delle risorse tramite la loro rappresentazione, i diversi componenti comunicano tramite il trasferimento di rappresentazioni delle risorse in formati standard (XML) selezionati dinamicamente in base alle capacità o alle informazioni desiderate.
- Messaggi auto-descrittivi, i messaggi contengono al loro interno dei metadati che ne indicano il tipo di richiesta oppure il significato delle risposte. Questa tecnica è utilizzabile per parametrizzare le richieste

- Link ipermediali, i client cambiano il loro stato attraverso le richieste che avvengono tramite dei link ipermediali.

3.2.1 Peer-to-Peer

Come visto nel capitolo 2 nei sistemi peer-to-peer non esistono dei ruoli definiti ma tutti i componendi giocano lo stesso ruolo. Come già detto i sistemi p2p a differenza di quelli client-server permettono di scalare in modo migliore

3.3 Object oriented

Nel caso *Object Oriented* i componenti distribuiti incapsulano i dati e permettono l'accesso e la modifica solo tramite un interfaccia messa a disposizione da ogni componente. I diversi componenti interagiscono tramite RPC. Questo tipo di sistema si basa sul modello p2p ma il più delle volte è utilizzato per implementare meccanismi client-server

3.4 Data centered

Nel caso incentrato sui dai i componenti comunicano, solitamente in modo passivo, con un repository centrale nel quale i dati possono essere recuperati o aggiunti. La comunicazione avviene tramite chiamata a procedura remote e l'accesso ai repository è solitamente sincronizzato.

3.4.1 Il modello Linda

Il modello Linda è un modello introdotto negli anni '80 ed incentrato sulla condivisione dei dati, tale modello è usato principalmente nei sistemi di calcolo parallelo.
In questo modello la comunicazione è persistente e basata sul contesto si ottiene così un alto grado di disaccoppiamento. Le caratteristiche principali di Linda sono:

- I dati sono memorizzati in sequenza in base al tipo di campi (*tuple*)
- Le tuple sono memorizzate in uno spazio persistente e globale (*spazio delle tuple*)
- Operazioni standard come **out**(t) che scrive le tuple nello spazio delle tuple **rd**(p) che legge le tuple che coincidono con il pattern p

Il problema principale di questo sistema è che il modello a spazio di tuple non è facilmente scalabile soprattutto quando aumenta l'area del dominio. Il sistema è proattivo, ovvero esso risponde solo a delle richieste.

3.5 Event-Based

Nei sistemi basati sugli eventi i componenti collaborano per scambiarsi delle informazioni al verificarsi di alcuni *eventi*. In particolare esistono dei componenti che *pubblicano* le informazioni relative all'evento e altri componenti che si *sottoscrivono* a tale eventi.
Il sistema è di tipo asincrono basato su messaggi di tipo multicast ed anonimo in quanto non è importante sapere chi pubblica.

3.6 Mobile Code

Questo modello è diverso dai precedenti, è basato sull'abilità di reallocare i componenti dei sistemi distribuiti a run-time. Esistono diversi tipi di mobile code:

- *Strong mobility:* è la possibilità del sistema di migrare sia il codice sia lo stato di esecuzione.
- *Weak mobility:* è la possibilità di muovere il codice attraverso differenti ambienti di esecuzione

4 Processi

In questo capitolo vedremo come i processi giochino un ruolo fondamentale nei sistemi distribuiti. Il concetto di processo proviene dall'ambito dei sistemi operativi ed è definito come un programma in esecuzione.
Per organizzare efficacemente un sistema client-server è spesso necessario utilizzare tecniche di *multithreading* in quanto questa tecnica permette ai client e ai server di essere costruiti in modo tale che la comunicazione e l'elaborazione locale siano sovrapposti ottenendo un alto livello di prestazione.

4.1 Thread

Anche se i processi costituiscono la base di tutti i sistemi, la loro granularità non è sufficiente a soddisfare i bisogni dei sistemi distribuiti. una gestione più fine, sotto forma di **thread**, rende più facile la costruzione di applicazioni distribuite e ottenere prestazione migliori.

4.1.1 Introduzione ai threads

Prima di capire che cos'è un thread e che ruolo esso gioca nella costruzione di applicazioni distribuite è utile capire che cos'è in realtà un processo e che ruolo ha con i thread.
Per eseguire un programma, un sistema operativo crea un certo numero di processi virtuali. Per tener traccia di questi processi il sistema operativo mantiene aggiornata una **tabella dei processi** contenente elementi che vanno dalla memorizzazione dei registri della CPU, alla mappe della memoria, alla lista dei file aperti alle informazioni sugli *account* e così via.
Il sistema operativo fa si che processi indipendenti non possano in alcun modo influire sulla correttezza degli altri processi, ovvero, è reso trasparente il fatto che più processi possano condividere concorrentemente la stessa CPU e le altre risorse hardware. Questa concorrenza però è ottenuta ad un prezzo abbastanza alto; ogni volta che viene creato un processo il sistema operativo deve creare uno spazio degli indirizzi completamente indipendente. Allocare memoria può voler dire inizializzare segmenti di memoria azzerando segmenti dati, copiare il programma in un segmento di testo e preparando uno *stack* per i dati temporanei. Altrettanto costoso è il passaggio tra un processo ed un altro a livello di CPU, in quanto oltre a salvare il contesto è necessario cambiare i registri ed invalidare la cache.
Come un processo un *thread* esegue il suo pezzo di codice indipendentemente dagli altri threads. A differenza dei processi nei threads non si cerca di ottenere un alto grado di trasparenza, in quanto il fatto di cercare di mantenere la trasparenza fa degradare le prestazione; di conseguenza un sistema basato sui thread gestisce l'insieme minimo delle informazioni per gestire la CPU. Infatti, il **contesto di un thread** è spesso costituito solamente dal contesto della CPU e dalle informazioni per gestire il thread stesso come ad esempio lo stato dovuto al blocco di una variabile *mutex*. È quindi compito degli sviluppatori proteggere l'accesso ai dati tra i vari threads di un singolo processo.

Utilizzo dei thread nei sistemi non distribuiti Il vantaggio principale dell'utilizzo dei thread nei sistemi non distribuiti deriva dal fatto che in un processo a singolo thread quando viene effettuata una chiamata di sistema bloccante l'intero processo viene messo in pausa. Come nel caso di un foglio elettronico dove più celle sono collegate tra loro; in questo caso quando l'utente modifica il valore di una cella anche altre celle vengono rielaborate, ma tale rielaborazione è impensabile in un sistema a singolo thread in quanto il processo resterebbe bloccato in attesa di input e non calcolerebbe il valore delle altre celle.

Un altro vantaggio del multithreading è la possibilità di sfruttare il parallelismo quando si esegue il programma su sistemi multiprocessore. Il multithreading è usato anche nelle grandi applicazioni, le quali solitamente sono sviluppate come un insieme di processi cooperanti; tale cooperazione è realizzata tramite meccanismi di comunicazione tra processi (*IPC, interprocess comunication*), ma questi meccanismi solitamente richiedono molti cambi di contesto che ne rallentano notevolmente le prestazioni. Invece di usare i processi un'applicazione può essere costruita mediante l'utilizzo di threads e la comunicazione tra questi avviene mediante l'uso dei dati condivisi, ed il passaggio da un thread all'altro può essere eseguito a livello utente.

Implementazione dei thread I thread sono spesso forniti sotto forma di pacchetto contente le operazioni di creazione e distruzione dei threads sia le operazioni per la loro sincronizzazione come *mutex* e *condition*. Gli approcci per implementare un pacchetto di thread sono due. Il primo è costruire una libreria che viene eseguita completamente a livello utente, il secondo è lasciare che il kernel sia conscio dei threads e si occupi del loro scheduling.
Usare una libreria utente ha notevoli vantaggi, prima di tutto la creazione e la distruzione dei threads a livello utente è molto meno costosa in quanto il costo è dovuto solo all'allocazione della memoria per creare uno *stack*. Inoltre il cambio di contesto a livello utente può essere fatto con poche istruzioni. L'inconveniente principale dei thread a livello utente però è che una chiamata bloccante di sistema bloccherà l'intero processo e quindi bloccherà tutti i thread del processo.
Questo problema può essere raggirato implementando i threads a livello del kernel ma questo comporta che ogni operazione eseguita su un thread (creazione, distruzione, sincronizzazione e così via) dovrà essere eseguita a livello del kernel richiedendo quindi una chiamata a sistema che risulta essere molto più lenta e costosa come quella di un processo.
La soluzione ai problemi sta nell'uso di una forma ibrida chiamata **processi lightweight**. Un processo leggero viene eseguito nel contesto di un singolo processo (pesante) e per ogni processo ci possono essere più processi leggeri. Oltre a questi il sistema fornisce un pacchetto a livello utente per i threads mettendo a disposizione le solite operazioni. Il pacchetto dei thread è condivisibile da tutti i processi leggeri; questo significa che ogni processo leggero può eseguire il suo thread. Le applicazioni multithread vengono costruite creando dei thread e successivamente assegnando questi thread a un processo leggero.
Il pacchetto dei thread ha una singola routine per pianificare il thread successivo. Quando si crea un processo leggero gli si assegna uno *stack* e lo si mette alla ricerca di un thread da eseguire. I thread in esecuzione sono sono salvati in una tabella in una tabella alla quale i processi leggeri accedono in mutua esclusione tramite l'uso di *mutex* nello spazio utente. Questo significa che la sincronizzazione tra threads è interamente eseguita a livello utente senza la necessità di informare il kernel.
Nel caso in cui vi sia una chiamata di sistema bloccante il contesto di esecuzione passa dalla modalità utente a quella kernel ma continua comunque nel contesto del processo leggero attuale. Nel momento in cui il processo leggero non può più proseguire allora il sistema può decidere di proseguire con un altro processo leggero ritornando alla modalità utente.
I vantaggi di utilizzare un sistema ibrido sono molti. Innanzitutto la creazione, la distruzione e la sincronizzazione dei threads è relativamente poco costosa in quanto avviene a livello utente. Se un processo ha abbastanza processi leggeri allora una chiamata bloccante di sistema non bloccherà l'intero processo. A livello di architetture multiprocessore processi leggeri diversi possono essere eseguiti su CPU diverse. L'unico inconveniente che si presenta è che i processi leggeri devono essere creati e distrutti ma fortunatamente tali operazioni non sono comuni.

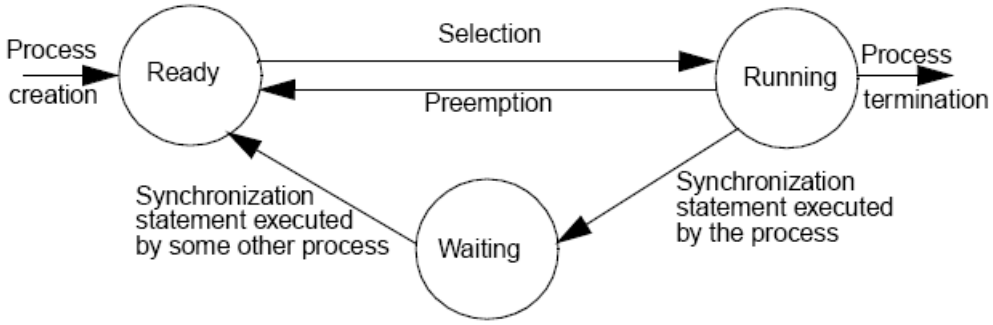

Figura 5: Modello preemtive

4.1.2 Thread nei sistemi distribuiti

Come abbiamo visto il vantaggio principale dell'uso dei threads è che una chiamata di sistema bloccante non blocca l'intero processo. Questa caratteristica è molto vantaggiosa nel caso di realizzazione di comunicazioni multiple come ad esempio la gestione di comunicazioni client-server.

Client multithread Per raggiungere un buon grado di trasparenza alla distribuzione i sistemi distribuiti che operano su reti globali hanno la necessità di nascondere lunghi tempi di propagazione dei messaggi. La tecnica più comune per nascondere la latenza dei messaggi è quella di avviare la comunicazione ed immediatamente iniziare a fare qualcos'altro. Un esempio molto diffuso sono i browser web che iniziano la comunicazione, ricevono una parte del codice HTML ed iniziano a visualizzare la pagina prima ancora di aver concluso la comunicazione.

Server multithread Anche se l'uso di client multithread offre notevoli vantaggi, il vero uso del multithreading è lato server. La pratica dimostra come l'uso del multithreading semplifica la codice e rende più facile lo sviluppo di applicazioni parallele per ottenere un alto livello di prestazioni.
Vediamo il caso di un *file server* dove un **dispacher** riceve in ingresso su di una porta le richieste provenienti da diversi client. Dopo averla esaminata il dispacher seleziona un **worker thread** inattivo a cui assegnare la richiesta. Il *worker* procede con la richiesta ed esegue una lettura bloccante sul file system locale; questo può far si che il thread vengo bloccato in attesa della lettura da disco, in tal caso viene selezionato un altro thread (*worker* o *dispacher*) che procede con la sua esecuzione.

