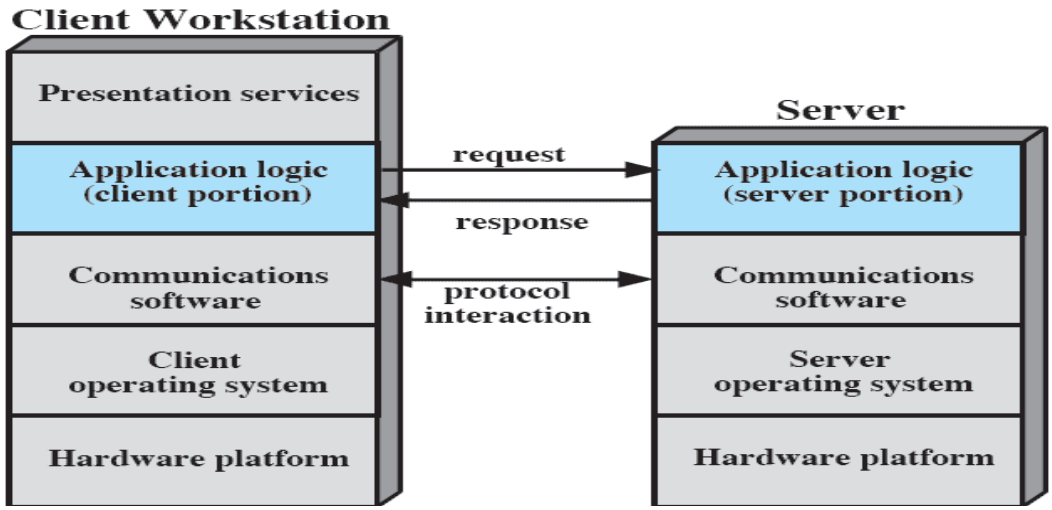
CLIENT/SERVER COMPUTING

I client di solito sono single-user PC o workstation che forniscono un'interfaccia decisamente "user-friendly" all'utente finale.
Ogni server fornisce una serie di servizi condivisi ai client.
Il server abilita molti client all'accesso a database condivisi e abilita l'uso si sistemi di computer ad alte prestazioni per gestire questo database di informazioni.
La configurazione client/server differisce dagli altri tipi di processamenti distribuiti:

- C'è un forte interesse nel portare applicazioni user-friendly sui sistemi degli utenti.
- C'è un'enfasi nella centralizzazione dei database aziendali e su molte funzioni di gestione e di utilizzo di rete.
- C'è impegno sia da parte di organizzazioni di utenti che dai venditori ad aprire e modulare sistemi
- Il networking è fondamentale per operare.

Inoltre nel client ritroviamo la logica dell'applicazione (cioè come è fatta) e ovviamente abbiamo una grande cura della presentazione, ma la maggior parte del software dell'applicazione viene svolta sui server.



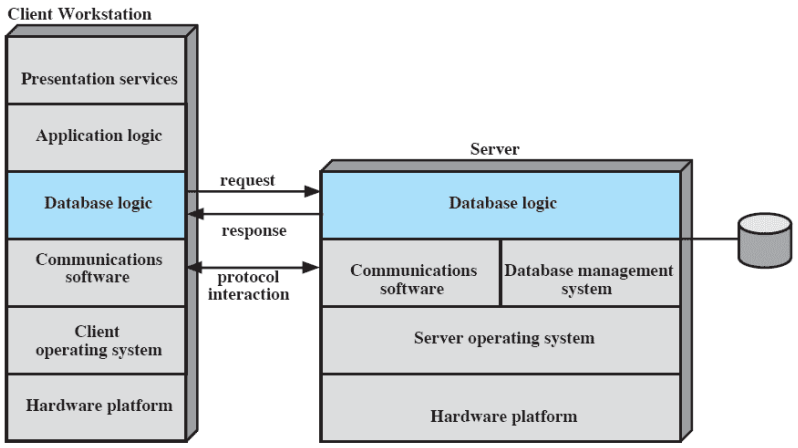
La chiave di una applicazione client/server è che i task dell'applicazione vengono svolti da entrambi i sistemi, mentre il loro sistema operativo o l'hardware potrebbe essere differente, ma poco importa poiché a livello di comunicazione usano lo stesso protocollo.
E' infatti il software di comunicazione che permette di far cooperare client e server (Es.TCP/IP)
Le funzione effettive di una applicazione possono essere quindi divise tra client e server in modo tale da ottimizzare l'uso di risorse.
NB il design di ogni applicazione deve essere facile da usare, da imparare, potente e flessibile in modo tale che l'esperienza di utilizzo sia alla portata di tutti.

COMPONENTI DI APPLICAZIONI DISTRIBUITE
Le applicazioni orientate al business sono composte di 4 principali componenti:

- Logica di presentazione: GUI.
- Logica di I/O: inserimento di dati da parte dell'utente.
- Logica di business: servizi e calcoli improntati a un dato business
- Logica di storage di dati: conservazione dati, richiesta dati, integrità dei dati.

Ovviamente per sapere come implementare queste funzionalità bisogna prima rispondere ad alcune domande che sorgono in base all'esigenza di tale business application.

Es. Database Applications
Il server è un database server.
Le interazioni tra client e server si basano su transazioni: il client richiede informazioni che riceve dal server.
Il server è responsabile di mantenere il database e gestirlo.



