CAPITOLO 1

ARCHITETTURE DEI SISTEMI DISTRIBUITI

Antonio Massari, Massimo Mecella, Enrico Melis, Gaetano Santucci

1. Introduzione

Le architetture dei sistemi informativi si sono sviluppate e evolute nel corso degli anni passando da schemi centralizzati a modelli distribuiti e diffusi, maggiormente rispondenti alle necessità di decentralizzazione e di cooperazione delle moderne organizzazioni. In questa tendenza alla distribuzione svolgono un ruolo importante le tecnologie a oggetti distribuiti e il *Distributed Object Computing (D.O.C.)*.

Questo contributo vuole fornire un quadro d'assieme dello stato dell'arte delle tecnologie per la realizzazione di sistemi distribuiti e permettere la comprensione delle loro potenzialità nel breve e medio termine. In particolare:

- Sono introdotti i concetti di *architettura centralizzata* e di *architettura distribuita* e, dall'analisi critica dei vantaggi e degli svantaggi offerti dai due tipi di architettura, è descritto nelle sue caratteristiche principali il concetto di *Distributed Object Computing*. Si introducono inoltre i concetti riguardanti i *sistemi distribuiti ad accoppiamento forte*, usati principalmente nell'ambito di una medesima organizzazione, e i concetti riguardanti i *sistemi di cooperazione*, usati da più organizzazioni fra loro autonome ma interessate a cooperare per il raggiungimento di un fine comune;
- Sono illustrate le tecnologie principali disponibili per realizzare sistemi ad oggetti distribuiti, individuando la classe dei *middleware* a oggetti, la classe delle tecnologie di incapsulamento di servizi offerti da sistemi *legacy*, la classe delle tecnologie di diffusione di servizi su *web* e la classe delle tecnologie basate su linguaggi di marcatura tipo *XML*;
- È fornita una trattazione delle principali architetture tecnologiche adottate e dei possibili contesti applicativi nei quali il *Distributed Object Computing* può trovare fertile campo di applicazione, sia in ambito pubblico, sia in ambito privato;
- Sono svolte alcune considerazioni finali riguardo alle criticità attuali e alle linee evolutive future prevedibili per il settore.

2. Sistemi centralizzati, sistemi distribuiti e sistemi cooperativi

Si parla di *sistema informatico centralizzato* quando i dati e le applicazioni risiedono in un unico nodo elaborativo (Figura 1).

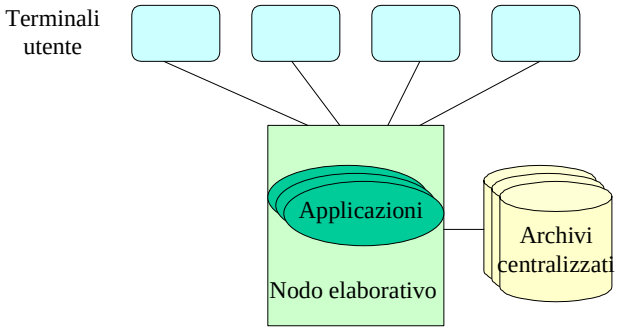

Figura 1: Sistema informatico centralizzato

Viceversa, si parla di *sistema informatico distribuito* quando almeno una delle seguenti due condizioni è verificata [Coulouris et al., 1994] [Goscinski, 1991] [Mullender, 1993] [Simon, 1996] (Figura 2):

- le applicazioni, fra loro cooperanti, risiedono su più nodi elaborativi (elaborazione distribuita);
- il patrimonio informativo, unitario, è ospitato su più nodi elaborativi (base di dati distribuita).

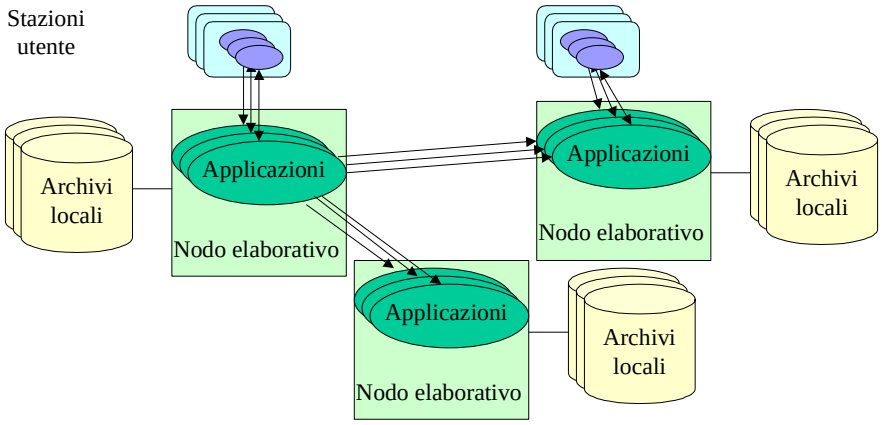

Figura 2: Sistema informatico distribuito

In termini generali, quindi, un sistema distribuito è costituito da un insieme di applicazioni logicamente indipendenti che collaborano per il perseguimento di obiettivi comuni attraverso una infrastruttura di comunicazione hardware e software.
I sistemi distribuiti possono essere suddivisi in due importanti categorie:

- *Sistemi ad accoppiamento forte*: questi sistemi sono tipicamente concepiti in modo unitario e usano risorse informative e elaborative controllate da una o più unità organizzative comunque facenti riferimento a una unica autorità. Esempi tipici di questa tipologia di sistemi possono essere i sistemi di supporto che le grandi organizzazioni private possono realizzare per le loro forze di vendita distribuite su tutto il territorio di interesse, ovvero i sistemi di automazione di sportello propri di enti pubblici e privati di tipo diverso, quali banche, presidi sanitari, enti di riscossione dei tributi. In questo scenario si prediligono logiche di *unitarietà* nelle scelte architetturali e tecnologiche, di *integrazione* dei sottosistemi in un unico sistema complessivo, di *transazionalità* nella realizzazione dei servizi da fornire all'utenza, concepiti come pacchetti di lavoro unitari di cui si deve garantire l'esecuzione e il risultato finale atteso. Il risultato che si ottiene è quello di un *elevata qualità del servizio* reso all'utenza e di un *aumento dei vincoli organizzativi e tecnologici* esistenti fra le unità organizzative che cooperano per la fornitura del servizio stesso;
- *Sistemi a accoppiamento debole*: questi sistemi, chiamati *sistemi di cooperazione* [Brodie 1998], [Laufmann et al., 1995], [Mylopoulos, Papazoglou 1997], nascono tipicamente dalla messa a fattore comune di risorse informative e elaborative preesistenti e proprie di soggetti organizzativi fra loro autonomi che, per motivi istituzionali o di *business*, hanno interesse a cooperare per fornire servizi a valore aggiunto. Esempi tipici di questa tipologia di sistemi sono i sistemi di cooperazione di cui si possono dotare le amministrazioni coinvolte in processi di servizio complessi di pubblico interesse (quali l'ordine pubblico, la formazione professionale, la sanità), ovvero i diversi soggetti privati coinvolti nelle catene del valore istituite per la produzione di beni e servizi da vendere sul mercato. In questo scenario si prediligono logiche di *normalizzazione* nelle scelte architetturali e tecnologiche, al fine di assicurare la presenza di *interfacce* verso le risorse messe a fattore comune rispondenti a una logica *unitaria*, logiche di *federazione* dei sottosistemi nel sistema complessivo, logiche di *proceduralità* nella realizzazione dei servizi da fornire all'utenza, concepiti come processi di cui si deve garantire l'avanzamento e il tracciamento. Il risultato che si ottiene è quello di un *aumento dell'efficienza e della trasparenza* dei processi di cooperazione automatizzati, di un *aumento dell'usabilità* dei servizi stessi mediante l'istituzione di *sportelli unici* e di un *modesto aumento dei vincoli*

organizzativi e tecnologici esistenti fra le unità organizzative che cooperano per la fornitura del servizio stesso, limitato all'istituzione di *regole di interfacciamento concordate*.

2.1. Il meccanismo RPC

Un meccanismo di base particolarmente importante per consentire il dialogo di applicazioni presenti su macchine diverse è il meccanismo *Remote Procedure Call – RPC*. Tale meccanismo, introdotto nel 1984 da Birrell e Nelson [Birrell, Nelson 1984], permette a un programma di richiamare procedure remote (ovvero eseguite su un calcolatore diverso da quello ospitante il programma chiamante) come se fossero locali. L'ottenimento di questo risultato permette ai progettisti e ai programmatori di realizzare con facilità applicazioni distribuite, in quanto le metodologie e le tecniche progettuali e di programmazione usate per i sistemi centralizzati possono essere riapplicate nello scenario distribuito, mentre la complessità legata alla distribuzione è gestita dal meccanismo *RPC* sottostante.
La figura 3 illustra il meccanismo *RPC* e i componenti architetturali che lo caratterizzano.