4.1.3 Il modello preemtive

Nei sistemi moderni oltre ai thread viene utilizzato il modello *preemtive*, ovvero è possibile forzare un processo ad abbandonare il suo stato di esecuzione. Solitamente questo meccanismo è utilizzato per implementare un meccanismo di *time slicing* come mostrato in Figura 5.

4.2 I thread in C

Tutti i sistemi UNIX sono multitasking con il sistema preemptive; tradizionalmente tutti i processi sono creati allo stesso modo tramite l'uso della primitiva `fork()`. La *fork*produce una copia del processo chiamante; questa copia è esattamente identica all'origina tranne per il valore

restituito dalla `fork` che per il processo figlio vale *0* mentre nel padre il valore restituito è il *pid* del figlio. Un piccolo esempio: La fork restituisce due copie completamente indipendenti

```
1 /*do parent stuff*/
2 ppid = fork ();
3 if (ppid < 0) {
4         fork_error_function ();
5 } else if (ppid == 0) {
6         child_function ();
7 } else {
8         parent_function ();
9 }
```

Codice 1: Esempio di uso della fork

dello stesso processo, questa indipendenza permette la protezione della memoria e la stabilità ma causa dei problemi quando si vuole che diversi processi lavorino sullo stesso problema; infatti sarebbe necessario usare *pipes* oppure *SysV IPC*. Inoltre il costo di switching tra processi multipli è molto alto, la sincronizzazione è lenta ed esistono dei limiti sul numero di processi che possono essere schedulati efficacemente.
Per questo sono stati introdotti i threads che invece possono essere schedulati all'interno del processo e risolvono molti problemi del lavoro multi processo. L'API più popolare per creare una applicazione multithread in ambiente UNIX è la *pthread* (*POSIX thread*).
Le operazioni che si possono eseguire con quest'API sono la creazione, la distruzione, la sincronizzazione (*join*), lo scheduling, il controllo dei dati e l'interazione con il processo principale. I threads dello stesso processo condividono le istruzioni di processo, gran parte dei dati, i descrittori dei file aperti, i segnali e lo user e il group id. Mentre per ogni thread abbiamo un distinto *ThreadID*, un certo numero di registri, uno stack pointer ed una certa priorità come possiamo vedere Figura 6. Vediamo ora quali sono le funzioni della API *pthread*

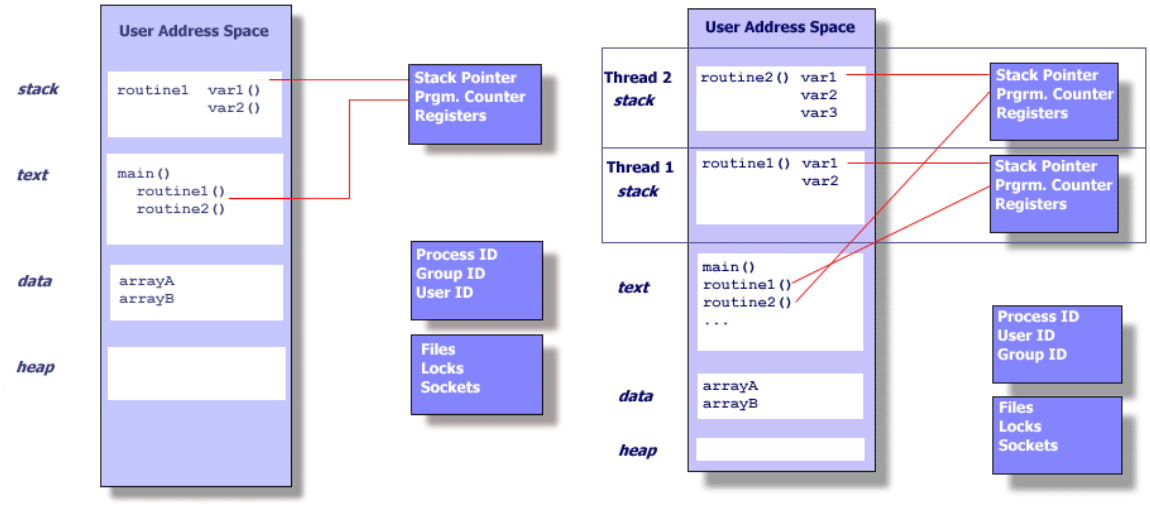

Figura 6: Memoria nel caso di processo (a) e di thread (b)

Thread creation La funzione per la creazione dei thread è: dove i valori sono:

```c
int pthread_create (pthread_t *id, const pthread_attr_t *attr, void *(*routine)(void *), void *arg)
```

Codice 2: Funzione di creazione dei thread

id: un valore che identifica il thread che viene restituito dalla funzione.

attr : un attributo che può essere utilizzato per impostare alcuni valori del thread. Se viene impostato a *NULL* vengono impostati i valori di default.

routine: indica la funzione C che il thread eseguirà una volta creato.

arg: un singolo argomento che può essere passato a`routine`, deve essere passato come riferimento ad un puntatore di tipo `void`; in caso non vi siano valori si imposta a *NULL*.

Thread termination Esistono diversi modi in cui un pthread può terminare:

- Il thread termina la sua routine.
- Nel thread viene richiamata la `pthread_exit`.
- Il thread è cancellato da un altro thread tramite la chiamata della funzione `pthread_cancel`.
- L'intero processo termina quando viene chiamata una delle funzioni `exec` o `exit`.

Tramite la `pthread_exit` è possibile specificare uno stato di terminazione che può essere restituito alla sincronizzazione del thread. Inoltre è molto importante ricordare che la `pthread_exit` non chiude i file ed ogni file aperto all'interno del thread rimane aperto anche alla sua terminazione. Se la funzione *main* termina con una `pthread_exit` prima che i threads siano conclusi i threads proseguono la loro esecuzione altrimenti terminano alla conclusione del *main*. Vediamo un esempio di creazione e terminazione dei thread in C nel Listato 3

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

#define NUM_THREADS 5

/* Esempio di programma che utilizza la libreria pthread */
void *PrintHello(void * threadid) {
       int *temp;
       temp = (int *) threadid;
       printf("\n%d: Hello World!\n", *temp);
       pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
       pthread_t threads[NUM_THREADS];
       int rc,t;

       for (t = 0; t < NUM_THREADS; t++) {
              printf("Creazione del thread %d\n", t);
              rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
              if(rc) {
```

```
23                      printf("ERROR; return code from pthread_create() %d\n",rc);
24                      exit(-1);
25              }
26      }
27
28      for (t = 0; t < NUM_THREADS; t++) {
29              pthread_join(threads[t],NULL);
30      }
31      pthread_exit(NULL);
32  }
```

Codice 3: Esempio di uso della API pthread

Passaggio di argomenti Come abbiamo visto nella *pthread_create* è possibile impostare l'ultimo attributo con un attributo da passare alla routine che il thread eseguirà. Tale attributo deve essere convertito in un puntatore di tipo void. Tale passaggio presenta però alcuni tranelli, vediamo come nel Listato 4 come il passaggio per indirizzo crei un errore nell'esecuzione. Infatti, provando ad eseguire tale programma si rischia che più di un thread acceda contemporaneamente alla variabile t e si rischiano quindi di ottenere dei valori sbagliati.

```
1  int rc, t;
2  for(t=0; t<NUM_THREADS; t++) {
3          printf("Creating thread %d\n", t);
4          rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
5          ...
6  }
```

Codice 4: Errore nel passaggio di argomenti ad un thread

Un possibile risultato di questo codice è quello seguente dove si può vedere che il thread numero 3 stampa il valore 4 anche se il thread numero 4 non è ancora stato creato (In realtà tutti i thread accedono alla variabile t con un ritardo in quanto manca la stampa del thread numero 0).

```
Creazione del thread 0
Creazione del thread 1
1: Hello World!
Creazione del thread 2
2: Hello World!
Creazione del thread 3
3: Hello World!
4: Hello World!
Creazione del thread 4
```

Per passare un argomento ad una routine è necessario controllare l'accesso ai dati da parte dei threads in modo che non vi siano possibili conflitti come nel caso del Listato 5. Nel quale viene passato ad ogni routine un puntatore ad un dato diverso.

```
1  int *taskids[NUM_THREADS];
2  for(t=0; t<NUM_THREADS; t++) {
3          taskids[t] = (int *) malloc(sizeof(int));
4          *taskids[t] = t;
5          printf("Creating thread %d\n", t);
```

```
6         rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
7         ...
8 }
```

Codice 5: Metodo corretto nel passaggio di argomenti ad un thread

Joining threads L'operazione di *join* è uno dei modi nel quale si può implementare la sincronizzazione tra thread.

```
1 int pthread_join(pthread_t thid,void **thread_return)
```

dove i diversi campi sono:

thid è l'identificativo del thread su cui fare la join

thread_return è il possibile valore di ritorno che si ottiene dall'invocazione della `pthread_exit`

Il funzionamento della funzione di *join* è specificato in Figura 7

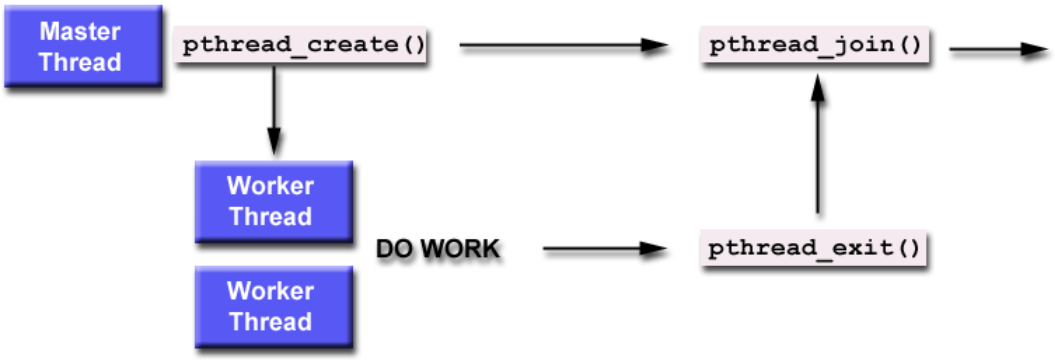

Figura 7: Funzionamento dell'operazione di join

I mutex Un *mutex* funziona come un *lock* proteggendo l'accesso a dei dati condivisi. Il concetto principale è che un solo thread alla volta può bloccare una variabile di tipo mutex, se più thread tenta di bloccare un mutex soltanto uno di questi effettuerà l'operazione con successo, inoltre i thread non possono bloccare un determinato mutex finché il thread che lo detiene non lo libera.
È compito del programmatore assicurarsi che ogni thread che utilizza dei dati condivisi usi i mutex.
Una variabile di tipo mutex può essere dichiarata sfruttando la parola chiave `pthread_mutex_t`, per inizializzare tale variabile, invece, è possibile sfruttare due metodi:

- Il primo è l'inizializzazione statica come ad esempio

  ```
  pthread_mutex_t mymutx = PTHREAD_MUTEX_INITIALIZER;
  ```

- Il secondo metodo è l'inizializzazione dinamica richiamando la routine

  ```
  pthread_mutex_init
  ```

In questo caso è possibile impostare alcuni parametri dell'oggetto mutex tramite le primitive `pthread_mutexattr_init` e `pthread_mutexattr_destroy` che rispettivamente creano e distruggono gli attributi di un mutex

In entrambi i casi di inizializzazione l'oggetto mutex è inizializzato *unlocked*. Infine, la routine `pthread_mutex_destroy` permette di rilasciare un mutex di cui non si ha più bisogno.
Esistono tre primitive per la gestione dei mutex, queste sono:

- `pthread_mutex_lock:` che si usa per acquisire un lock su di una variabile, nel caso in cui tale lock sia detenuto da un altro thread il thread che ha richiesto il lock si blocca finché il blocco non viene rilasciato.
- `pthread_mutex_trylock` molto simile alla routine precedente solo che nel caso in cui il blocco sia detenuto da un altro thread allora la routine restituisce un codice di errore che indica *busy*; è molto utile nel caso si vogliano prevenire condizioni di deadlock.
- `pthread_mutex_unlock:` questa routine permette di rilasciare il lock in possesso del thread ma restituisce un errore nel caso in cui si voglia rilasciare un lock su di una variabile già sbloccata o bloccata da un altro thread.

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  typedef struct{
6         double *a;
7         double *b;
8         double sum;
9         int veclen;
10 } DOTDATA;
11
12 /*Define global variables and mutex*/
13 #define NUMTHREADS 4
14 #define VECLEN 100
15 DOTDATA dotstr;
16 pthread_t callThd[NUMTHREADS];
17 pthread_mutex_t mutexsum;
18
19 void *dotprod(void *arg) {
20        int i, start, end, offset;
21        double mysum;
22
23        offset = (int) arg;
24        start = offset * dotstr.veclen;
25        end = start + dotstr.veclen;
26        mysum = 0;
27        for (i = start; i < end; i++) {
28               mysum += (dotstr.a[i] * dotstr.b[i]);
29        }
30
31        pthread_mutex_lock (&mutexsum);
32        dotstr.sum += mysum;
33        pthread_mutex_unlock (&mutexsum);
34        pthread_exit((void *)0);
```

```
35 }
36
37 int main (int argc, char *argv[]) {
38        int i;
39        int status;
40
41        /* Assign storage and initialize values */
42        dotstr.a = (double*) malloc (NUMTHREADS*VECLEN*sizeof(double));
43        dotstr.b = (double*) malloc (NUMTHREADS*VECLEN*sizeof(double));
44        for (i=0; i<VECLEN*NUMTHREADS; i++) {
45               dotstr.a[i]=1;
46               dotstr.b[i]=1;
47        }
48        dotstr.veclen = VECLEN;
49        dotstr.sum=0;
50        /* initialize the mutex */
51        pthread_mutex_init(&mutexsum, NULL);
52        /* Create threads to perform the dotproduct */
53        for(i=0;i<NUMTHREADS;i++) {
54               pthread_create(&callThd[i], NULL, dotprod, (void *)i);
55        }
56        /* Wait on the other threads */
57        for(i=0;i<NUMTHREADS;i++) {
58               pthread_join( callThd[i], (void **)&status);
59        }
60        /* After joining, print out the results */
61        printf ("Sum = %f \n", dotstr.sum);
62        free (dotstr.a);
63        free (dotstr.b);
64        pthread_mutex_destroy(&mutexsum);
65        pthread_exit(NULL);
66 }
```