CLASSI DI CLIENT/SERVER APPLICATIONS
Abbiamo 4 classi generali:

1. Elaborazione basata su host
2. Elaborazione basata su server
3. Elaborazione basata sulla cooperazione
4. Elaborazione basata sul client

Le prime non sono vere e proprie applicazioni client/server, hanno più un aspetto tradizionale

(a) Host-based processing

Le seconde prevedono l'elaborazione totale dei dati da parte del server, mentre il client offre solo una logica di presentazione.

(b) Server-based processing

Le terze offrono un'elaborazione ottimizzata tra client e server, ma molto più complessa da implementare e mantere

(c) Cooperative processing

Le ultime invece prevedono l'elaborazione dei dati da parte del client, mentre le convalide e lo storage dei dati viene effettuato dal server.

(d) Client-based processing

L'architettura client/server è a tre livelli:

- User machine ossia il client.
- Un middle-tier server ossia un server intermedio che converte protocolli, fa il merge di differenti dati e altro.
- Backend server ossia lo storage vero e proprio

DEFINIZIONE DI SISTEMI DISTRIBUITI

Un sistema distribuito è un insieme di entità spazialmente separate, ognuna delle quali con un certo potere computazionale, capaci di comunicare e sincronizzarsi tra loro in funzione di un comune obiettivo sembrando ai loro utenti di essere un unico sistema coerente.

Ci sono moltissimi vantaggi che ci portano a scegliere spesso i sistemi distribuiti, cioè una collezione di computer indipendenti che appare come un unico sistema.

Es. network of workstations.

Vantaggi rispetto a sistemi centralizzati:

- Economici: una serie di microprocessori offre un miglior rapporto qualità/prezzo dei mainframes. Prezzi minori potenza di calcolo maggiore.
- Velocità: un sistema distribuito può avere una potenza maggiore di calcolo dei mainframes poiché più microprocessori ad una velocità bassa concorrono ad uno stesso risultato. Molto più ergonomico di un unico processore la cui potenza di calcolo equivalga a quella totale.
- Distribuzione intrinseca: molte applicazioni hanno intrinsecamente una organizzazione distribuita. Es. Catena di supermercati.
- Affidabilità: Se una macchina crasha, il sistema sopravvive.
- Crescita incrementale: Il sistema è modularmente espandibile potendo così aumentare la potenza di calcolo con piccole modifiche.
- Un altro vantaggio: l'esistenza di un grande numero di PC, la necessità delle persone di collaborare e scambiare informazioni

Vantaggi rispetto a PC indipendenti:

- Condivisione di dati: è permesso a molti utenti di accedere a dati condivisi.
- Condivisione di risorse: Es. costose periferiche come stampanti.
- Comunicazione: accresce la comunicazione tra uomini con mail, chat.
- Flessibilità: Distribuisce il lavoro da fare su le macchine disponibili.

Svantaggi dei sistemi distribuiti:

- Software: è più difficile sviluppare software per sistemi distribuiti.
- Network: problemi come saturazione o perdita di pacchetti.
- Sicurezza: è più facile accedere a dati protetti.

Il primo obiettivo è la condivisione di dati e risorse. Ci sono due problemi da affrontare: la sincronizzazione e la coordinazione. Purtroppo ci sono delle caratteristiche che si differenziano da un sistema centralizzato che ci limitano nel risolvere alcuni problemi di coordinazione come: concorrenza temporale e spaziale, mancanza di un orologio globale, fallimenti, latenze imprevedibili.

Problemi di progettazione dei sistemi distribuiti:

1. Trasparenza
2. Flessibilità
3. Affidabilità
4. Performance
5. Scalabilità

TRASPARENZA

Come far sembrare all'utente una serie di computer come un singolo computer. Questo obiettivo può essere raggiunto a due livelli:

1. Nascondendo la distribuzione all'utente.
2. O ad un livello più basso, fare si che il sistema sia trasparente ai programmi.

Ci sono diversi tipi di trasparenza:

- Trasparenza della posizione: gli utenti non possono sapere dove fisicamente si trovano le risorse.
- Trasparenza della migrazione: le risorse devono poter essere spostate in tranquillità senza che i loro nomi cambino.
- Trasparenza di replica: l'OS può effettuare copie aggiuntive di file e risorse senza che l'utente se ne accorga.
- Trasparenza di concorrenza: gli utenti non sono a conoscenza che altri utenti usino il sistema. E' quindi necessario gestire la mutua esclusione alle risorse.
- Trasparenza del parallelismo: uso automatico del parallelismo senza dover programmare esplicitamente in quella direzione. (IL SANTO GRAAL)

NB gli utenti non sempre vogliono completa trasparenza.

FLESSIBILITA'

Rende più facile fare modifiche. Il Kernel Monolitico esegue tutte le syscall che vengono catturate dal sistema. Il Microkernel invece fornisce servizi minori come la gestione della memoria, la gestione e lo scheduling di quale processo a basso livello e qualche dispositivo I/O di basso livello.

AFFIDABILITA'

I sistemi distribuiti devono essere più affidabili di sistemi singoli. Bisogni che ci sia sicurezza nell'utilizzo del sistema e soprattutto deve essere protetto da guasti o errori.