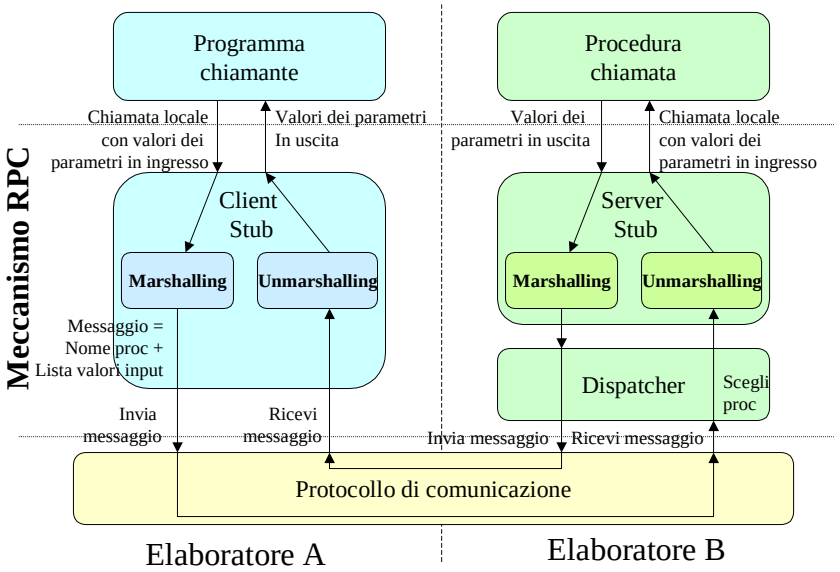

Figura 3: Schema architetturale del meccanismo RPC

Quando il programma principale invoca l'esecuzione di una procedura remota, il meccanismo di invocazione è del tutto analogo a quello previsto, nel particolare linguaggio di programmazione usato, per la chiamata di una procedura locale. Poiché la procedura remota da chiamare è caratterizzata da un *nome* che la identifica, da una *lista di parametri di ingresso* e da una *lista di parametri di uscita*, la sua chiamata mediante *RPC* provoca in realtà l'invocazione di un modulo locale, chiamato *client stub*, avente lo stesso nome e gli stessi parametri in ingresso e in uscita della procedura remota. Il *client stub*, quindi, ha il compito di *rappresentare localmente* la procedura remota, nascondendo al programma chiamante la complessità della sua invocazione fisica. Il *client stub* riceve, dapprima, i valori associati ai parametri d'ingresso, passati dal programma chiamante, e provvede a impacchettarli in un messaggio (*marshalling*). Al messaggio è accodato il nome della procedura richiesta. Il messaggio così composto è trasmesso al sistema che ospita il programma chiamato mediante il protocollo di comunicazione disponibile. Il messaggio ricevuto dal sistema che ospita il programma chiamato è dapprima interpretato per individuare il nome della procedura da eseguire. Il modulo che individua, in base al nome passato, la procedura da invocare è chiamato *dispatcher*. Il *dispatcher* quindi invoca il *server stub* della procedura richiesta, passando a questo il resto del messaggio ricevuto. Il *server stub* spacchetta il messaggio (*unmarshalling*) e chiama la procedura richiesta con la lista dei valori dei parametri in ingresso passati. L'esecuzione della procedura fornisce al *server stub*, di ritorno, i valori calcolati per i parametri in uscita previsti. Tali valori sono impacchettati (*marshalling*) e ritrasmessi sottoforma di messaggio al sistema chiamante

attraverso il protocollo di comunicazione disponibile. Il *client stub* riceve il messaggio, ricostruisce i valori passati per i parametri di ritorno previsti (*unmarshalling*) e li restituisce, in ultimo, al programma chiamante.
Lo schema di funzionamento illustrato è soggetto a una serie di problematiche e di difficoltà che rendono non completamente analoga la chiamata di una procedura remota rispetto alla chiamata di una procedura locale. In particolare, le problematiche principali riguardano:

- I tipi di dato che possono essere usati per i parametri in ingresso e in uscita. Sono generalmente disponibili i tipi di dato elementari (come i caratteri, la sequenza di caratteri, i numeri interi) mentre possono sussistere limitazioni per i tipi di dato composti e per i tipi di dato puntatore. Per i puntatori, in particolare, la difficoltà di renderne possibile l'uso come parametri risiede nel fatto che fanno riferimento ad aree di memoria locali al programma chiamante ma non accessibili da parte della procedura chiamata che è ospitata su un elaboratore diverso. Per lo stesso motivo, non è generalmente possibile usare il meccanismo delle variabili globali come sistema per lo scambio implicito di informazioni fra programma chiamante e procedura chiamata;
- I formati dei dati adottati dal programma chiamante e dalla procedura chiamata. In presenza di elaboratori di tipo diverso (ambienti eterogenei), può accadere che il formato fisico delle informazioni scambiate sia diverso nei due ambienti. In queste situazioni, è necessario che il protocollo di comunicazione adottato fornisca un formato di trasferimento delle informazioni unitario per i due ambienti, nonché funzioni di trasformazione dei dati dal formato nativo di ogni ambiente al formato di trasferimento e viceversa;
- La gestione delle eccezioni. Il comportamento di una procedura remota, in condizioni normali, può apparire del tutto analogo a quello di una procedura locale. L'insieme dei malfunzionamenti che possono presentarsi in uno schema *RPC* è però più ampio e complesso di quello che caratterizza un sistema centralizzato. In particolare, il malfunzionamento di una procedura non blocca necessariamente la procedura che con questa coopera remotamente, ovvero può presentarsi un malfunzionamento sul canale trasmissivo senza che ciò blocchi l'elaborazione sulle macchine ad esso connesse. Tale scenario più complesso non può essere completamente mascherato dal meccanismo *RPC* per cui la gestione delle eccezioni da prevedere all'interno dei programmi chiamanti e delle procedure chiamate è più vasta e difficile;
- L'invocazione concorrente della stessa procedura remota da parte di più programmi chiamanti ospitati in macchine diverse. Il meccanismo *RPC*, permettendo l'invocazione remota di procedure, rende realistico lo scenario in cui più programmi chiamano contemporaneamente la stessa procedura da macchine diverse. In questo scenario, il meccanismo *RPC* deve assicurare l'accesso concorrente alla stessa procedura in modo consistente, accodando le diverse richieste e costituendo, per ognuna di esse, uno spazio di esecuzione isolato e distinto.