Codice 6: Esempio di uso delle variabili mutex

Condition variables Mentre i mutex implementano la sincronizzazione tramite il controllo degli accessi sui dati le *condition variables* permettono ai thread di sincronizzarsi in base ad un determinato valore di un dato. Senza quest'aspetto dei thread bisognerebbe implementare un polling per verificare quando una particolare condizione viene riscontrata. Le *condition variables* sono un modo per ottenere lo stesso risultato senza il polling, e possono essere utilizzate anche insieme ai mutex. L'utilizzo principale sono tutti quei problemi della categoria**produttore-consumatore**.
Come per i mutex le condition variabile sono dichiarate utilizzando la parola chiave `pthread_cond_t`; per l'inizializzazione esistono due metodi:

- Statico

  ```
  pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER
  ```

- Dinamico tramite la funzione `pthread_cond_init` che permette di settare anche gli attributi della variabile tramite le due primitive

  ```
  pthread_condattr_init
  pthread_condattr_destroy
  ```

Che permettono rispettivamente di inizializzare e distruggere gli attributi della variabile

Infine tramite la primitiva `pthread_cond_destroy` è possibile liberare una variabile condizionale che non è più necessaria.
Per la gestione di questo tipo di variabile esistono diverse primitive:

- `pthread_cond_wait` è una routine che il thread finché una determinata condizione non si verifica, se chiamata quando vi è un lock attivo la routine sblocca il mutex e lo blocca nuovamente quando il thread si sveglia.
- `pthread_cond_signal` questa routine risveglia gli altri thread in attesa di una *condition variables*
- `pthread_cond_brodcast` può essere utilizzata al posto della routine precedente se ci sono più thread bloccati in uno stato di *wait*.

4.3 Concorrenza in Java

Come per il C anche il Java fornisce il supporto alla concorrenza a livello di linguaggio, esso mette a disposizione delle classi per istanziare ed eseguire nuovi thread più i metodi di sincronizzazione e le variabili di condizione. Il modo più semplice per creare un thread è quello di utilizzare la classe `java.lang.Thread` in questo caso è sempre necessario implementare un metodo `run()`.

```
1  public class MyThread extends Thread {
2         private String message;
3         public MyThread(String m) {message = m;}
4         public void run() {
5                 for(int r=0; r<20; r++)
6                         System.out.println(message);
7         }
8  }
9
10 public class ProvaThread {
11        public static void main(String[] args) {
12                MyThread t1,t2;
13                t1=new MyThread("primo␣thread");
14                t2=new MyThread("secondo␣thread");
15                t1.start();
16                t2.start();
17        }
18 }
```

Codice 7: Uso della classe Thread in Java

Come vediamo la nostra classe che implementa un thread estende l'oggetto *Thread*, in questa classe viene fatto l'override del metodo `run()` il quale non è altro che la routine che viene eseguita dal thread. Per far partire il thread è necessario richiamare il metodo `start()` dopo aver creato un nuovo oggetto.
Un'altra possibile soluzione è l'utilizzo dell'interfaccia `Runnable` la quale specifica soltanto che deve esistere un metodo `run()` che deve essere implementato. La classe `Thread` implementa anch'essa l'interfaccia `Runnable`. Come vediamo nel Listato 8 a differenza del caso precedente oltre all'oggetto *MyThread* deve anche essere creato un oggetto *Thread* corrispondente, al quale viene poi passato l'oggetto MyThread, ed infine il metodo `start()`viene invocato sull'oggetto Thread.

```
1  public class MyThread implements Runnable{
2          private String message;
3
4          public MyThread (String m) { message = m; }
5          public void run() {
6                  for (int r = 0; r < 20; r++)
7                          System.out.println(message);
8          }
9  }
10
11 class ProvaThread {
12         public static void main (String[] args) {
13                 Thread t1, t2;
14                 MyThread r1, r2;
15                 r1 = new MyThread("PrimoThread");
16                 r2 = new MyThread("Secondo␣Thread");
17                 t1 = new Thread(r1);
18                 t2 = new Thread(r2);
19                 t1.start();
20                 t2.start();
21         }
22 }
```

Codice 8: Utilizzo dell'interfaccia Runnable

L'esecuzione dei thread non segue un ordine predefinito ma lo stesso codice può produrre risultati diversi su diversi computer o addirittura sullo stesso. Questo caratteristica è chiamata *non-determinismo* ed è un punto focale nella concorrenza.
Java di per se implementa il modello *preemtive* e nel caso sia disponibile un meccanismo di *time-slicing* allora java esegue i thread con la stessa priorità tramite una meccanismo di *round-robin*.
Per definire quando un sistema multithread è corretto si devono rispettare due proprietà:

- *Sicurezza:* Un sistema si dice sicuro quando gli eventi malevoli non accadono.
- *Longevità:* Un sistema è longevo quando le cose buone possono accadere.

I possibili guasti che rientrano nella categoria Sicurezza sono quei guasti che avvengono a livello di esecuzione come i conflitti *read/write* e *write/write*. I meccanismi che invece riguardano la Longevità sono quei meccanismi che bloccano l'esecuzione del programma come:

- Lock
- Waiting
- CPU contention

Solitamente, purtroppo, le cose più semplici che si possono fare per aumentare la longevità ne riducono però la sicurezza e vice versa.
Vediamo ora quali sono i meccanismi che Java mette a disposizione per supportare la concorrenza.

Exclusion In un sistema sicuro ogni oggetto protegge se stesso da possibili violazioni della sua integrità. le tecniche di esclusione preservano l'invariante di un oggetto. Tre sono le tecniche principali per permettere l'*esclusione*:

- Immutabilità
- Esclusione dinamica (Locking)
- Esclusione strutturale

Per quanto riguarda l'**immutabilità** si ottiene creando le classi in modo che gli oggetti proteggano se stessi come nel Listato 9

```
1 class ImmutableAdder {
2         private final int offset;
3         public Immutableadder (int a) {
4                 offset = a;
5         }
6         public int addOffset (int b) {
7                 return offset + b;
8         }
9 }
```

Codice 9: Esempio di oggetto immutabile

I vantaggi di questa tecnica sono il fatto che non richiede sincronizzazione ed è molto utile per condividere degli oggetti tra i threads, ma sfortunatamente ha dei limiti di applicabilità.
Per parlare di sincronizzazione introduciamo prima l'esempio del Listato 10

```
1  public class RGBColor {
2         private int r;
3         private int g;
4         private int b;
5
6         public void setColor (int r, int g, int b)
7                 checkRGBVals(r, g, b);
8                 this.r = r;
9                 this.g = g;
10                this.b = b;
11        }
12 }
```

Codice 10: Esempio sincronizzazione

Ora immaginiamo che due thread chiamati *red* e *blue* vogliano impostare contemporaneamente il loro colore sullo stesso oggetto di tipo `RGBColor` a questo punto potrebbero verificarsi dei problemi in quanto i due thread tentano di scrivere lo stesso dato violando così la sua integrità.
Per risolvere questo problema java tramite il locking serializza l'esecuzione del codice dichiarato *synchronized*. Ogni istanza di un oggetto possiede tali meccanismi di lock in quanto derivati dalla classe `Object`, l'unica eccezione si ha con l'utilizzo di array, infatti, bloccare un array non blocca gli elementi di tale array.
Esistono due modi per bloccare una parte di codice, si può dichiarare *synchronized* un intero metodo o un singolo blocco di codice, in caso di singolo blocco la funzione `synchronized` richiede l'oggetto sul quale effettuare il lock (Listato 11 e Listato 12).

```
synchronized (object) {
        //Lock is held
        ...
}
//Lock is released
```

Codice 11: Sincronizzazione di una parte di codice

```
synchronized void f() {
        //Lock is held
        /* Body */
}
//Lock is released
```

Codice 12: Sincronizzazione di un metodo

I lock vengono automaticamente acquisiti all'ingresso del blocco o del metodo dichiarato `synchronized` e rilasciato all'uscita da esso.
Alcune regole chiave per l'uso della sincronizzazione sono:

- Sempre quando si effettua un aggiornamento a dei campi di un oggetto.

```
synchronized (point) {
        point.x = 5; point.y = 7;
}
```

- Tutte le volte che si accede a dei dati che potrebbero essere aggiornati.

```
synchronized (point) {
        if (point.x > 0) {...}
}
```

- Si può fare a meno di sincronizzare parti di metodo stateless

```
public void f() {
        synchronized (this) {
                state = ...;
        }
        operations();
}
```

- **Mai** sincronizzare parti di codice che contengono invocazioni ad altri oggetti

```
public void f() {
        synchronized (this) {
                ...
        }
        h.foo();
}
```

La strategia più sicura (ma non la più efficace) per realizzare un'applicazione OO concorrente è quella di utilizzare oggetti completamente sincronizzati, anche detti oggetti *atomici*, nei quali tutti i metodi sono sincronizzati, non esistono campi pubblici o altri tipi di violazione nell'incapsulamento, tutti i metodi sono finiti e hanno modo di rilasciare il lock, tutti i campi sono

inizializzati ad un valore consistente nel costruttore, ed infine lo stato dell'oggetto è consistente sia all'inizio che alla fine di ogni metodo anche in presenza di eccezioni.
Uno dei problemi principali della programmazione concorrente è il *deadlock*, tale problema si verifica quando due o più oggetti sono acceduti da due o più threads e tali thread detengono un lock mentre tentano di acquisire un lock detenuto da un altro thread.
L'assegnamento di un valore ad una variabile è un'operazione atomica (a parte per i *long* e i *double*), questo significa che generalmente non è necessario sincronizzare l'accesso ad una variabile. Tuttavia i thread solitamente memorizzano il valori delle variabile in memoria locale, questo significa che se un thread cambia il valore di una variabile un altro thread non vede il cambiamento. Per evitare questo meccanismo bisogna sincronizzare la variabile oppure dichiararla di tipo `volatile` che significa che ogni volta che una variabile è usata deve prima essere letta dalla memoria principale.
Il confinamento implementa l'incapsulamento garantendo che al massimo un'attività alla volta acceda agli oggetti. Questo meccanismo permette l'accesso ad un solo thread alla volta senza utilizzare i locking dinamici. Il punto principale è quello avere un punto di uscita dal thread. Esistono quattro categorie per verificare che un riferimento r ad un oggetto x può uscire da un metodo m:

- m passa r come argomento di un'invocazione ad un metodo o ad un costruttore di un oggetto
- m passa r come valore di ritorno di un metodo.
- m registra r in un campo accessibile da altre attività
- m rilascia un riferimento che però permette l'accesso ad r

Per quanto riguarda le collezioni, il framework `java.util.Collection` basata su uno schema *Adapter* permette la sincronizzazioni delle classi collection, infatti, ad eccezione di `Vector` e `Hashtable` le classi base per le collezioni (come `java.utili.ArrayList`) sono non sincronizzate. Sono state così costruite una serie di classi sincronizzate attorno alle classi base come la `Collection.synchronizedList`.
Come abbiamo detto prima però la sincronizzazione non è molto efficiente infatti richiamare un metodo sincronizzato richiede un tempo quattro volte maggiore rispetto a metodi non sincronizzati; inoltre, esso riduce la concorrenza e diminuisce le performance, infine non vi è alcun modo di controllare il meccanismo dei lock.
Con java versione 5 sono stati introdotti nuovi meccanismi di sincronizzazione come la *sincronizzazione condizionata.* Prendiamo come esempio un parcheggio con una certa capacità e dei metodi che permettono l'arrivo e la partenza di automobili come esemplificato nel Listato 13.

```
1  public class CarParkControl {
2         protected int space;
3         protected int capacity;
4
5         public CarParckControl (int n) {
6                capacity = space = n;
7         }
8
9         syncronized public void arrive() {
10               ...; --space; ...;
11        }
12        syncronized public void depart() {
```

```
13                ...; ++space; ...;
14        }
15  }
```

Codice 13: Esempio di controllore di un parcheggio

Come per il C esistono però dei metodi che permettono una gestione più efficiente del controllore rispetto all'uso della syncronized; questi metodi sono:

- `public final void notify()`: che risveglia un singolo thread in attesa.
- `public final void notifyAll()`: risveglia tutti i thread in attesa.
- `public final void wait() throws InterruptedException`: pone il thread in attesa di un *notify*. Quando un thread viene posto in uno stato di wait esso rilascia il lock acquisito e lo riacquista al suo risveglio.

```
1   public class CarParkControl {
2         private int space;
3         private int capacity;
4
5         public CarParkControl (int n) {
6                 capacity = space = n;
7         }
8         synchronized public void arrive() throws InterruptedException {
9                 while (space == 0) wait();
10                --space;
11                notifyAll();
12        }
13        synchronized public void depart() throws InterruptedException {
14                while (space == capacity) wait();
15                ++space;
16                notifyAll();
17        }
18  }
```

Codice 14: Esempio di variabili condizionali in Java

Si può ridurre l'overhead dovuto al contex-switching sostituendo la `notifyAll` con la `notify`. Tale meccanismo può essere usato per migliorare le performance quando si è certi che almeno un thread è in atteso per eseguire un lavoro.
Alla condizione di *wait* è possibile associare un timer molto utile per migliorare la longevità del sistema in quanto tende a risolvere in modo automatico i deadlock.