PERFORMANCE

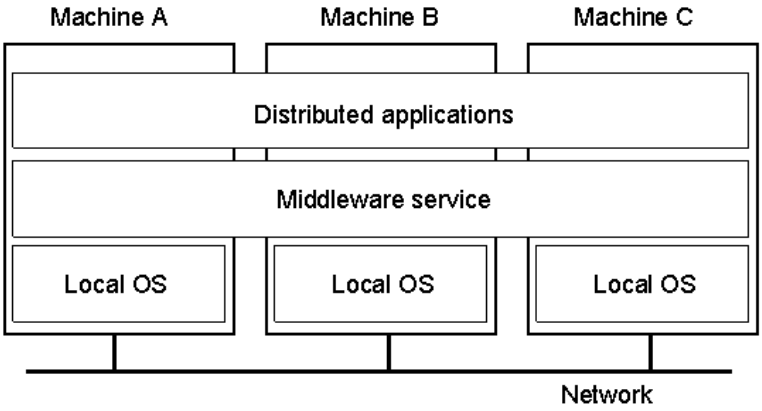
Se non ci si interessasse delle performance non avrebbe senso parlare di sistemi distribuiti.
Le performance perdono a causa della comunicazione che a volte potrebbe portare dei ritardi.
Inoltre rendere il sistema robusto ai guasti e agli errori ovviamente fa perdere di prestazioni.

SCALABILITA'

La scalabilità denota in genere la capacità di un sistema di aumentare o diminuire di scala in funzione delle necessità e disponibilità. In questo caso un aumento di potenza di calcolo deve poter migliorare tutto il sistema. Altrimenti se esiste uno o più colli di bottiglia quella miglioria potrebbe essere ininfluente.

Esempi di sistemi distribuiti sono Internet/World-Wide Web, una LAN o un sistema Peer-to-peer (P2P).
Le interazioni possono essere di tipo Client/Server ossia il client invoca il server e quest'ultimo risponde, oppure peer-to-peer dove ogni sistema è pari ad un altro e ognuno può chiedere e rispondere in funzione di un servizio. Es. Download torrent, in download richiedo un file in upload partecipo all'invio.

MIDDLEWARE



Un sistema distribuito è organizzato come un middleware, il quale estende tutte le macchine che sono comprese nel sistema distribuito.

Problemi da affrontare:

- Eterogeneità: Os, velocità di clock, rappresentazione dei dati, memoria e HW.
- Asincronia locale: differente Hardware, interrupts.
- Mancanza di conoscenza globale: la conoscenza si propaga attraverso messaggi il cui tempo di propagazione è molto più lento dell'esecuzione di un evento interno.
- Errori di nodi o partizioni di rete.
- Mancanza di un globale ordine degli eventi.
- Coerenza vs disponibilità vs partizioni di rete.

BAKERY ALGORITHM

L'algoritmo di Dijkstra garantisce mutua esclusione, evita il deadlock, non garantisce però l'evitarsi dello starvation. Inoltre si ha bisogno di operazioni di read/write atomiche e una memoria condivisa per il parametro k (turno).
Introduciamo ora l'algoritmo del panettiere.
Si pensi ad un bancone affollato di gente che vuol essere servita. Nel caso in cui non ci fosse nessuno in negozio il ticket è superfluo, altrimenti è il ticket a dettare l'ordine in cui la gente verrà servita.

Primo tentativo

```
while (1){
        /*NCS*/
        number[i] = 1 + max {number[j] | (1 <= j <= N) except i} //dorway
        for j in 1 .. N except i {
                while (number[j] != 0 && number[j] < number[i]); //bakery
        }
        /*CS*/
        number[i] = 0;
}
```

Qui il problema è che l'assegnazione del ticket a ogni processo non è descritta in maniera atomica. Questa condizione potrebbe assegnare un ticket uguali a due processi diversi.
In questo casi il risultato sarebbe che due processi entrano in CS, NO MUTUAL EXCLUSION.
Es P1 ha 1 e P2 ha 2, quando vanno a verificare la condizione del while verrà false poiché 1 < 1 false.
Se sostituisco al < il <= otterei una condizione probabile di deadlock perché entrambi aspetterebbero l'altro poiché 1 <= 1 true.

Secondo tentativo

```
while (1){
        /*NCS*/
        number[i] = 1 + max {number[j] | (1 <= j <= N) except i}
        for j in 1 .. N except i {
                while (number[j] != 0 && (number[j],j) < (number[i],i));
        }
        /*CS*/
        number[i] = 0;
}
```

La scrittura (number[j],j) < (number[i],i) significa
(number[j] < number[i] || (number[j] == number[i] && j < i))

Anche qui è possibile che due processi abbiano lo stesso ticket, la condizione nel while tratta quel caso ma questo tentativo non offre la mutua esclusione poiché potrei effettuare il for di controllo prima che ogni processo abbia preso il ticket. Inserirò quindi un array booleano per evitare ciò.

Quello finale

```
while (1){
        /*NCS*/
         choosing[i] = true;
         number[i] = 1 + max {number[j] | (1 <= j <= N) except i}
        choosing[i] = false;
        for j in 1 .. N except i {
                while (choosing[i] == true);
                while (number[j] != 0 && (number[j],j) < (number[i],i));
         }
        /*CS*/
         number[i] = 0;
}
```

Questo è l'algoritmo definitivo dove l'array choosing permette a ogni processo di entrare nella dorway in modo tale che gli sia assegnato un numero valido (anche non univoco). Un processo aspetterà finchè il valore dell'array choosing sarà false.