2.2. I sistemi client-server

Le applicazioni che costituiscono un sistema distribuito sono caratterizzate dal ruolo che svolgono nel sistema stesso:

- *Cliente (Client)*: una applicazione assume il ruolo di Cliente quando è utilizzatore di servizi messi a disposizione da altre applicazioni;
- *Servente (Server)*: una applicazione assume il ruolo di Servente quando è fornitore di servizi usati da altre applicazioni;
- *Attore (Actor)*: una applicazione assume il ruolo di Attore quando assume in diverse situazioni sia il ruolo di Cliente sia quello di Servente.

Un tipo particolare di sistema distribuito è quello *client-server*, caratterizzato dalla relazione di servizio secondo la quale più processi cliente avviano un dialogo richiedendo servizi forniti da processi serventi corrispondenti. Il dialogo fra processi clienti e processi serventi è generalmente di tipo sincrono, ovvero:

- Il processo servente è in generale inattivo, in attesa che gli pervenga una richiesta di servizio;

- Il processo cliente, quando invia una richiesta di servizio, rimane bloccato, in attesa del ricevimento della corrispondente risposta (Figura 4).

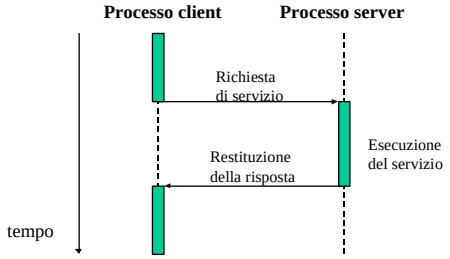

Figura 4: Relazione di servizio fra processo cliente e processo servente

La relazione cliente-servente in generale implica l'uso di protocolli di comunicazione *asimmetrici*, in quanto tipicamente un processo servente può servire più processi cliente; per questo motivo il processo servente gestisce risorse condivise (dalla pluralità di processi cliente che a lui si riferiscono) e deve essere progettato con molta cura, in quanto deve operare in condizioni di carico elevato (molte richieste contemporanee), deve fornire meccanismi per salvaguardare la consistenza nell'accesso alle risorse condivise (tipicamente la base informativa a cui si appoggia) e deve essere scalabile, cioè deve potere aumentare le proprie dimensioni e capacità di servizio per soddisfare un numero crescente di processi cliente.
Per quanto detto in precedenza, i sistemi *client-server* si inquadrano naturalmente nell'ambito dei sistemi distribuiti ad accoppiamento forte per i seguenti due motivi:

- il processo cliente, una volta che abbia richiesto un servizio, rimane generalmente bloccato in attesa che il servizio stesso sia completato. Tale schema comportamentale del processo cliente è ragionevole solo se il servizio richiesto è di tipo transazionale e quindi sia completato in tempi brevi e in qualche modo predicibili; in altre situazioni è più conveniente una logica *non bloccante* di *invio asincrono della richiesta di servizio* da parte del processo cliente al processo servente interessato e di successiva *notifica di esecuzione del servizio*, da parte del processo servente al processo cliente, quando il servizio stesso sia stato completato;
- il processo servente rimane in attesa dell'arrivo di richieste da parte dei processi cliente che a lui fanno riferimento. Tale schema comportamentale, che implica il consumo di risorse elaborative e di comunicazione, è ragionevole solo se il flusso di richieste provenienti dai processi cliente è sostenuto e predicibile, altrimenti risulta maggiormente conveniente una logica di *attivazione* del processo servente ogni volta che pervenga una richiesta da soddisfare.