5 Comunicazione

La comunicazione tra processi è il cuore di tutti i sistemi distribuiti, infatti, non ha senso studiare i sistemi distribuiti senza esaminare come i processi su posti macchine diverse si scambiano le informazioni. La comunicazione nei sistemi distribuiti si basa sempre sullo scambio dei messaggi a basso livello come fornito dalla rete sottostante, anche se ciò rende la realizzazione del sistema distribuito molto complicata.
In questo capitolo analizzeremo prima di tutto le regole che i diversi processi devono rispettare per comunicare tra loro, queste regole sono conosciute anche come protocolli e solitamente vengono strutturati a livelli.
Analizzeremo in seguito tre modelli di comunicazione molto diffusi, le chiamate a procedure remote (RPC, *remote procedure call*), i middleware orientati agli oggetti (MOM, *message-oriented middleware*) e gli *streaming* di dati. Ed infine analizzeremo il problema dell'invio di dati a destinatari multipli, ovvero, il *multicast*.

5.1 Il modello OSI

A causa della mancanza di una memoria condivisa tutta la comunicazione nei sistemi distribuiti avviene mediante tramite l'invio e la ricezione di messaggi a basso livello. Quando un processo A vuole comunicare con un processo B prima di tutto costruisce un messaggio nel proprio spazio degli indirizzi e poi effettua una *chiamata di sistema* che fa in modo che il SO si occupi dell'invio del messaggio attraverso la rete fino a raggiungere B. Anche se il principio è semplice esistono diversi ostacoli al completamente di questa operazione, prima di tutto A e B devono concordare sul significato dei bit inviati, esistono molti altri aspetti sui quali bisogna accordarsi, come il valore in volt usati per indicare un bit a 1, individuare l'ultimo bit del messaggio, bisogna inoltre capire se un messaggio è stato danneggiato o perso ecc.
Per poter trattare facilmente i numerosi aspetti di una comunicazione la *international standards organization* (ISO) ha sviluppato un modello di riferimento che identifica i vari livelli di comunicazione coinvolti, gli assegna dei nomi standard e identifica le diverse funzionalità per ogni livello. Questo modello è chiamato **open system interconnection reference model** o più comunemente modello **ISO OSI** ed è illustrato in Figura 8 Tuttavia è bene far presente che i protocolli sviluppati nel modello OSI non sono stati mai ampiamente utilizzati, tuttavia il modello sottostante si è rivelato particolarmente utile per comprendere le reti di computer.
Il modello OSI è progettato per consentire la comunicazione tra sistema aperti ovvero tra sistemi preparati per comunicare tramite regole standard che ne regolano il formato, i contenuti ed il significato di messaggi ricevuti ed inviati. Queste regole sono dette **protocolli** e devono essere concordate a priori per permettere la comunicazione tra gruppi di computer.
Esistono due grandi tipologie di protocolli, quelli **orientati alla connessione** nei quali mittente e destinatario stabiliscono esplicitamente una connessione prima di scambiarsi dei dati ed alla fine devono rilasciare tale connessione. Con i protocolli **senza connessione** non è necessaria alcuna premessa, quando il messaggio è pronto il mittente invia il messaggio come nel caso di un invio di una lettera.
Nel modello OSI la comunicazione è suddivisa in sette livelli o *layer*, ogni livello tratta uno specifico aspetto della comunicazione, in modo da suddividere il problema in parti gestibili ciascuna delle quali può essere trattata indipendentemente. Per realizzare questo meccanismo ogni livello fornisce un'interfaccia al livello superiore, la quale specifica un insieme di operazioni che il livello è pronto a fornire.
Quando un processo A sulla macchina 1 vuole comunicare con un processo B sulla macchina 2

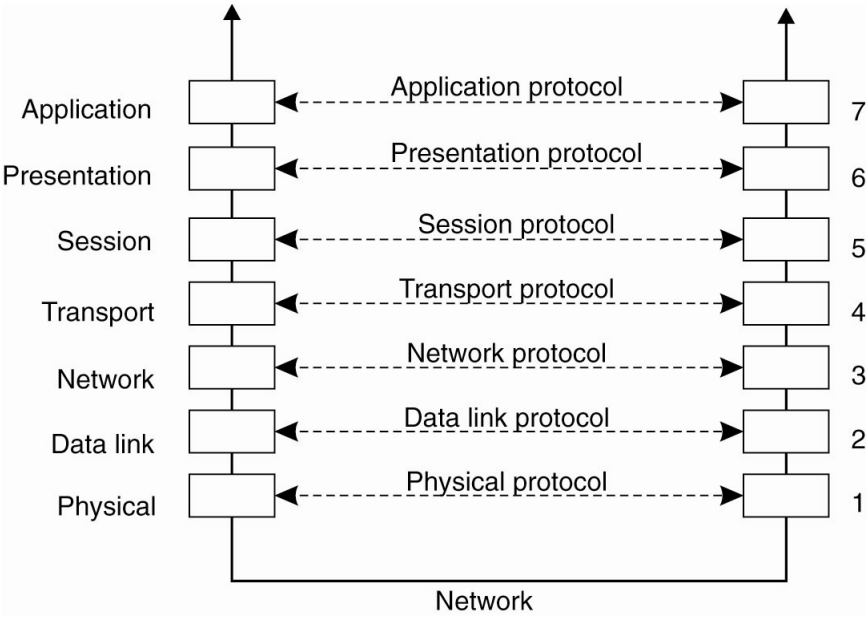

Figura 8: Modello ISO OSI

costruisce un messaggio e lo passa al livello applicativo sulla sua macchina; tale livello potrebbe essere una procedura ad una libreria oppure implementato in qualche altro modo come ad esempio all'interno del sistema operativo. Il software a livello applicativo aggiunge un intestazione (*header*) all'inizio del messaggio e passa tutto al livello di presentazione tramite l'interfaccia tra i livelli 6 e 7. A sua volta il livello di presentazione passa il messaggio al livello di sessione non prima di aver aggiunto il suo *header* e così via. Alcuni livelli aggiungono oltre all'header anche un *trailer* in chiusura al pacchetto come mostrato in Figura 9. Quando il messaggio raggiunge

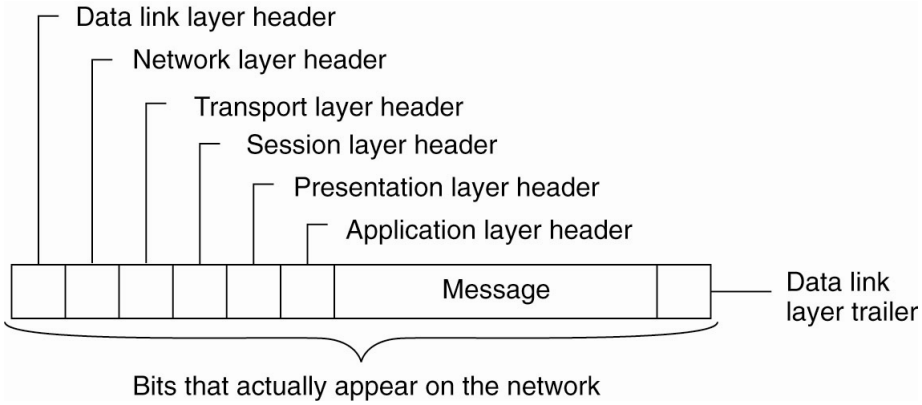

Figura 9: Elenco header e trailer di un messaggio che attraversa i vari layer

il fondo esso viene trasmesso dal livello fisico sul mezzo di trasmissione. Quando il messaggio raggiunge la macchina 2 viene passato verso l'alto e ogni livello stacca la sua intestazione e la esamina, infine il messaggio raggiunge il suo destinatario , ovvero il processo B il quale può rispondere utilizzando il percorso inverso.
Ogni livello ha il suo protocollo che può essere cambiato indipendentemente dagli altri ed è proprio questa indipendenza a rendere i protocolli a livelli interessanti.

L'insieme di protocolli usati in un particolare sistema è detto **suite di protocolli** o **stack di protocolli**.

5.1.1 I layer

Analizziamo ora i diversi layer che compongono il modello OSI, vedremo quali sono le loro funzionalità e dove possibile indicheremo quali protocolli sono attualmente utilizzati nell'ambito di internet. Partiamo dai tre livelli più bassi della suite di protocolli, insieme questi tre livelli implementano le funzioni base di una rete di computer.
Il *livello fisico* trasmette gli 0 e gli 1 sul canale fisico di comunicazione, elementi importanti per questo livello sono la quantità di volt che contraddistingue gli 0 e gli 1, il numero di bit al secondo, la possibilità di trasmettere in entrambe le direzioni, infine rivestono una notevole importanza la forma dei connettori (*plug*) ed il numero di piedini (*pin*). Il protocollo che identifica questo livello ha a che fare con la standardizzazione delle interfacce elettriche e meccaniche e di segnale; un esempio di questi standard è l'interfaccia RS-232-C per la comunicazione seriale.
Il livello *data link* si occupa della rilevazione e della correzione degli errori di trasmissione. Per fare ciò il data link layer raggruppa i bit in unità chiamate **frame** e controlla che ogni frame sia ricevuto correttamente. Per eseguire questo compito il livello di collegamento dei dati applica un *pattern* di bit all'inizio ed alla fine di ogni frame per delimitarli ed eseguire una **somma di controllo** (*checksum*) sommando tramite algoritmi specifici i byte del frame ed inserisce tale somma nei campi all'inizio o alla fine del frame. All'arrivo di un nuovo pacchetto il livello data link esegue la somma sui dati arrivati e la confronta con quella inviata insieme al pacchetto nel caso le due checksum coincidano il pacchetto è considerato corretto, in caso contrario il destinatario ne richiede la ritrasmissione grazie al numero di sequenza inserito nell'header del data link.
Il *livello di rete* si occupa del **routing** dei pacchetti, ovvero, della scelta del percorso ottimale che permetta ad un pacchetto di andare da un mittente ad un destinatario. Il problema si complica in quanto il percorso più breve non è sempre quello ottimo. Al momento il protocollo di rete più utilizzato è l'IP (**Internet Protocol**) senza connessione che è parte dei protocolli Internet. Un **pacchetto** IP può essere inviato senza alcun preparativo. Ogni pacchetto è instradato verso la sua destinazione indipendentemente da tutti gli altri.
Il *livello di trasporto* costituisce l'ultima parte di quelli che possiamo definire *stack del protocollo di rete di base* nel senso che implementa tutti i quei servizi non forniti dall'interfaccia del livello di rete ma necessari per l'implementazione di applicazioni di rete. In altre parole il livello di trasporto in un qualcosa di utilizzabile.
Uno dei compiti del livello di trasporto è quello di fornire una connessione affidabile anche se molte applicazioni gestiscono autonomamente la perdita di pacchetti. Quando arriva un messaggio dal livello applicativo il livello di trasporto lo spezza in parti sufficientemente piccole per essere trasmesse ed assegna un numero di sequenza. Le informazioni nell'header del livello di trasporto riguardano il numero di pacchetti inviati, il numero di pacchetti ricevuti, quali devono essere ritrasmessi e così via.
Connessioni di trasporto affidabili possono essere costruite sopra servizi di rete orientati alla connessione o senza connessione. Nel primo caso i pacchetti arriveranno nella sequenza corretta, nel secondo caso non vi è alcun metodo per stabilire a priori l'ordine di arrivo dei pacchetti, ed è compito del software del livello di trasporto di riordinare i dati. Un aspetto importante del livello di trasporto è quello di fornire questo comportamento *end-to-end*.
Il protocollo di trasporto di Internet è chiamato **TCP (transmission control protocol)**. La combinazione TCP/IP è oggi lo standard de facto per la comunicazione in rete. Oltre al TCP esiste anche un protocollo di trasporto senza connessione chiamato UDP (*universal datagram*