Caratteristiche dell'algoritmo:

- I processi comunicano attraverso variabili condivise come nell'algoritmo di Dijkstra.
- Read/write non sono operazioni atomiche.
- Tutte le variabili condivise sono di proprietà di un processo che può scriverci, gli altri possono leggerla.
- Nessun processo può eseguire due scritture simultanee
- I tempi di esecuzione non sono correlati

BAKERY ALGORITHM IN CLIENT/SERVER APP

```
while (1){ //client thread
        /*NCS*/
        choosing = true; //doorway
        for j in 1 .. N except i {
                send(Pj,num);
                receive(Pj,v);
                num = max(num,v);
        }
        num = num+1;
        choosing = false;
```

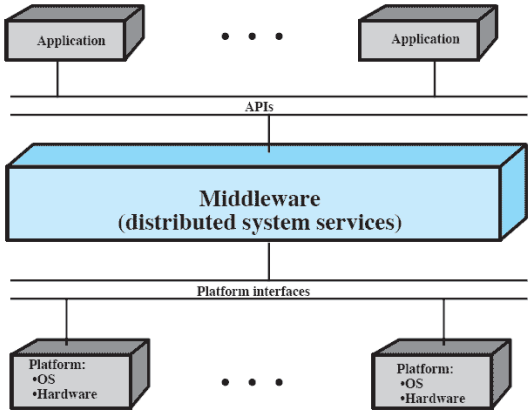
```
        for j in 1 .. N except i {  //backery
        do{
                send(Pj,choosing);
                receive(Pj,v);
        }while (v == true);
        do{
                send(Pj,v);
                receive(Pj,v);
        }while (v != 0 || (v,j) < (num,i));
        }
        /*CS*/
        num = 0;
}

while (1){ //server thread
        receive(Pj,message);
         if (message is a number) send(Pj,num);
         else send(Pj,choosing);
 }
```

MIDDLEWARE

Per ottenere i veri benifici dell'approccio client/server su sistemi distribuiti gli sviluppatori devono avere una serie di strumenti che permettono loro di avere un mezzo e uno stile di accesso uniforme alle risorse di sistema su tutte le piattaforme.

Ciò consentirebbe ai programmatori di costruire applicazioni basate su un sistema che sembra lo stesso per tutti. Permette ad esempio di accedere ai dati dovunque essi siano localizzati con un metodo univoco.

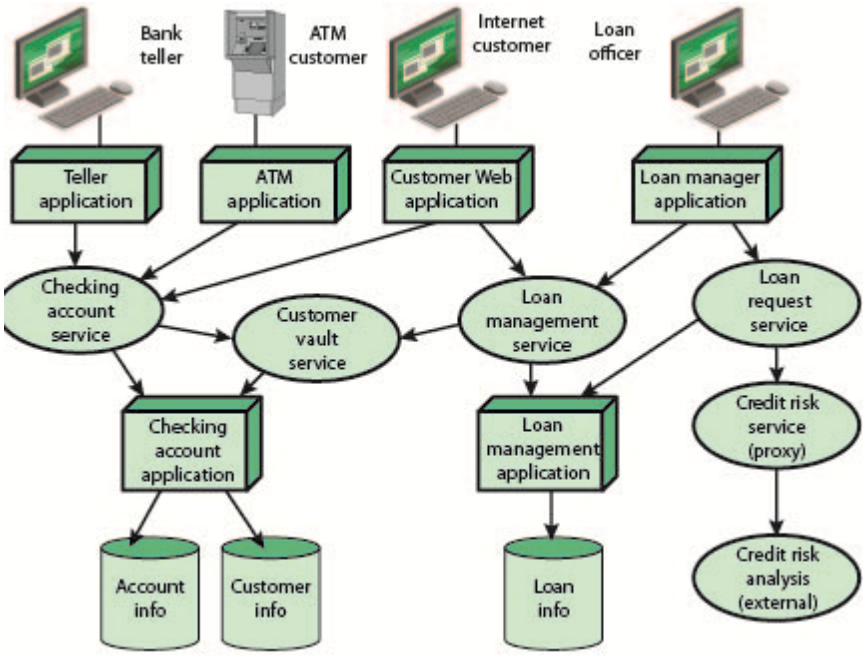
Per fare ciò bisogna utilizzare interfacce di programmazione standardizzate e protocolli che si interpongono tra l'applicazione e il sistema operativo. L'insieme delle interfacce e dei protocolli prende il nome di middleware.



SERVICE-ORIENTED ARCHITECTURE (SOA)

Una forma di architettura client/server utilizzata nei sistemi aziendali.

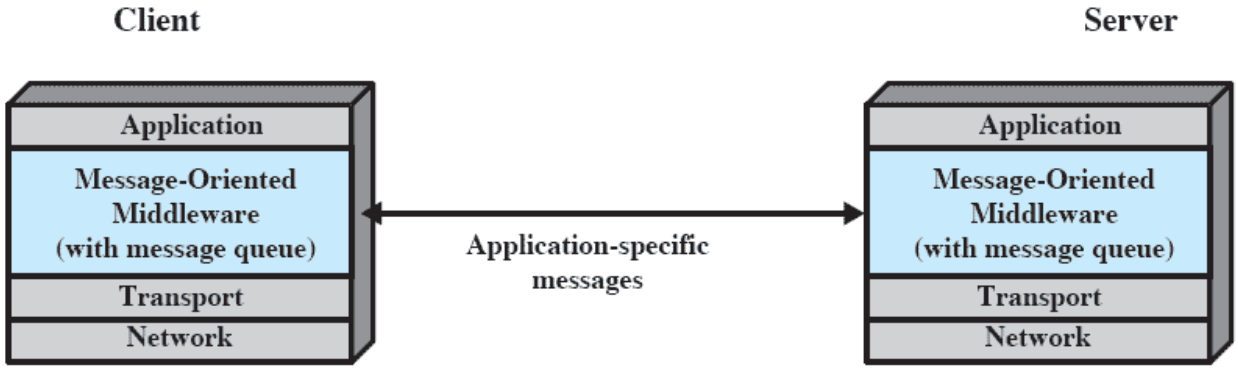
Organizza le funzioni aziendali in una struttura modulare invece che applicazioni monolitiche per ciascun dipartimento. Di conseguenza le funzioni comuni possono essere utilizzate da diversi reparti interni e anche da partner commerciali esterni. E' costituito da un insieme di servizi e un insieme di app client che ne fanno uso. Le interfacce standardizzate sono usate per permettere la comunicazione tra i moduli di servizio e tra questi ultimi e le applicazioni.