2.3. L'evoluzione delle architetture dei sistemi informatici

I sistemi centralizzati sono nati con l'informatica moderna negli anni '50 e si sono sviluppati negli anni '60 e '70 grazie all'affermarsi delle tecnologie dei *mainframe*, dei sistemi operativi *time-sharing*, dei *file system* e dei sistemi di gestione di basi di dati (*Data Base Management System – DBMS*) centralizzati di tipo gerarchico e reticolare. La nascita e lo sviluppo, negli anni '70 e '80, di nuove tecnologie più economiche, versatili e facili da usare (mini e micro elaboratori, reti locali di comunicazione, sistemi di gestione di basi di dati relazionali, architetture *client-server*, interfacce utente di tipo grafico) ha portato alla crisi del modello centralizzato, evidenziandone la minore economicità e qualità del servizio, e ha promosso la realizzazione di sistemi distribuiti in realtà grandi, medie e piccole, dando luogo al fenomeno dell' "informatica diffusa".
Questo fenomeno di decentralizzazione ha avuto grande impulso per via dei seguenti motivi principali:

- crollo dei prezzi degli apparati hardware e delle relative licenze software;
- maggiore scalabilità, continuità e qualità del servizio da parte dei sistemi distribuiti rispetto a quelli centralizzati;

- maggiore capacità, da parte dei sistemi distribuiti, di venire incontro alle esigenze di flessibilità e di autonomia delle moderne organizzazioni.

I sistemi distribuiti implicano scelte gestionali differenti da quelle dei sistemi centralizzati tradizionali. In questi, tipicamente omogenei, il controllo del sistema operativo e della comunicazione fra i terminali e l'elaboratore centrale richiede un gruppo di analisti e di sistemisti dedicato, che sia in grado di assicurare il regolare funzionamento operativo del sistema, e la manutenzione e la gestione di configurazione del sistema sono perfettamente definiti, in quanto vengono svolti in modo centralizzato.

Con l'avvento dei sistemi distribuiti nasce la possibilità di scegliere e combinare, a vari livelli dell'architettura, componenti provenienti da fornitori diversi. La definizione del sistema richiede la risoluzione di un insieme di problemi, quali la scelta della piattaforma per le stazioni di lavoro degli utenti, la scelta della piattaforma (in generale differente) che deve ospitare le logiche elaborative e le risorse informative, la scelta dei protocolli di comunicazione e dei formati dei dati fra i diversi componenti architetturali. A fronte della maggiore complessità nella definizione e gestione del sistema, causata principalmente da fenomeni di *eterogeneità*, si ottenengono in cambio vantaggi economici dovuti alla possibilità di scegliere per ogni componente quanto di meglio (in termini economici e/o qualitativi) offre il mercato.

Nei primi anni '90, sulla base dell'esperienza maturata nella gestione dell'informatica diffusa, il modello distribuito è stato sottoposto a forte critica [Duchessi, Chengalur-Smith, 1998] proprio per la maggiore complessità progettuale, che determina in generale maggiori costi realizzativi e minore robustezza delle realizzazioni, e per la maggiore complessità gestionale, che genera costi spesso nascosti per gli utenti e le organizzazioni.

L'analisi critica dei punti di forza e dei punti di debolezza del modello centralizzato e del modello distribuito ha portato la comunità scientifica ed i fornitori di tecnologie ad elaborare un nuovo modello di elaborazione, il *Distributed Object Computing* - *D.O.C.*, che tende a fornire un contesto virtualmente unitario di elaborazione, in cui più processi elaborativi (oggetti) cooperano come se risiedessero su un'unica macchina [Krieger, Adler, 1998] (Figura 5).

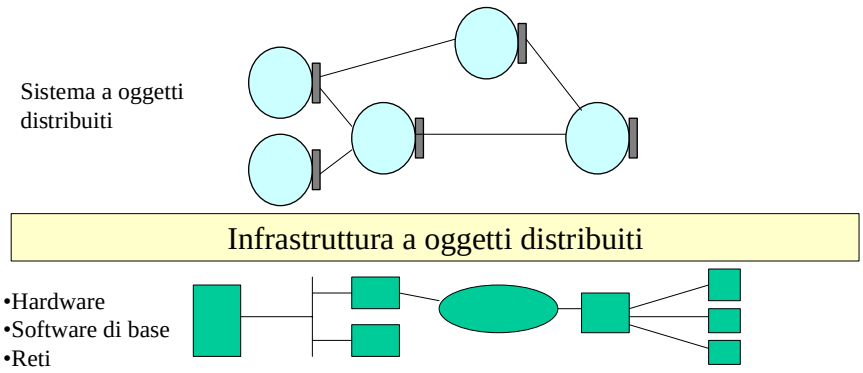

Figura 5: Infrastruttura per sistemi a oggetti distribuiti

Tale contesto unitario di elaborazione è realizzato mediante una infrastruttura tecnologica ad oggetti distribuiti che ne permette la distribuzione fisica su più macchine connesse tra loro in rete, in modo che:

- la progettazione, la realizzazione e la gestione delle applicazioni basate sull'infrastruttura può avvenire a livello operativo con tecniche e complessità simili a quelle dell'ambito dei sistemi centralizzati;
- si demanda ai fornitori e agli specialisti di infrastrutture ad oggetti distribuiti la risoluzione di tutte le problematiche architetturali, tecnologiche e gestionali proprie delle infrastrutture stesse.