protocol) che è molto simile all'IP con alcune aggiunte minori.
Ulteriori protocolli sono proposti regolarmente, un esempio è il **real-time transport protocol (RTP)** il quale specifica il formato dei pacchetti per il trasferimento dei dati in tempo reale ma non fornisce alcun meccanismo per garantire la loro consegna.
Sopra il livello di trasporto sono il modello OSI identifica altri tre livelli in realtà solo il livello applicativo è sempre usato. Per quanto riguarda i sistemi middleware nel il modello OSI nel l'approccio Internet sono soddisfacenti. Ad esempio il livello di sessione mette a disposizione funzionalità per il controllo del dialogo fornendo la possibilità di impostare *checkpoint* che in caso di *crash* permettano la ripresa della trasmissione senza ricominciare da capo. Tale meccanismo non è mai implementato nella suite di protocolli Internet. Tuttavia nel contesto di soluzioni middleware il concetto di sessione sono piuttosto cruciali. Il compito del livello di prestazione è invece quello di dare un significato ai bit trasmessi. La maggior parte dei messaggi è composta da informazioni strutturate come nomi, indirizzi, quantità di denaro e così via. Nel *livello di presentazione* è possibile definire dei *record* contenenti campi come quelli precedentemente elencati in modo che il mittente possa comunicare al destinatario che il pacchetti contengono dei dati in un certo formato.
Il *livello applicativo* era originariamente inteso per contenere semplici applicazioni di rete come l'e-mail o il trasferimento di file, ora è divenuto il contenitore di tutte le applicazioni. Ciò che manca in questo livello è una netta distinzione tra protocolli di una specifica applicazione e protocolli più generali.

Protocolli middleware Il middleware pur essendo posizionato in maniera logica nel livello applicativo contiene molti protocolli specifici che ne giustificano l'esistenza in un livello proprio. i protocolli per supportare una gran varietà di servizi middleware sono diversi, molti sono pensati per stabilire un autenticazione, non essendo legati ad una specifica applicazione questo tipo di protocolli può essere accorpato in un sistema middleware come servizio generale. Un altro esempio di protocollo che rientra a far parte dei protocolli middleware è quello del *commit distribuito* dove un operazione è portata a termine solo se è portata a termine in tutte le sue parti. Questa proprietà è detta **atomicità**ed è ampiamente utilizzata in tutti i tipi di transizioni.
Come abbiamo visto dai due esempi appena fatti i protocolli middleware supportano servizi di comunicazione di alto livello, ma oltre a questi esistono protocolli per supportare lo *stream* di dati in tempo reale, oppure, protocolli più specifici del livello di trasporto ma che, dovendo tener conto dei requisiti delle applicazioni devono essere situati ad un livello più alto di quello di trasporto come ad esempio il caso di *multicast* che devono garantire la scalabilità.
Il modello che si viene così a creare è quello di Figura 10 nel quale il livello di sessione e quello di presentazione vengono sostituiti da un unico livello middleware contenete quei protocolli indipendenti dalle applicazioni.

5.1.2 Tipi di comunicazione

Esistono diverse alternative nella comunicazione che il middleware mette a disposizione delle applicazioni; partiamo dall'esempio mostrato in Figura 11. In questo caso possiamo pensare che ogni *host* esegua uno *user agent* ovvero un processo che esegue le operazioni di comunicazione tra i vari sistemi.

Prendiamo ora come esempio il caso di un invio di e-mail in questo caso i due *user agent* saranno rispettivamente il sistema che invia e quello che riceve le e-mail. L'agente dal lato del mittente passa la mail al sistema per la consegna delle mail nella convinzione che tale sistema consegni la mail, l'agente dal lato del destinatario a sua volta si connette al sistema per sapere se è giunta qualche nuova mail in caso affermativo la mail viene trasferita dal sistema allo user

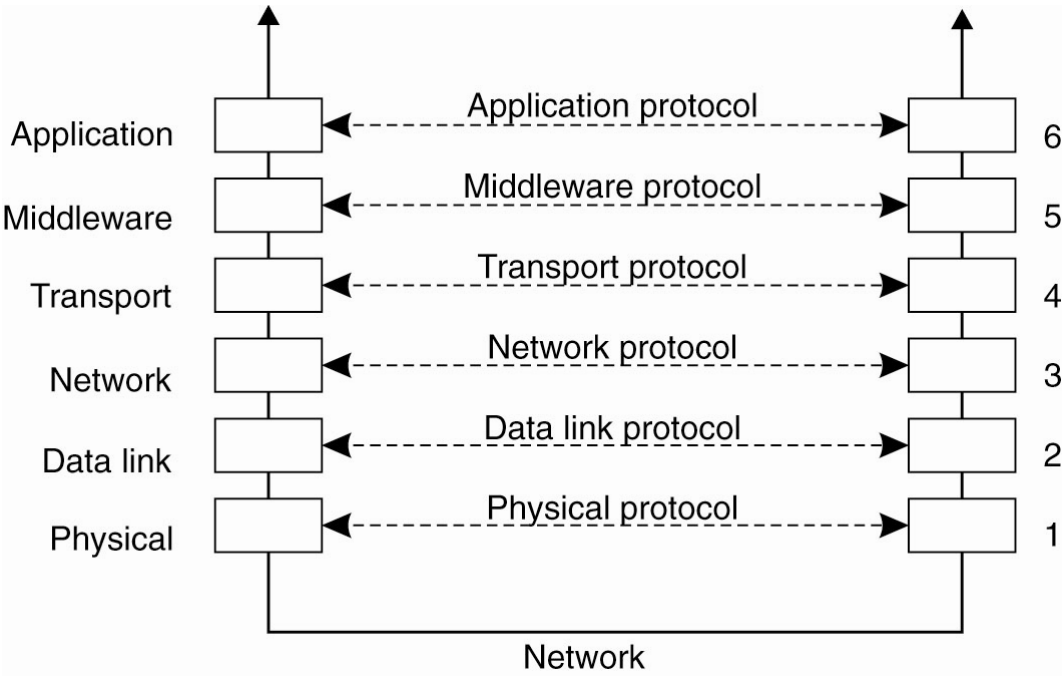

Figura 10: Modello OSI-Middleware

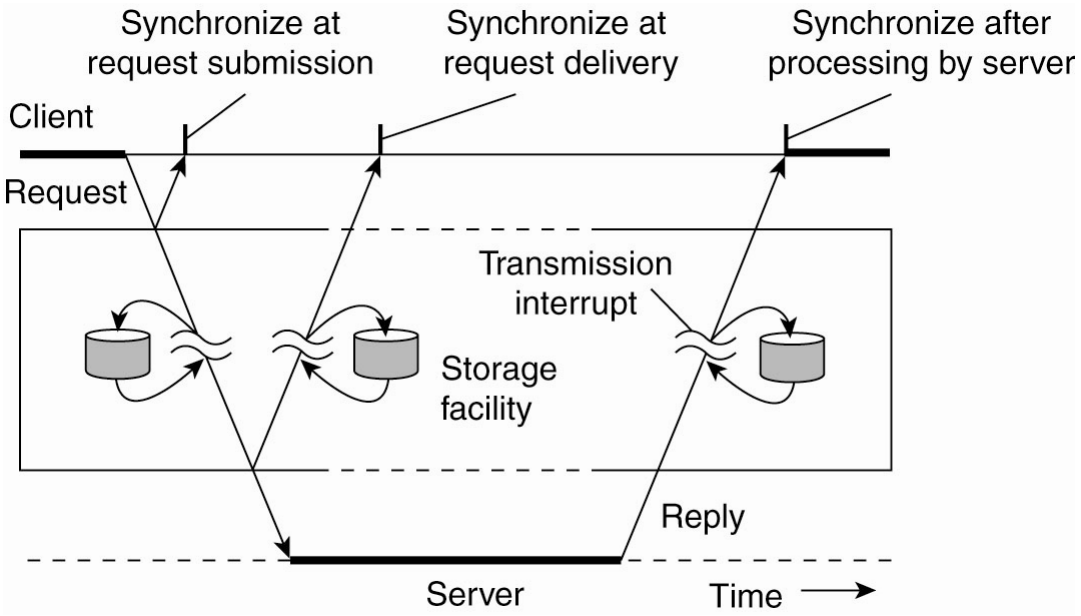

Figura 11: Esempio di tipi di comunicazione

agent del destinatario.
Questo tipo di meccanismo è detto **comunicazione persistente** in quanto il messaggio viene memorizzato dal middleware per tutto il tempo necessario affinché la consegna non vada a buon fine durante questo periodo non è necessario che i due elementi siano in esecuzione contemporaneamente. Nel caso di **comunicazione transiente** invece, il sistema memorizza i messaggi scambiati solo finché sia il mittente che il destinatario sono in esecuzione.
Oltre che persistente o transiente una comunicazione può essere asincrona o sincrona. Si parla di comunicazione **asincrona** quando il mittente continua la sua elaborazione subito dopo l'invio del messaggio. Nel caso di comunicazione **sincrona**, invece, il mittente è bloccato fino a quando la sua richiesta non viene accettata. Ciò può avvenire fino a quando il middleware non comunica la presa in consegna della richiesta, oppure quando la richiesta non viene consegnata al destinatario l'ultima possibilità è che il mittente resta bloccato fino alla fine dell'elaborazione da parte del destinatario e invio di una risposta all'elaborazione.
Esistono molti tipi in cui combinazione persistente, transiente, sincrona e asincrona possono essere combinati ma le più diffuse sono la persistenza e la sincronizzazione, la transiente con la sincronizzazione alla fine dell'elaborazione.
Dobbiamo distinguere, infine tra comunicazione discreta e a *stream*.

5.2 Chiamate a procedure remote

Molti sistemi si basano sullo scambio di messaggi esplicito tra processi, questo scambio avviene tramite l'utilizzo delle procedure `send` e `recive` che però non nascondono del tutto la comunicazione. Una nuova tecnica fu introdotta nel 1984 quando si pensò di permettere ai programmi di richiamare procedure situate su altre macchine. Quando un processo sulla macchina A richiama una procedura sulla macchina B il processo chiamante viene sospeso e ha luogo l'elaborazione sulla macchina B. Le informazioni sono passate dal chiamante al chiamato tramite i parametri e ritornano indietro come risultato della procedura. Questo tipo di meccanismo è conosciuto come **chiamata a procedura remota** o **RPC** (*remote procedure call*).
Sebbene l'idea di base risulti molto semplice i problemi relativi sono molti, primo fra tutti visto che chiamante e chiamata risiedono su due macchine diverse lo spazio degli indirizzi è completamente diverso, inoltre, il passaggio di parametri e dei risultati non è così semplice in quanto le macchine potrebbero avere rappresentazione dei dati differenti.

5.2.1 Operazioni di base sulle RPC

Iniziamo a vedere come funzionano realmente le RPC partendo dall'analisi di una chiamata a procedura locale per poi analizzare le chiamate remote.

Chiamate a procedura locali Per capire come lavorano le RPC è necessario capire comprendere come lavorano le chiamate a procedura convenzionali, ovvero su una singola macchina. consideriamo una chiamate in C tipo:

```
count = read(fd, buf, nbytes)
```

dove *fd* è un intero che indica un file, *buf* è un array di caratteri e *nbytes* è il numero intero di byte da leggere ed immagazzinare in *buf*.Se la chiamata è stata eseguita dal *main* allora lo *stack* della chiamata sarà quello mostrato in Figura 12. Come vediamo nella figura (b) il chiamante inserisce (*push*) i parametri nello stack in ordine inverso. Alla fine dell'esecuzione della procedura il valore di ritorno è posizionato in un registro vengono rimossi i parametri e viene restituito il controllo al chiamante.