Ogni dispositivo ha la propria applicazione che può accedere a determinati servizi, anche in comune utilizzando un interfaccia standardizzata e attraverso un Service Broker (mediatore).

MESSAGGE PASSING NEI SISTEMI DISTRIBUITI

Nei sistemi distribuiti non esiste una memoria condivisa, bisogna utilizzare tecniche basate sul message passing.



Una comunicazione molto semplice: solo due primitive sono richieste, Send e Receive. Il processo che invia usa il modulo del message-passing standardizzato che sarà quello che tradurrà la richiesta al processo ricevente. Attraverso quindi il servizio messo a disposizioni del sistema intero dal modulo del message passing si ha un implementazione del protocollo di comunicazione.

AFFIDABILITA' VS INAFFIDABILITA'

La struttura del message-passing può essere "affidabile" ossia utilizza protocolli che prevedono il check dell'errore, ACK, la ritrasmissione e il riordino di messaggi spaiati. Siccome la spedizione è garantita non è necessario far sapere al processo inviante che la spedizione è andata a buon fine, comunque sia non è affatto sbagliato far tornare un ACK di conferma. Nel caso in cui, network down per una serie di motivi, la spedizione non dovesse andare a buon fine il processo inviante sarebbe avvisato di tutto.

All'altro estremo invece potremmo avere una struttura "inaffidabile" ossia il messaggio potrebbe essere spedito tranquillamente senza interesse nel sapere una condizione di successo o meno. Questa alternativa riduce enormemente sia la complessità che il processamento della spedizione e in più farebbe diminuire l'overhead del messaggio. Per quelle applicazioni che necessitano di conferma di spedizione, si potrebbe inviare una richiesta e rispondere ad essa con stesso un messaggio.

BLOCKING VS NONBLOCKING

Con un approccio nonblocking, un processo non è bloccato nel momento in cui fa uso di primitive di comunicazione (Send e receive). I processi fanno un uso efficiente e flessibile della struttura del message passing. E' più difficile testare e debuggare programmi che usano primitive nonblocking poiché le sequenze di invio possono creare problemi difficili e sottili.

Con un approccio blocking invece un processo che invia viene bloccato finchè il messaggio non è stato inviato ed è stata ricevuta la conferma di spedizione e in ricezione il processo è bloccato finchè il messaggio non è stato conservato in un buffer.

REMOTE PROCEDURE CALLS

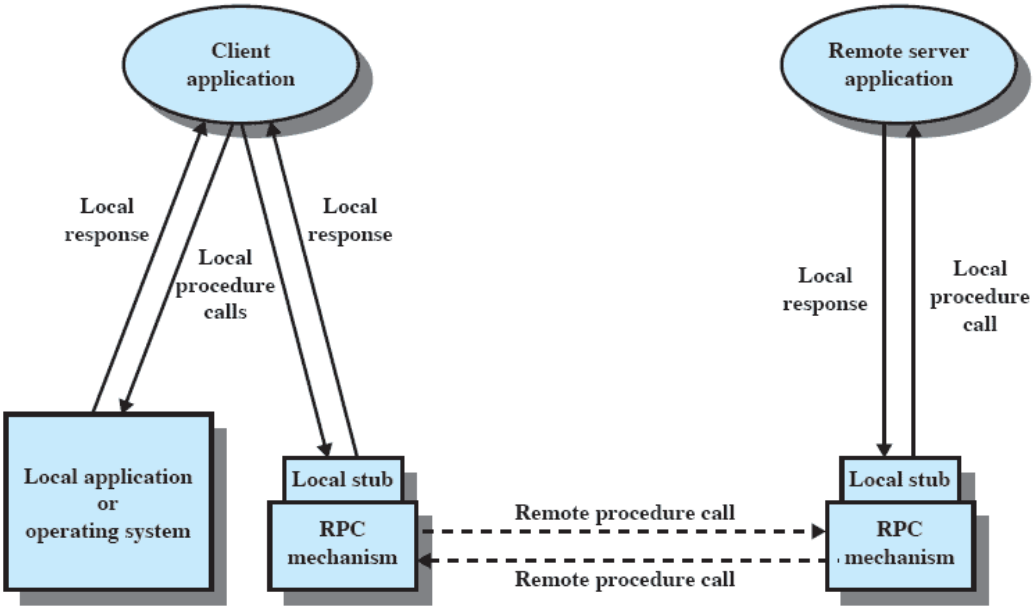
Una variazione del modello basato sul message passing è il moeddo RPC. E' ormai ampiamente accettato e comunemente utilizzato per la comunicazione nei sistemi distribuiti. L'essenza è di permettere ai programmi su differenti macchine di interagire tra loro utilizzando semplici chiamate e ritorni come se stessero sulla stessa macchina.