L'uso pratico del *Distributed Object Computing* nei diversi contesti reali ha portato infine, nel corso della seconda metà degli anni '90, a una ulteriore maturazione delle metodologie e delle tecnologie

da adottare in situazioni di *distribuzione ad accoppiamento forte* (nel caso di sistemi destinati a unità organizzative per le quali è possibile identificare, a qualche livello, una unica autorità controllante) e in situazioni di *distribuzione ad accoppiamento debole* (nel caso di *sistemi di cooperazione* fra organizzazioni diverse e fra loro autonome).
Nelle situazioni di distribuzione ad accoppiamento forte, risulta pienamente valido il concetto di *unica macchina virtuale*, costituita dall'infrastruttura a oggetti distribuiti, grazie alla quale è possibile realizzare il sistema come se risiedesse su un'unica macchina e come se sfruttasse un insieme di risorse informative e elaborative omogenee. In tale ambito, gli strumenti metodologici caratteristici sono quelli della *progettazione top-down* delle risorse informative (basi di dati) e delle funzioni che su queste operano, ovvero della *integrazione in schemi unitari* di risorse informative e elaborative esistenti. I servizi da erogare all'utenza sono tipicamente concepiti come *transazioni* per le quali sono da assicurare il completamento positivo ovvero il rigetto totale, evitando di lasciare il sistema in stati intermedi inconsistenti. Le tecnologie tipiche che si adoperano per questi sistemi sono le *tecnologie di integrazione di basi di dati*, le *tecnologie di gestione di basi di dati distribuite*, le *tecnologie di gestione di transazioni* in ambiente distribuito e a oggetti.
Nelle situazioni di distribuzione ad accoppiamento debole (sistemi di cooperazione), l'infrastruttura a oggetti distribuita è chiamata non tanto a integrare contributi informativi e funzionali eterogenei fornendo di questi, a qualche livello, una visione unitaria, quanto a permettere a tali contributi, forniti da organizzazioni fra loro autonome e aventi interesse alla collaborazione, di essere messi a fattore comune per fornire nuovi servizi a valore aggiunto. In questo quadro di federazione, l'enfasi è quindi posta sull'identificazione di interfacce di scambio di dati e di richiesta di servizi rispondenti a una logica unitaria e concordata, nonché su meccanismi che permettano l'attivazione e l'esecuzione progressiva dei servizi messi a fattore comune, rispettando contemporaneamente l'autonomia organizzativa e tecnologica dei diversi soggetti cooperanti. In tale ambito, gli strumenti metodologici caratteristici sono quelli della *progettazione di schemi unitari* di risorse informative e elaborative esistenti. Nella progettazione degli schemi unitari dei dati si deve fare largo uso di formalismi e tecniche che permettano di distinguere i *concetti astratti di interesse comune* dai *meccanismi concreti* con i quali tali concetti sono istanziati nelle diverse organizzazioni che partecipano al progetto. In tale ambito hanno quindi importanza primaria le attività di progettazione di *dizionari dei dati* (*repository*) che contengano le definizioni dei concetti di interesse comune e le *regole di corrispondenza* di tali concetti con i concetti trattati presso le diverse organizzazioni. La corretta attuazione di questo passo progettuale è condizione indispensabile per garantire *stabilità* nel tempo degli schemi unitari concordati e *indipendenza* delle organizzazioni cooperanti relativamente alle proprie politiche di sviluppo e gestione dei patrimoni informativi locali. Quanto affermato per le risorse informative vale anche per le funzionalità che ogni organizzazione partecipante deve assicurare al sistema cooperativo. In questo tipo di sistemi particolare cura deve essere riposta nella *definizione dei servizi messi a disposizione e delle loro interfacce di invocazione* (che cosa fa il servizio e come può essere richiesto), lasciando piena libertà all'organizzazione che lo rende disponibile di variare nel tempo le modalità organizzative e tecnologiche con le quali il servizio concordato è assicurato. I servizi da erogare all'utenza sono tipicamente concepiti come *procedure* per le quali sono da assicurare il *tracciamento* e l'*avanzamento affidabile* lungo la catena di passi concordata. In questo scenario, quindi, la procedura nel suo complesso non è trattata come una unica transazione, bensì come un *insieme di passi* ognuno dei quali può godere delle caratteristiche di transazionalità. Il passaggio da un passo a quello successivo è ottenibile con logiche di tipo *asincrono* e a *scambio di messaggi*, diverse da quelle di tipo *sincrono* e a *colloquio diretto fra processi* proprie di scenari transazionali distribuiti. Le tecnologie tipiche che si adoperano per questi sistemi sono conseguentemente le *tecnologie di descrizione e trattamento di meta-informazioni* (fra le quali sono in particolare da ricordare le tecnologie facenti riferimento ai linguaggi di marcatura tipo *XML*), le *tecnologie di specifica delle interfacce di invocazione fra oggetti*, le *tecnologie di gestione di reti di code di messaggi* per realizzare gli schemi procedurali previsti in fase progettuale.