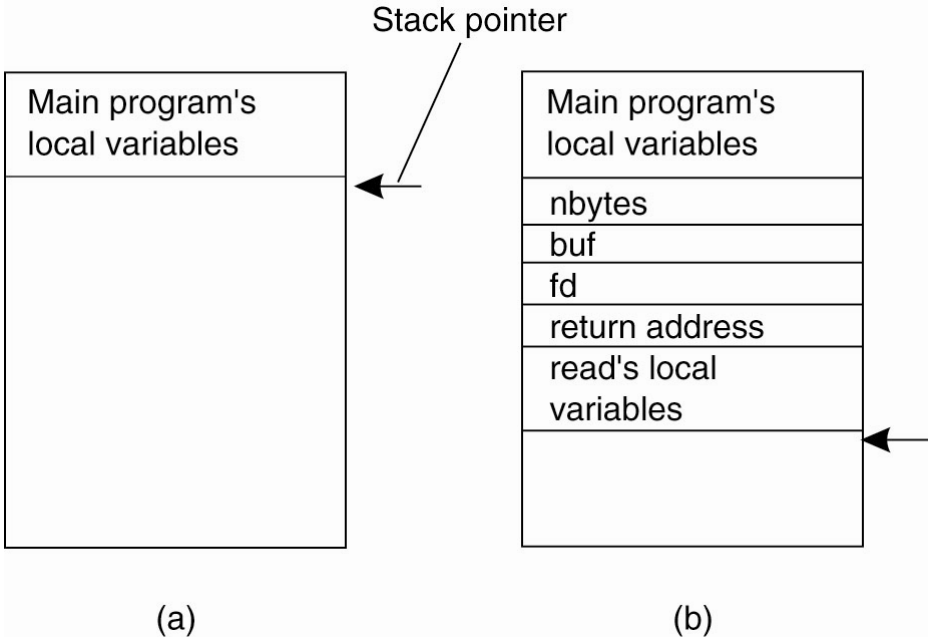

Figura 12: Stack prima e dopo la chiamata di una procedura

Notiamo ora che in C i parametri possono essere passati per **valore** come *fd* e *nbytes* per i quali il valore viene copiato nello stack o per **referenza** come nel caso di *buf* il quale è un puntatore ad un array di char; in questo caso nello stack della chiamata non vi è il valore dell'array ma semplicemente un indirizzo che indica dove l'array è situato. Nel caso in cui la procedura modifichi i valori contenuti nell'array tali cambiamenti saranno effettivi anche all'uscita dalla procedura.
Esiste un terzo meccanismo di passaggio dei parametri anche se non è usato in C ed è quello per **copia/ripristino**, questo passaggio consiste nel copiare il valore della variabile nello stack come nel caso di passaggio per valore, e quindi ricopiarla al termine della chiamata sovrascrivendo il valore originale.

Client e server stub L'idea di base delle RPC è quella di rendere le chiamate a procedura remote il più simile possibile ad una chiamata locale. Vogliamo che le RPC siano trasparenti alla distribuzione. Per ottenere tale trasparenza quando il linker assembla il codice al posto di mettere la versione di sistema della procedura, nel nostro caso la `read`, esso la sostituisce con una versione chiamata **client stub**. Come quella originale anche questa versione viene richiamata usando la sequenza di Figura 12, anche questa esegue una chiamata di sistema, ma a differenza di quella tradizionale questa chiamata impacchetta i dati in un messaggio e ne richiede l'invio al server tramite una `send`. Dopo l'invio il *client stub* richiama la procedura `receive` e si blocca in attesa della risposta come mostrato in Figura 13. Quando il messaggio raggiunge il server esso lo passa al **server sub**, l'equivalente lato server del *client stub*, questo pezzo di codice trasforma la RPC in una chiamata a procedura locale. Il server esegue il proprio lavoro e restituisce il risultato al chiamante. Quando il *server stub* riprende il controllo impacchetta il risultato in un messaggio e richiama la `send` per inviare la risposta al client e si rimette in attesa dell'arrivo di una nuova richiesta con la `receive`. Quando il messaggio di risposta arriva alla macchina client il sistema operativo lo indirizza al *client stub* che lo spacchetta, copia i dati nel buffer e restituisce il controllo al processo client.

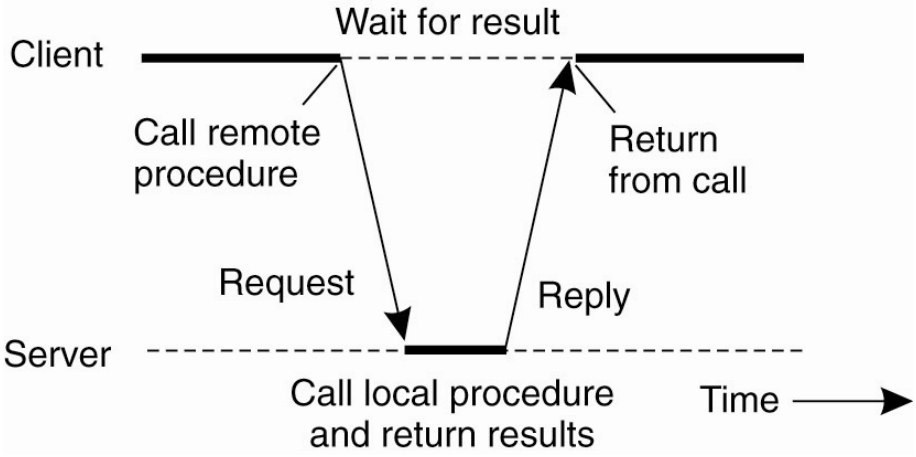

Figura 13: Esempio di chiamata a procedura remota

Quando il client riprende il controllo non ha idea di che cosa sia avvenuto e non ha idea se la chiamata è stata eseguita remotamente o in locale. Ricapitolando una chiamata a procedura remota segue i seguenti passi:

1. la procedura client richiama il *client stub* nel modo normale;
2. il *client stub* costruisce un messaggio e richiama il sistema operativo locale;
3. il SO del client invia il messaggio al SO remoto;
4. il SO remoto passa il messaggio al *server stub*;
5. il *server stub* spacchetta i parametri e richiama il server;
6. il server esegue il lavoro e restituisce il risultato allo *stub*;
7. il *server stub* lo impacchetta in un messaggio e richiama il su SO;
8. il SO del server invia il messaggio al SO del client;
9. il SO del client passa il messaggio al *client stub*;
10. lo *stub* spacchetta il risultato e lo restituisce al client.

5.2.2 Passaggio di parametri

La funzione del client stub è quella di prendere i propri parametri di impacchettarli e di inviarli al server stub. Il problema è che questa operazione pur sembrando molto semplice in realtà non lo è.

Passaggio di parametri per valore L'operazione di impacchettare i parametri in un messaggio è detta **marshaling dei parametri**. Come esempio consideriamo una procedura remota molto semplice come `add(i,j)` che prende in ingresso due valori interi i, j e restituisce la loro somma aritmetica. L'esecuzione di questa procedura è mostrata in Figura 15, nella parte sinistra vediamo la chiamata a tale procedura. Il *client stub* prende i due parametri e li mette in un messaggio semplicemente copiando i valori; insieme ai parametri viene anche inserito il nome

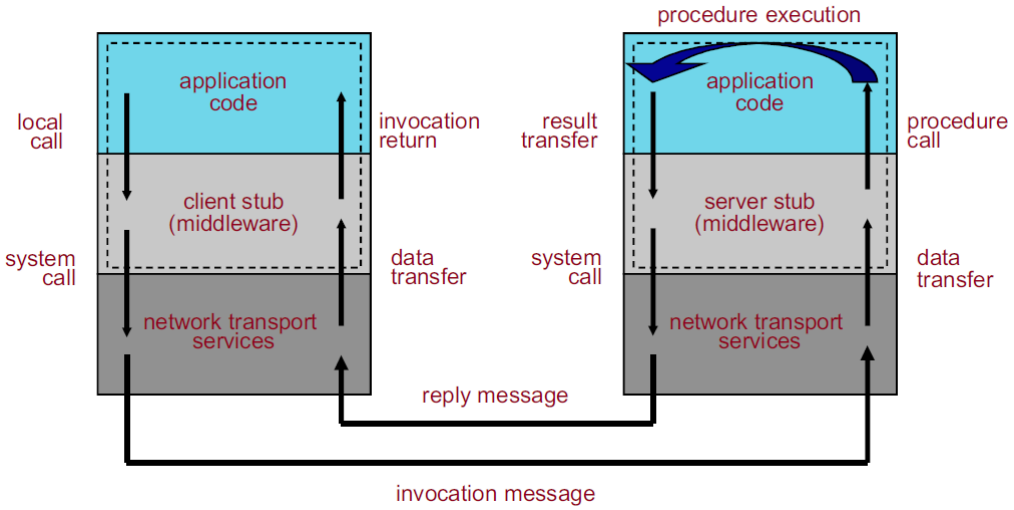

Figura 14: Esecuzione di una chiamata a procedura a procedura remota

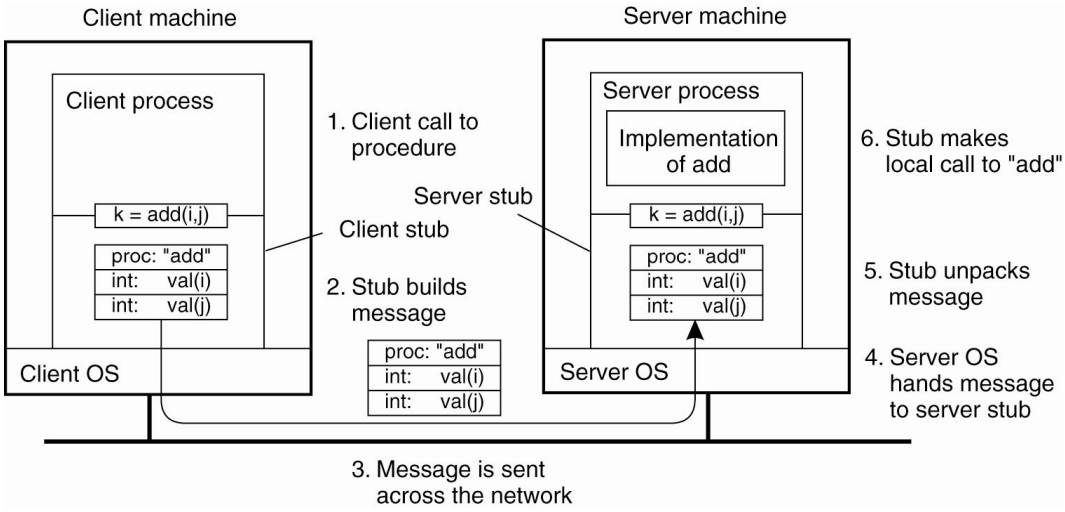

Figura 15: Passaggio di parametri ad una procedura remota

(o il numero) della procedura da chiamare in quanto il server potrebbe supportare diverse chiamate. Quando il messaggio raggiunge il server lo *stub* lo esamina per capire quale procedura è richiesta e quindi esegue la chiamata a tale procedura. La chiamata alla procedura è una normale chiamata locale l'unica differenza è che i parametri che vengono passati sono stati estratti dal messaggio e inizializzati nello spazio degli indirizzi dello *stub*.
Quando il server ha completato l'esecuzione il controllo ritorna al *server stub* il quale preleva il risultato e lo inserisce nel messaggio di ritorno al *client stub* il quale effettua *unmarshal* e restituisce il risultato al client.

I problemi iniziano a sorgere quando due macchine non sono dello stesso tipo, ad esempio i *mainframe IBM* usano una codifica *EBCDIC* per i caratteri mentre i personal computer usano l'*ASCII*; problemi simili possono verificarsi con la rappresentazione degli interi o dei numeri in virgola mobile, in quanto alcune macchine utilizzano la numerazione **little endian** (numerazione dei bit da destra a a sinistra) altre invece utilizzano la **big endian** (numerazione da sinistra a destra).

Passaggio di parametri per referenza Abbiamo visto il passaggio di parametri per valore, che a parte i problemi di rappresentazione non solleva grosse problematiche. Vediamo ora invece il passaggio di parametri tramite puntatori o riferimenti in generale.
Un puntatore ha senso solo nello spazio degli indirizzi del processo in cui è usato. Ad esempio nel caso della `read` visto in precedenza il secondo parametro è un puntatore ad un array di caratteri e contiene quindi l'indirizzo al primo elemento dell'array (ad esempio 1000). Se noi passassimo al server tale indirizzo non avrebbe alcun senso in quanto lato server l'indirizzo 1000 potrebbe essere occupato da una istruzione del programma. La strategia più semplice in questo caso è che il client stub, visto che conosce sia la tipologia che la dimensione dell'array, faccia una copia dell'array nel messaggio e lo invii al server stub il quale a questo punto può richiamare la procedura utilizzando come indirizzo quello dell'aria di memoria nella quale il server stub ha posizionato l'array appena giunto. Quando il server termina il server stub copia nuovamente l'array nel messaggio e lo restituisce al client. Quando il messaggio giunge al client stub esso applica le modifiche all'array originale. Il meccanismo di passaggio per riferimento è stato così sostituito da un meccanismo di **copia/ripristino**.
Esistono delle ottimizzazioni per rendere più efficace tale meccanismo, se gli *stub* conoscono se il parametro è un input oppure un output per il server una delle due copie può essere eliminata.
Abbiamo visto come passare un dato di cui conosciamo la dimensione, nel caso più generale di dati come strutture o oggetti i dati possono essere passati come uno stream di byte, questo meccanismo è chiamato *serializzazione*.