Vantaggi dell'RPC:

1. E' ampiamente accettato, usato, ed è coerente con l'astrazione dei sistemi distribuiti.
2. L'uso del RPC abilita la specifica delle interfacce remote come un insieme di operazioni nominate con tipo designati. Si può quindi documentare facilmente le interfacce e sui programmi distribuiti è facile ora fare debugging su compatibilità del tipo.
3. Siccome le interfacce definite e standardizzate sono ora specificate, il codice di comunicazione per l'applicazione può essere generato automaticamente.
4. Siccome le interfacce definite e standardizzate sono ora specificate, gli sviluppatori possono scrivere moduli client e server che possono essere trasferiti attraverso pc e sistemi operativi con piccole modifiche e puntualizzazioni.

Il meccanismo dell'RPC può essere visto come un riferimento a un approccio affidabile di un modulo message passing bloccante.

Il programma chiamante effettua una semplice chiamate del tipo CALL P(X,Y) con P il nome della procedura, X i parametri e Y il valore di ritorno.



Un sistema client può effettuare chiamate a funzioni definite localmente oppure affidarsi a chiamate di funzioni che invocano il meccanismo di RPC (può essere o non essere trasparente questa intenzione nei confronti dell'utente). Una volta che quest'ultimo è stato invocato viene contattato il server attraverso la creazione di un messaggio che contiene il nome del servizio e i parametri in dotazione. Il server, una volta computata la risposta, sempre attraverso il meccanismo di RPC ossia creando un messaggio, consegnerà il risultato al Client chiamante che lo conserverà in una variabile di ritorno.
NB la chiamata che il server fa al metodo è equivalente ad una chiamata locale, è giusto quindi affermare che i parametri verranno recuperati in stack.

PARAMETER PASSING
Passare dei parametri per valore è molto semplice, il valore stesso è copiato nel messaggio da inviare. Molto più complicato è passare un parametro per riferimento poiché è necessario un unico puntatore ad un unico ampio sistema per ogni oggetto. Inoltre il peso dell'overhead potrebbe non valerne la pena.

PARAMETER REPRESENTATION
Un altro problema è come rappresentare parametri e valori di ritorno nei messaggi. Nel caso in cui il programma chiamante e quello chiamante siano scritti nello stesso linguaggio di programmazione e siano sullo stesso tipo di macchina con lo stesso OS non ci sono problemi. Ma se differiscono in qualcosa potrebbero esserci incompatibilità. Una soluzione potrebbe essere quella di standardizzare alcuni formati primitivi di comuni oggetti come interi e caratteri così da poter convertire tutti i tipi di parametri specifici in tipi standardizzati.

CLIENT/SERVER BINDING
Il binding specifica come la relazione tra la RPC e il programma chiamante sarà stabilita.
Abbiamo il NONPERSISTENT BINDING, ossia una connessione logica è stabilita tra due processi nel momento in cui si ha una chiamata ad una procedura remota ed è subito smontata nel momento in cui il valore di ritorno è stato consegnato. Siccome l'overhead coinvolto nello stabilire la connessione è importante, questo rende inappropriato l'uso di un nonblocking binding per RPC chiamate frequentemente dallo stesso chiamante.

Il PERSISTENT BINDING invece consiste in una connessione stabilita al momento della chiamata e mantenuta al momento della consegna del valore di ritorno. Potrà così essere riutilizzata per chiamate future. Se poi per un periodo di tempo specifico non vengono rilevate attività la connessione è smontata. E' consigliata per applicazioni che fanno più volte chiamate a RPC, risparmiando sull'overhead di stabilimento di connessione.

SYNCHRONOUS VS ASYNCHRONOUS

Il concetto di sincrono e asincrono sulle RPC è simili a quello di blocking e nonblocking sui message passing. L'approccio tradizionale è quello blocking cioè sincrono che prevede che il processo chiamante aspetti il valroe di ritorno della RPC. Il comportamento della RPC sincrona è prevedibile. Tuttavia non si riesce a sfruttare a pieno il parallelismo offerto dal sistema distribuito, di conseguenza avremo prestazioni inferiori dovute appunto a limiti di interazioni.

In alternativa abbiamo l'approccio nonblocking o asincrono che a sua volta non blocca il chiamante potendo quindi ricevere le risposte solo quando se ne ha il bisogno. Si ha quindi la possibilità di usare il client localmente in completo parallelismo con l'invocazione del server.

MECCANISMO ORIENTATO AGLI OGGETTI

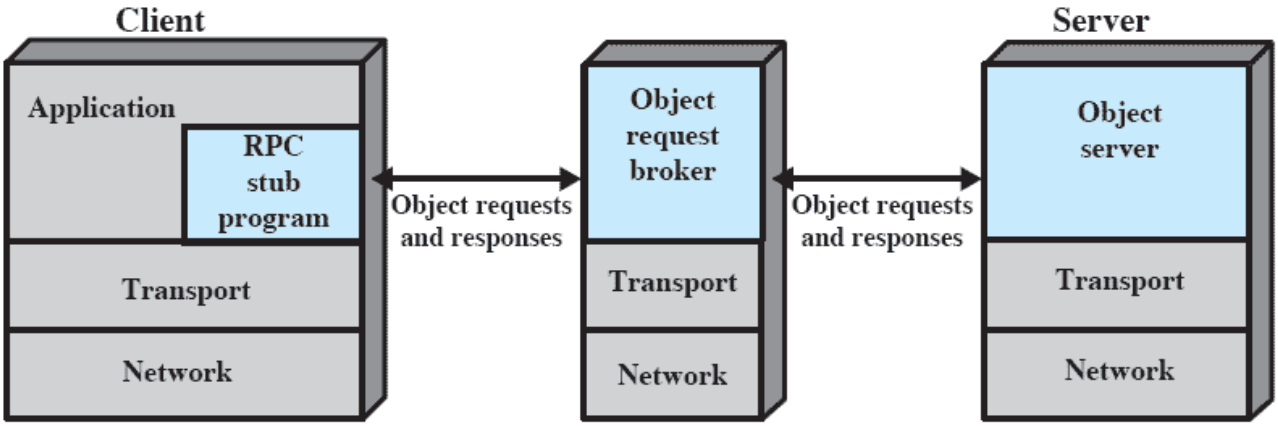
Client e server si inviano messaggi attraverso oggetti.