Al fine di completare la parte introduttiva riguardante i sistemi distribuiti e il *Distributed Object Computing*, i due paragrafi che seguono descrivono:

- le caratteristiche principali del *modello basato su oggetti* che si adotta nella descrizione di sistemi a oggetti distribuiti. Tale modello è quindi l'*astrazione* messa a disposizione, in generale, da una infrastruttura a oggetti distribuiti;
- le caratteristiche principali dell'insieme di tecnologie che costituiscono l'infrastruttura a oggetti distribuiti e che correntemente rientrano nella denominazione generale di *middleware*.

2.4. Il modello basato su oggetti

Il modello *basato* su oggetti si sviluppa agli inizi degli anni '90 grazie all'opera di diversi ricercatori operanti nell'area dei sistemi operativi distribuiti [Mullender, 1993] come semplificazione del modello *orientato* agli oggetti, in modo da consentire una rappresentazione a oggetti di applicazioni complesse che siano facilmente realizzabili e distribuibili su reti di grandi dimensioni.

Il modello mette a disposizione i seguenti costrutti di rappresentazione principali:

- l'**oggetto**, insieme integrato di dati (che costituiscono il suo *stato*) e di funzioni (che costituiscono il suo *comportamento*) su di essi operanti; ogni oggetto è dotato di una propria *identità* che lo differenzia dagli altri a prescindere dallo stato in cui si trova ed è creato come istanziazione di un modello, detto **classe**;
- l'**interfaccia**, che raggruppa i servizi messi a disposizione da un oggetto e invocabili da altri oggetti; ogni oggetto può possedere una o più interfacce;
- la **connessione** tra due oggetti, che costituisce il meccanismo con il quale i servizi forniti dalle interfacce dell'uno (servente) sono usati dall'altro (cliente); una connessione può essere *sincrona* (bloccante per il cliente) o *asincrona* (non bloccante), di tipo *pull* (il cliente chiede l'esecuzione del servizio) o di tipo *push* (il servente propone attivamente il servizio).

La figura seguente mostra la simbologia comunemente usata per rappresentare i diversi costrutti.

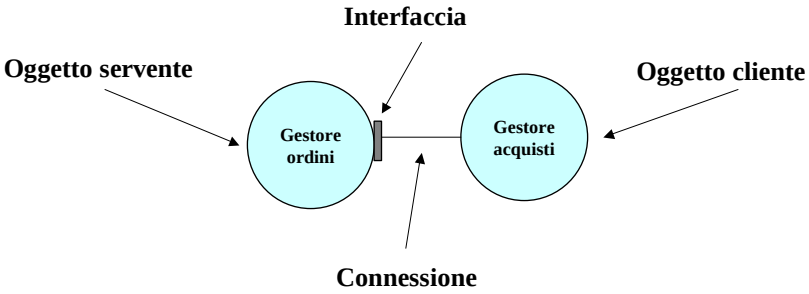

Figura 6: Il modello basato su oggetti

Il modello basato su oggetti è più povero del modello orientato agli oggetti almeno per i seguenti due motivi:

- Non è presente il concetto di *ereditarietà* fra classi;
- Non è affermata esplicitamente la corrispondenza, tipica della modellazione concettuale dei dati, fra oggetti del modello e oggetti del mondo reale da rappresentare.

Il primo aspetto è giustificato dalla difficoltà, da parte di molte tecnologie a oggetti distribuiti, di definire e mantenere correttamente e efficientemente le gerarchie di classi di oggetti in un contesto distribuito.

Il secondo aspetto permette al modellista di individuare oggetti corrispondenti a componenti architetturali (processi) che possano essere trattabili da una infrastruttura a oggetti distribuiti, piuttosto che a concetti del mondo reale. Ad esempio, nella precedente Figura 6, l'oggetto "Gestore ordini", componente del sistema, sostituisce il più generale oggetto "Ordine", rappresentativo di un concetto della realtà rappresentata.