Specifica dei parametri e generazione degli stub Come abbiamo visto fino ad ora per nascondere una chiamata remota bisogna che il chiamante e il chiamato concordino sul formato dei messaggi e che eseguano gli stessi passi, in altre parole entrambi i lati di una RPC devono seguire lo stesso *protocollo*. Definire il formato dei messaggi non è però sufficiente è necessario anche che il client e il server concordino sulla rappresentazione delle strutture dati e su come i messaggi debbano essere scambiati, come ad esempio utilizzare un protocollo orientato alla connessione come il TCP/IP.
Una volta che il protocollo è stato definito bisogna implementare gli stub, fortunatamente questi differiscono soltanto per le loro interfacce verso le applicazioni . Per semplificare le cose le interfacce sono spesso definite tramite un **linguaggio per la definizione di interfacce** (IDL *interface definition language*). Un'interfaccia specificata tramite IDL viene successivamente compilata in un *client stub* e in un *server stub* unitamente alle interfacce *compile-time* o *run-time*.
Utilizzare un linguaggio per la definizione delle interfacce semplifica considerevolmente le applicazione client-server basate su RPC.

5.2.3 RPC in pratica

Abbiamo visto fino ad ora la teoria che sta dietro ad una chiamata a procedura remota, vediamo ora invece come viene implementato nella realtà un sistema basato sulle RPC.
Esistono due vie per produrre un sistema che sfrutta le chiamate a procedura remote, il primo è lo standard *de facto* ed è stato introdotto dalla Sun Microsystem. Questo sistema è parte del Network File System dei sistemi UNIX e specifica il formato dei dati tramite un XDR (*eXternal Data Representation*), il protocollo di trasporto utilizzato è indifferentemente il TCP o UDP, il passaggio di parametri è consentito solo tramite copia e con un massimo di un input ed un output per funzione, infine tale meccanismo fornisce dei meccanismi per la criptazione dei dati.
Il secondo meccanismo è quello proposto dall'Open Group ed è chiamato *Distributed Computing*

Environment (DCE) ed è un insieme di specifiche e riferimenti. Esistono diverse specifiche per la chiamata di una procedura:

- **At-Most-Once:** in cui una chiamata non viene portata avanti più di una volta
- **Idempotent:** in questo caso la procedura può essere ripetuta più di una volta senza problemi
- **Broadcast:** marcando una procedura in questo modo la richiesta sarà inoltrata a tutte le macchine della rete

Come primo passo per creare un'applicazione distribuita dopo aver scelto i meccanismi è quello di chiamare i programmi *uuidgen* i quali generano un prototipo di file IDL contente un identificatore d'interfaccia. A questo punto si completano i file IDL compilando i nomi delle procedure remote ed i loro parametri. Una volta completato il file IDL viene richiamato il compilatore IDL che genera un file *header* da includere nelle applicazione, il *client stub* e il *server stub*. A questo punto il programmatore deve sviluppare le due applicazioni client e server le quali verranno poi compilate e linkate insieme alle librerie e ai rispettivi client e server stub come mostrato in Figura 16 L'ultimo problema che dobbiamo affrontare è come *legare* un client al server che implementa la procedura richiesta dal client. Anche in questo caso Sun e DCE affrontano il problema in due modi differenti. Sun introduce un demone chiamato `portmap` che associa un server ad una porta, il server occupa una porta disponibile e comunica la sua scelta al *portmap* il client contatta il portmap e richiede la porta necessaria. Il problema principale è che il portmap risolve solo il problema di come stabilire la connessione al server ma deve già conoscere l'ubicazione del server.
DCE, invece utilizza due demoni, uno situato sulla stessa macchina del server che svolge la stessa funzione del portmap, ed un secondo demone situato su un server differente (*directory server*) il quale implementa la trasparenza alla distribuzione. Il meccanismo con il quale tale sistema opera è illustrato nella Figura 17.

5.2.4 Tipi di chiamate a procedure remote

Fino ad ora abbiamo dato per scontato che le chiamate a procedura remote fossero solamente di tipo sincrono con sincronizzazione alla fine dell'esecuzione della procedura remota. In alcuni casi, però, tale meccanismo non è efficiente in quanto spesso si sprecano risorse lato client. Una possibile variante sono le RPC asincrone le quali possono essere utilizzate quando non sono attesi valori di ritorno e lo sblocco del client avviene all'arrivo di un acknowledgment quando la richiesta viene presa in carico oppure all'arrivo di una *promessa* di risposta che verrà inviata tramite un'altra RPC asincrona dal server.
La Sun inoltre ha implementato delle RPC *batched* che sono particolari chiamate a procedura che non si aspettano un risultato, queste sono immagazzinate (*buffered*) dal client ed inviate in un unica comunicazione quando si presenta una chiamata non-batched.
Un esempio simile avviene nei dispositivi mobili; quando l'host è disconnesso immagazzina le richieste e periodicamente tenta l'invio di richieste, lo stesso viene fatto dal server per l'invio delle risposte. Le richieste e le risposte sono inviate/ricevute su due canali differenti.

5.2.5 Remote method invocation

Il *Remote Method Invocation* (RMI) è simile alle RPC ma è applicato alla programmazioni ad oggetti, un'importante differenza è che i riferimenti oggetti remoti possono essere passati. L'IDL

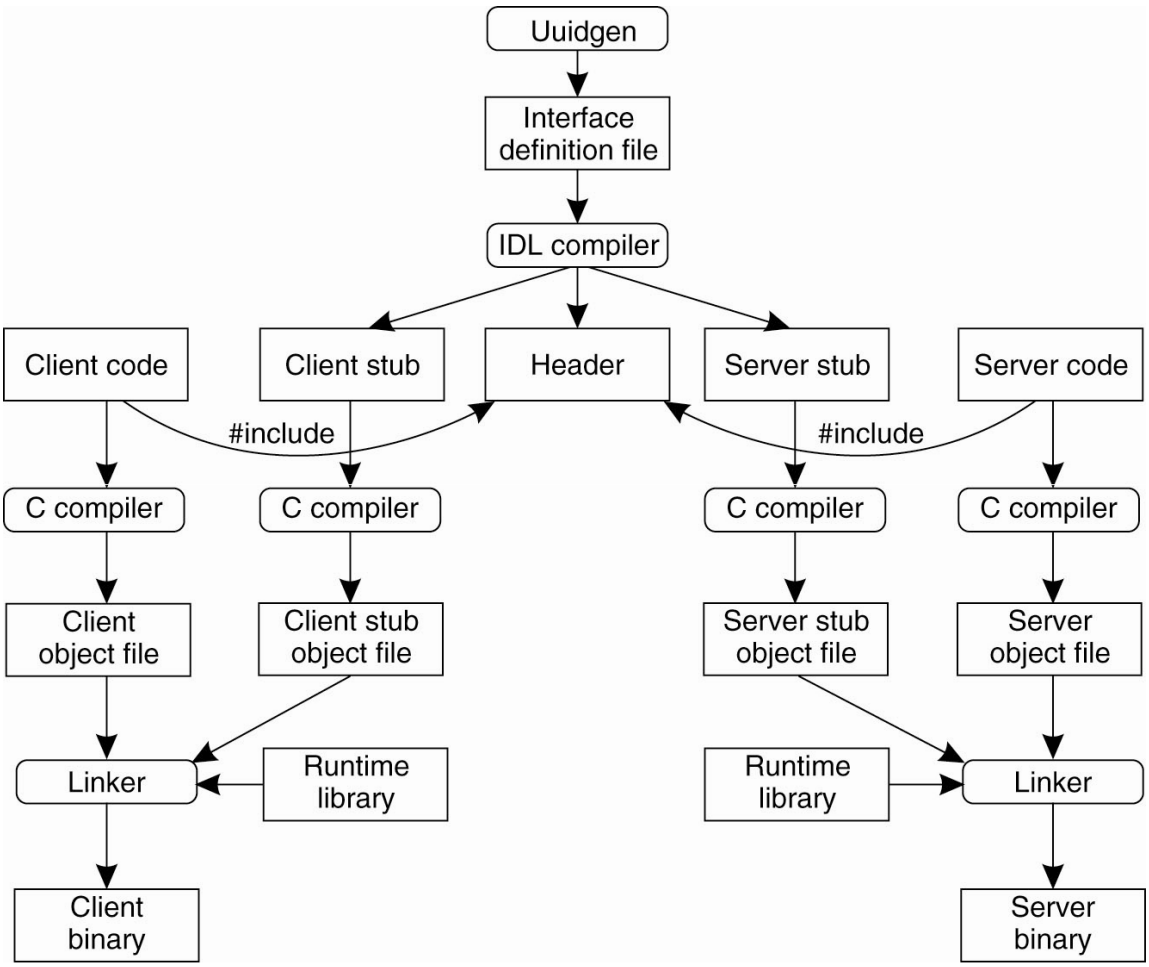

Figura 16: Flusso di sviluppo di un'applicazione distribuita

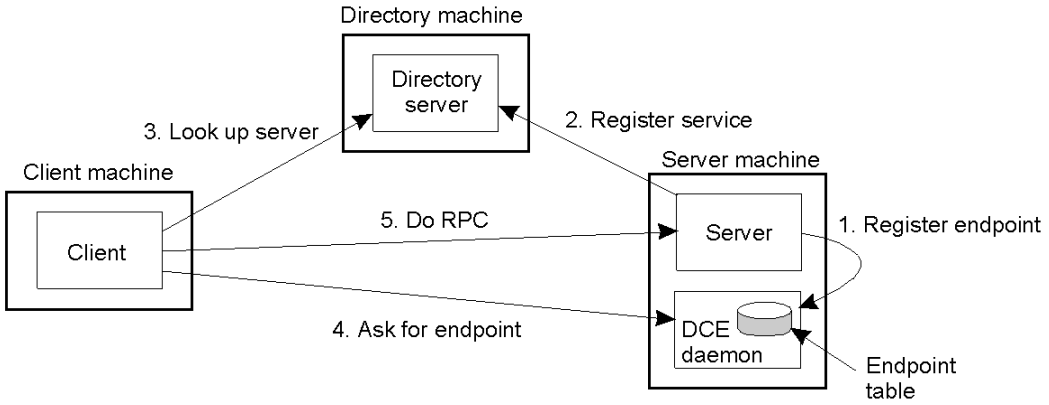

Figura 17: Esecuzioni di un binding tra client e server in DCE

nel caso di RMI è molto più completo includendo eccezioni ed altro in quanto la separazione tra interfaccia ed implementazione è alla base della programmazione ad oggetti.
Le più importanti tecnologie che implementano l'RMI sono *Java RMI* che può essere utilizzata solo implementando in Java ed utilizzando una Java Virtual Machine, è molto semplice e permette il passaggio di parametri sia per riferimento che per valore. *OMG CORBA* è multilinguaggio e multipiattaforma, anche *CORBA* permette il passaggio di parametri sia per riferimento che per valore ma nel secondo caso è compito del programmatore garantire la stessa semantica sia lato server che lato client.

5.3 Comunicazione orientata agli oggetti

Le chiamate a procedure remote o le RMI aiutano a nascondere la comunicazione nei sistemi distiribuiti, ovvero forniscono un certo grado di trasparenza all'accesso. Tuttavia esistono dei casi in cui questi meccanismi non sono appropriati, come ad esempio nel caso in cui non si è certi che il destinatario sia attivo nell'istante di invio di un messaggio.La soluzione a queste problematiche sono i *messaggi*.

5.4 Comunicazione orientata ai messaggi

Le RPC come abbiamo visto hanno di per se una natura sincrona ed è necessario che sia mittente che destinatario siano in esecuzione all'invocazione della procedura. In molti casi però questo non è possibile o non è conveniente perciò le applicazioni vengono sviluppate utilizzando l'invio di messaggi.

5.4.1 Comunicazione transiente orientata ai messaggi

Molti sistemi e applicazioni distribuite sono costruite direttamente in base al semplice modello orientato ai messaggi fornito dal livello di trasporto. Tali applicazioni sono sviluppate attraverso l'utilizzo delle *socket*.