Un client che ha bisogno di un servizio invia una richiesta ad un object broker (intermediario).

Il broker chiama l'appropriato oggetto e passa qualsiasi dato rilevante.

L'oggetto remoto risolve la richiesta e risponde al broker che ritornerà la risposta al client.

Il successo di questo approccio è dipeso dalla standardizzazione del meccanismo degli oggetti.



SEMANTICA DELLA CHIAMATA RPC

Il normale funzionamento di una RPC potrebbe essere interrotto:

- Il messaggio di chiamata o di risposta è perso.
- Il nodo chiamante si arresta in modo anomalo e viene riavviato
- Il nodo chiamato si arresta in modo anomalo e viene riavviato

La semantica della chiamata determina quanto spesso la procedura remota può essere eseguita sotto condizioni di errore.

At Least Once: Questa semantica garantisce che la chiamata è eseguita una o più volte ma non specifica quali risultati sono tornati al chiamante. Ha poco overhead ed è molto semplice da implementare: usando un timeout basato sulla ritrasmissione senza considerare le chiamate che sono crashsate sul server, il client

continua ad inviare una richiesta al server finché non riceve un ACK. Se uno o più ACK vanno persi il server ptorebbe eseguire la chiamata più volte. Funziona per oeprazioni idempotenti. (non so che vuol dire)

At Most Once: Questa semantica garantisce che la chiamata RPC è eseguita al massimo una volta. O viene eseguita una sola volta o non viene eseguita affatto, dipende dal server se crasha o meno. A differenza della semantica precedente questa semantica richiede il rilevamento di pacchetti duplicati e funziona per operazioni non idempotenti.

Exactly once: questa semantica garantisce che la chiamata RPC avvenga, supponendo che il server se crashato venga riavviato prima o poi. Tiene conto delle chiamate orfane (quelle in cui il server è crashato) e consente loro di essere ripetute, un nuovo server le adotterà. Richiede però un'implementazione molto complessa.

CLUSTERS
E' un'alternativa al symmetric multiprocessing (SMP), possiamo definire un cluster come un gruppo di computer interconnessi che lavorano insieme come se fossero un'unica unità.
Ogni computer può eseguire operazioni in proprio e non solo in riferimento al cluster. Infatti ci si riferisce a ogni pc come nodo.
I benefici sono:

- Assoluta scalabilità: è possibile creare cluster così grande da superare la potenza della più grande macchina singola.
- Scalabilità incrementale: piccoli incrementi dipende dall'aggiunta di nuovi sistemi al cluster.
- Alta disponibilità: il fallimento di un nodo non è una condizione critica del sistema.
- Alto rapporto prezzo/performance: usando componenti a costo inferiore è possibile costruire un sistema la cui potenza in una macchina singola costerebbe molto di più.

METODI DI CLUSTERING: BENEFICI E LIMITAZIONI
Passive Standby: prevede un server secondario che si attiva in casi di fallimento del principale, facile da implementare ma ha un costo maggiore perché non è usato nel processamento di altri task.

Active Secondary: il server secondario è anche usato per processare task. Riduce i costi rispetto al precedente metodo ma aumenta la complessità.

Server separati: Server separati hanno i propri dischi. I dati sono continuamente copiati dal primo al secondo server. Altà disponibilità ma alto anche l'overhead nelle operazioni di copia.

Server connessi a dischi: i server sono collegati agli stessi dischi ma ognuno ha i propri, nel caso un fallisse i suoi dischi verrebbero presi da un altro server. Riduce l'overhead di copia ma necessita di tecnlogia RAID per evitare fallimenti dovuti a errori nei dischi.

Server che condividono dischi: molti server condividono l'accesso ai dischi. Richiede un software che gestisca l'accesso ed è spesso usato con tecnlogia RAID.

PROBLEMI DI PROGETTAZIONE DEL SISTEMA OPERATIVO

Per quanto riguarda la GESTIONE DEGLI ERRORI esistono prevalentemente due approcci:
Utilizzo di cluster ad alta disponibilità:

- Offre un'alta probabilità che tutte le risorse saranno in servizio.
- Ogni richiesta persa, se riprovata, sarà gestita da un altro pc nel cluster.
- L'OS del cluster non da garanzie riguardo lo stato di transizioni eseguite parzialmente.
- Se si verifica un errore, le richieste in corso vengono perse

Cluster che gestiscono gli errori:

- Assicura che tutte le risorse sono sempre disponibili grazie all'uso di dischi condivisi ridondanti e meccanismi per il ritorno di transazioni non richieste e transazioni richieste completate.

La funzione di cambiare sistema ad un applicazione o alle risorse nel momento in cui quello in uso fallisce si chiama FALLOVER.
Il ripristino delle applicazioni e delle risorse sul sistema originale una volta riparato si chiama FALLBACK.
Quest'ultimo approccio può essere automatizzato ma lo si consiglia solo se realmente il problema può essere riparato ed è improbabile che si ripeta.
Il fallback automatico può scatenare una sequenza di errori che porterebbero le risorse a rimbalzare da un pc ad un altro rendendo difficile il recupero e diminuendo drasticamente le prestazioni.

Per quanto riguarda il LOAD BALANCING, un cluster richiede una capacità efficace di bilanciare il carico del lavoro tra il PC disponibili. Ciò ovviamente richiede il fatto che sia incrementalmente scalabile. Quando un nuovo PC viene aggiunto al cluster, l'impianto di bilanciamento del carico dovrebbe automaticamente comprendere la nuova unità. Il middleware deve riconoscere che i servizi possono apparire su diversi nodi del cluster e che possono migrare da un nodo ad un altro.

Per quando riguarda il PARALLELIZING COMPUTATION, esistono tre approcci:

1. Parallelizing compiler: determina, in tempo di compilazione, quali parti dell'applicazioni possono essere eseguite in parallelo. Queste poi sono divise su diversi pc nel cluster. Le prestazioni dipendono dalla natura del problema e se il compilatore è ben progettato.
2. Parallelized application: il programmatore scrive l'applicazione dall'inizio con l'intento di runnarla su un cluster, utilizzando ad esempio message passing per lo scambio di dati tra i nodi del cluster. E' l'approccio più rognoso per il programmatore ma quello che sfrutta al meglio la potenza e la struttura del cluster.
3. Parametric computing: si può utilizzare se l'essenza dell'applicazione è un algoritmo o un programma che deve essere eseguito un grande numero di volte, ogni volta con parametri e condizioni differenti. Esempio un modello di simulazione che effettua un test con differente parametri in ingresso. Sono importanti però tool di gestione dei test che una volta distribuito il carico su vari pc, raccolgano i risultati e li confrontino.

Nella prossima pagina c'è la foto di una tipica architettura di un cluster. Ogni pc è connesso all'altro tramite una rete LAN, ogni pc ha unità di calcolo indipendente e il middleware permette di nascondere all'utente finale le modalità del lavoro effettuato dal sistema distribuito. Su di esso possono essere eseguite sia applicazioni sequenziali che parallele. Quest'ultime con prestazioni decisamente più alte sfruttando la struttura del cluster.

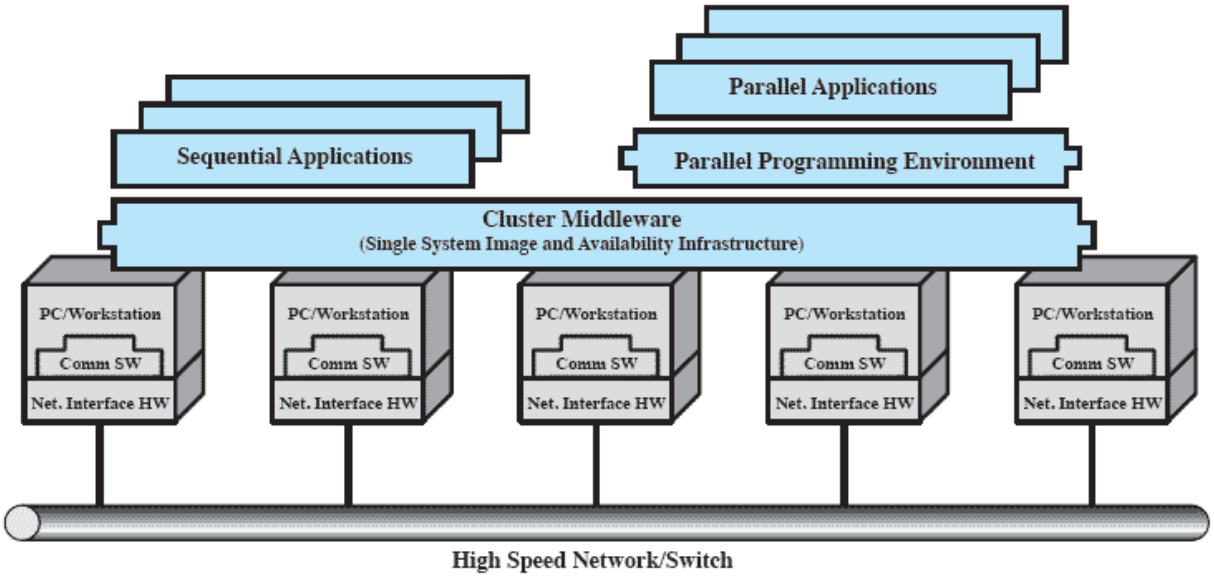


Figure 16.14 Cluster Computer Architecture [BUYY99a]

COMPARAZIONE TRA CLUSTER E SYSTEM MULTIPROCESSOR

- Entrambi forniscono una configurazione multi-proccessor per supportare applicazioni che richiedono prestazioni elevate.
- Entrambe le soluzioni sono commercialmente disponibili.
- SMP “ha più esperienza” (has been around longer than cluster”).
- SMP è più facile gestirlo e configurarlo.
- I prodotti SMP sono ben consolidati e stabili.
- I cluster sono migliori per migliorie e scalabilità
- I cluster sono superiori in termini di disponibilità.