Berkeley socket Un'attenzione particolare è stata messa nella standardizzazione dell'interfaccia del livello di trasporto in modo da consentire ai programmatori di far uso dell'intera suit

Primitiva	Significato
`Socket`	Crea una nuova porta di comunicazione
`Bind`	Collega un indirizzo locale ad una socket
`Listen`	Annuncia la disponibilità ad accettare connessioni
`Accept`	Blocca il chiamante finche la richiesta di connessione non arriva
`Connect`	Cerca attivamente di stabilire una connessione
`Send`	Invia dati sulla connessione
`Receive`	Riceve dati sulla connessione
`Close`	Rilascia la connessione

Tabella 1: Primitive delle socket

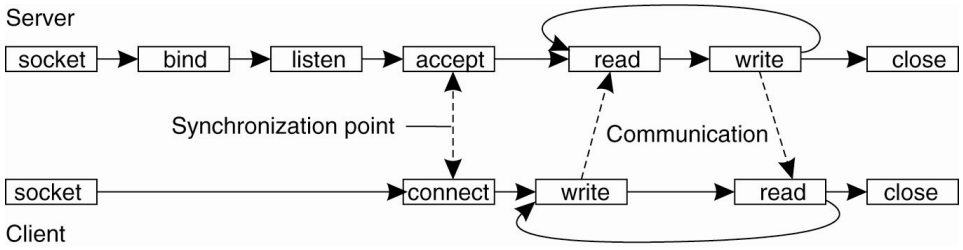

Figura 18: Schema generale di una connessione socket

di protocolli attraverso un semplice insieme di primitive. Una di queste interfacce è **l'interfaccia socket**introdotta in UNIX Berkeley, un'altra interfaccia importante è la **XTI (X/open transport interface)**. Le *socket* e XTI sono molto simili nel modello di programmazione ma differiscono nelle primitive.
Una socket è una porta di comunicazione su cui un'applicazione può scrivere dati da inviare e da cui poter leggere le risposte. Le primitive delle socket sono mostrate in Tabella 1 In generale i server eseguono le prime quattro primitive nell'ordine in cui sono presentate. Quando viene richiamata la primitiva `socket` il chiamante crea una nuova porta di comunicazone ed il sistema operativo locale riserva tale risorsa per la ricezione e l'invio di dati. La primitiva `bind` associa un indirizzo locale con la socket appena creata, il collegamento (*binding*) indica al sistema operativo che il programma vuole ricevere messaggi solo all'indirizzo e sulla porta specificati. La primitiva `listen` viene richiamata solo nel caso di comunicazione orientata alla connessione, serve per comunicare al SO quante risorse riservare in base al numero di connessioni massime che il server intende accettare. Tramite la primitiva `accept` il server si blocca fino all'arrivo di una richiesta di connessione, al suo arrivo il sistema operativo locale crea una nuova *socket* con le stesse proprietà dell'originale e la restituisce al chiamate. Questo meccanismo permette al server di effettuare una *fork* e gestire una comunicazione mentre attende una nuova connessione. Lato client il procedimento è leggermente differente, anche qui è necessario creare la risorsa tramite la primitiva `socket` ma il *binding* esplicito di tale risorsa non è necessario. La primitiva `connect` richiede che il chiamante specifichi l'indirizzo a cui va inviata la richiesta di connessione; il client è bloccato fino a quando la connessione non è stata stabilita dopo di che entrambi i lati possono iniziare a scambiarsi messaggi tramite le primitive `send` e `receive`. Nelle socket la connessione è simmetrica perciò la chiusura della connessione si ha quando sia il server che il client richiamano la primitiva `close`. Il modello generale di una connessione socket è mostrato in Figura 18.

Primitiva	Significato
MPI_bsend	Aggiunge un messaggio in uscita a un buffer per l'invio locale
MPI_send	Invia un messaggio e aspetta finché non viene copiato in un buffer
MPI_ssend	Invia un messaggio e aspetta finché non inizia la ricezione
MPI_sendrecv	Invia un messaggio e aspetta la risposta
MPI_isend	Invia il riferimento a un messaggio in uscita e continua
MPI_issend	Invia il riferimento a un messaggio e aspetta finché non inizia la ricezione
MPI_recv	Riceve un messaggio; si blocca se non ce ne sono
MPI_irecv	Controlla se ci sono messaggi in ingresso, ma non si blocca

Tabella 2: Elenco delle primitive MPI

Interfaccia per lo scambio di messaggi Con l'arrivo dei computer ad alte prestazioni gli sviluppatori hanno avuto bisogno di primitive orientate ai messaggi che consentissero la scrittura di applicazioni in modo facile ed efficiente. Queste caratteristiche fanno si che le primitive siano ad un livello di astrazione sufficentemente alto per sviluppare le applicazioni in modo veloce ma che richiedessero un incremento minimo rispetto alle *socket*, le quali non potevano essere sfruttate in quanto avevano un livello di astrazione troppo basso, ed erano state pensate solo per la comunicazione in rete tramite TCP/IP.
Serviva qualcosa che di adatto a reti ad alta velocità come quelle usate nei *cluster* ed inoltre serviva un'interfaccia più avanzata che potesse gestire funzionalità più avanzate come diverse forme di *buffering* e di sincronizzazione.
Si è arrivati così alla definizione di uno standard chiamato semplicemente **interfaccia per lo scambio di messaggi** o **MPI** (*message-passing interface*). Nato come progetto per le applicazioni parallele e come tale adatto alla comunicazione transiente, utilizza la rete sottostante e presuppone che eventi come la caduta di un processo siano fatali.
L'MPI suppone che la comunicazione avvenga tra un gruppo di processi conosciuti, a cui viene assegnato un identificatore del tipo (*groupID,processID*) che identifica univocamente un processo e che viene utilizzato in sostituzione dell'indirizzo del livello di trasporto.
Il cuore di MPI è costituito da primitive per lo scambio di messaggi per supportare la comunicazione transiente, esse sono elencate nella Tabella 2 LA comunicazione transiente asincrona è supportata dalla primitiva `MPI_bsend`, il mittente invia un messaggio che viene copiato localmente dal sistema runtime di MPI, quando il messaggio è stato copiato il mittente continua il proprio lavoro. Il sistema runtime locale cancellerà il messaggio dal suo buffer e si occuperà della trasmissione non appena un destinatario chiamerà una `receive`.
Esiste anche un'operazione di invio bloccante, la `MPI_send` la quale blocca il chiamante fino a quando il messaggio non è stato copiato nel buffer runtime del destinatario oppure fino a quando il destinatario non ha iniziato la ricezione. La comunicazione sincrona nella quale il mittente si blocca fino a quando la sua richiesta non è accettata avviene tramite la `MPI_ssend`, infine, è disponibile anche una comunicazione fortemente sincrona nella quale il sistema attende fino all'esecuzione della richieste e ricezione della relativa risposta che avviene tramite la primitiva `MPI_sendrecv`; quest'ultima corrisponde ad una normale RPC.
Le due primitive `MPI_send` e `MPI_ssend` hanno delle varianti che evitano di copiare il messaggio nel buffer locale, queste varianti corrispondono ad un tipo di comunicazione asincrona e si ottengono tramite la `MPI_isend` tramite il quale il mittente invia un puntatore ad un messaggio dopodiché il sitema runtime di MPI si occupa della comunicazione mentre il mittente prosegue con la sua esecuzione. La primitiva `MPI_issend` fornisce la sicurezza che il destinatario abbia accettato il messaggio e stia lavorando sulla richiesta.

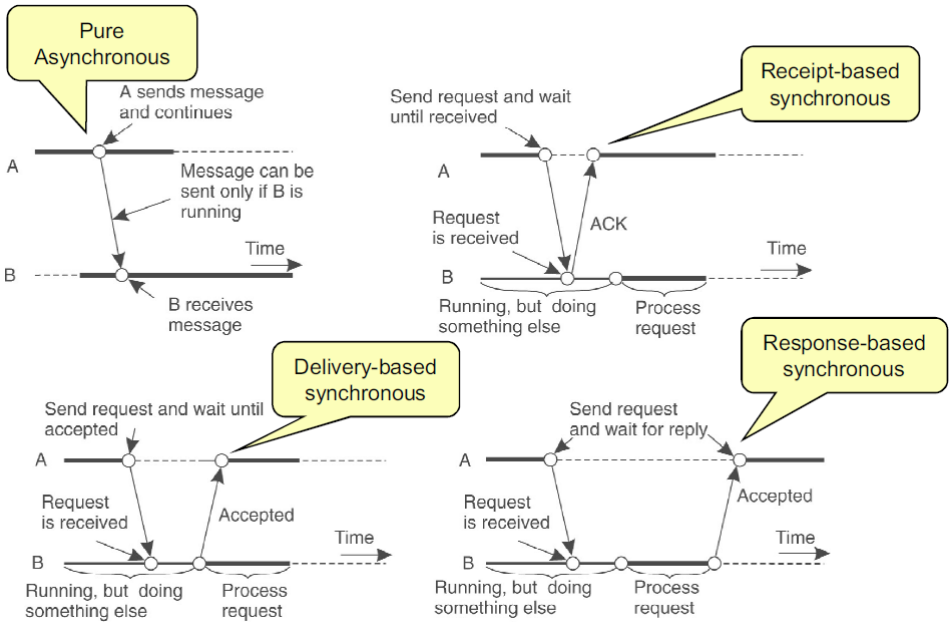

Figura 19: Esempi di comunicazione di tipo transiente

Le operazioni `MPI_recv` e `MPI_irecv` sono rispettivamente le primitive bloccanti e non bloccanti usate per la ricezione dei messaggi.
I diversi tipi di comunicazione sono mostrati in Figura 19.

5.4.2 Comunicazione persistente orientata ai messaggi

La classe più importante di servizi middleware orientati ai messaggiè quella conosciuta come **sistemi a code di messaggi** o semplicemente **middleware orientati ai messaggi** (MOM, *message oriented middleware*). Questo tipo di sistemi forniscono un ampio supporto alla comunicazione asincrona persistente. La caratteristica fondamentale di questi sistemi è che permettono di memorizzare i messaggi senza che il mittente e il destinatario siano attivi durante la trasmissione del messaggio.

Modello a code per lo scambio di messaggi L'idea di base è che le applicazioni comunicano inserendo i messaggi in code specifiche, questi messaggi vengono poi inoltrati tramite una serie di serve e alla fine consegnati al destinatario.
La particolarità è che un mittente non ha la garanzia che un messaggio venga effettivamente letto ma ha solo la garanzia che tale messaggio sarà inserito prima o poi nella coda dei messaggi del destinatario.
Questo aspetto fa si che vi sia un forte disaccoppiamento nelle comunicazioni e non vi è quindi la necessità che il destinatario sia in esecuzione quando viene inviato il messaggio, analogamente non vi è la necessità che il mittente sia in esecuzione quando il messaggio viene prelevato dal destinatario come mostrato in Figura 20. Questo significa che un messaggio rimarrà in coda fino a quando non verrà rimosso senza tenere conto se il mittente o il destinatario sono in esecuzione.
I tipi di messaggi che possono essere inviati sono infiniti, l'unica restrizione è che essi siano opportunamente indirizzati; per l'indirizzamento viene fatto fornendo un nome univoco a livello del sistema della coda di destinazione. La semplicità di questa architettura si rispecchia anche

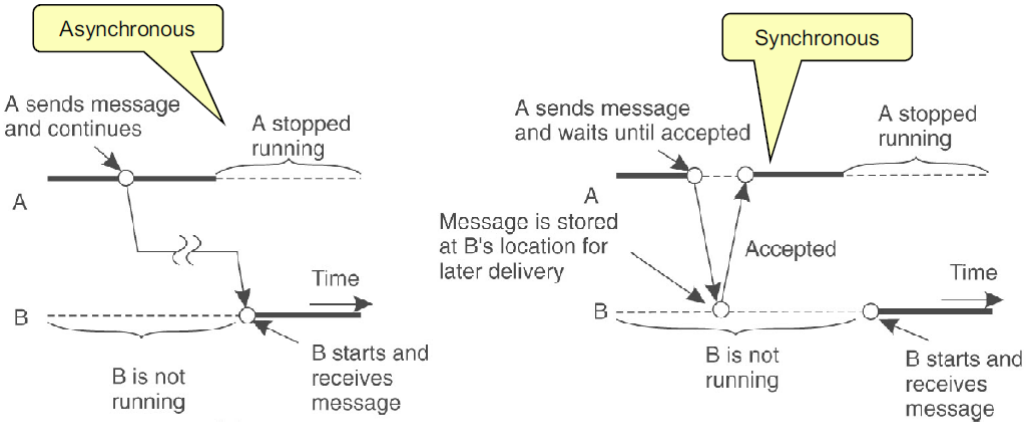

Figura 20: Esempio di comunicazione persistente orientata ai messaggi

Primitiva	**Significato**
`Put`	Aggiunge un messaggio ad una coda specifica
`Get`	Preleva il primo messaggio in coda nel caso sia vuota si blocca
`Poll`	Preleva il primo messaggio in coda ma non è bloccante
`Notify`	Installa un handler da chiamare quando arriva un messaggio nella coda

Tabella 3: Interfaccia di base per una coda in un sistema a code di messaggi

nella sua interfaccia che è mostrata in Tabella 3. La primitiva `put` viene richiamata da un mittente per passare un messaggio al sistema sottostante e posizionarlo in una coda specifica. La `get` è una chiamata bloccante attraverso la quale un processo autorizzato può rimuovere dalla coda specificata il messaggio pendente da più tempo; nel caso la coda sia vuota il processo viene bloccato. La corrispettiva chiamata non bloccante è la `pull`. Molti sistemi inoltre permettono l'installazione di un *handler* sotto forma di *funzione callback* che viene richiamata all'arrivo di ogni nuovo messaggio; molto utili nel caso si volesse avviare un processo per la ricezione dei messaggi.

Architettura generale di un sistema a code In realtà la vera architettura di un sistema a code di messaggi ha alcune restrizioni; prima fra tutti è che i messaggi possono essere inseriti solo in code *locali* al mittente ovvero code sulla stessa macchina o comunque sulla stessa LAN raggiungibile in modo *efficiente* tramite una RPC