Il modello basato su oggetti salvaguarda la proprietà di **incapsulamento** del patrimonio privato di dati e di funzioni di ogni oggetto. Tale proprietà, presente nei modelli orientati agli oggetti, è assicurata dal costrutto di interfaccia, che è l'unico meccanismo mediante il quale oggetti clienti possono usufruire dei servizi messi a disposizione da oggetti serventi. La proprietà di incapsulamento del modello permette di affermare che ogni oggetto è:

- **unità di esecuzione**: ogni oggetto è dotato delle risorse necessarie (dati e funzioni) necessarie per l'esecuzione dei servizi forniti all'esterno;
- **unità di fallimento**: i malfunzionamenti che si verificano nell'esecuzione di un oggetto sono circoscritti all'oggetto stesso e non degradano il funzionamento degli altri oggetti del sistema;
- **unità di attivazione**: ogni oggetto costituisce un insieme autoconsistente di dati e funzioni da allocare su un elaboratore;
- **unità di distribuzione**: ogni oggetto, nell'ambito di un sistema distribuito, può essere collocato su un nodo diverso della rete, senza che ciò impedisca il funzionamento del sistema;
- **unità di replicazione**: ogni oggetto è candidato ad essere replicato, per aumentare la robustezza e l'efficienza del sistema;
- **unità di parallelismo**: ogni oggetto può costituire un processo elaborativo autonomo, permettendo così l'elaborazione parallela su una o più macchine;
- **unità di realizzazione**: ogni oggetto può essere realizzato in modo indipendente dagli altri, costituendo quindi il pacchetto di lavoro elementare da affidare ad un gruppo di sviluppatori.

La Figura 7 mostra, a titolo esemplificativo, lo schema ad oggetti di un sistema per il commercio elettronico.

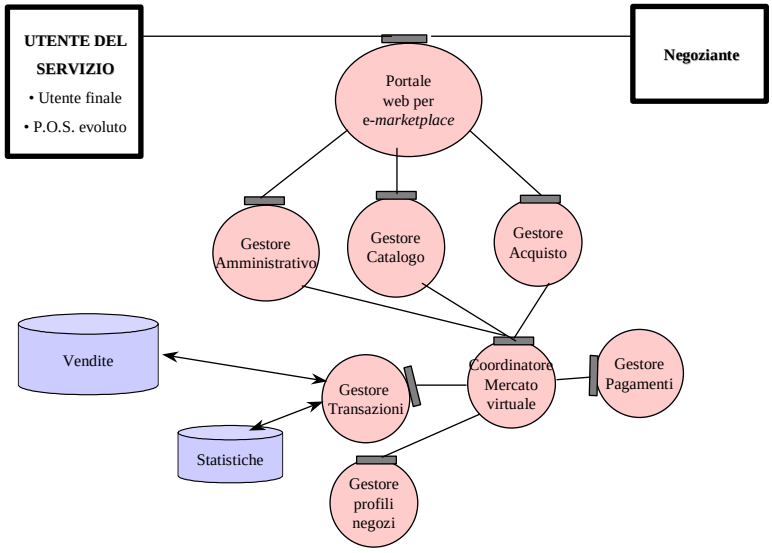

Figura 7: schema a oggetti di un sistema di commercio elettronico

Nel modello introdotto, la comunicazione tra due oggetti può avvenire secondo due differenti modalità: *sincrona* e *asincrona*.

La modalità sincrona è analoga alla chiamata a procedura dei linguaggi di programmazione tradizionali, con il relativo passaggio dei parametri. Secondo questa modalità, l'oggetto che richiede il servizio (oggetto cliente) usa l'interfaccia messa a disposizione dall'oggetto che lo fornisce (oggetto servente), identificando il servizio per nome e passando i valori dei parametri d'ingresso previsti. L'oggetto servente, ricevuta la richiesta, esegue l'insieme delle operazioni necessarie per l'erogazione del servizio, restituendo infine all'oggetto cliente l'insieme dei valori calcolati per i parametri di ritorno previsti. Gli oggetti cliente e servente devono essere contemporaneamente attivi nel momento in cui il servizio viene richiesto e l'oggetto cliente rimane in attesa finchè l'oggetto servente non completa il compito richiesto e restituisce la risposta. Tale modalità implica un forte

accoppiamento tra gli oggetti cliente e servente, in quanto l'oggetto cliente rimane bloccato, consumando risorse, dal momento in cui ha inviato la richiesta a quello in cui ottiene la risposta.
Nella modalità asincrona gli oggetti interagenti possono essere attivi in momenti differenti, in quanto la comunicazione avviene attraverso lo scambio di messaggi unidirezionali: l'oggetto cliente invia un messaggio a una coda d'attesa, l'oggetto servente preleva i messaggi dalla coda. In questo modo c'è totale indipendenza tra i due oggetti e la coda di messaggi in attesa costituisce l'elemento di disaccoppiamento. La modalità di comunicazione asincrona può essere realizzata secondo quattro schemi di interazione tra oggetti cliente e servente:

- *Publish & Subscribe* – Gli oggetti cliente e servente non comunicano in modo diretto, ma partecipano a una relazione ternaria in cui gli oggetti servente (*editori* o *publisher*) forniscono informazioni, gli oggetti cliente (*sottoscrittori* o *subscriber*) le ricevono e un terzo soggetto (il *distributore* o *distributor*) si occupa della distribuzione ai sottoscrittori delle informazioni prodotte dagli editori. La Figura 8 mostra lo schema a oggetti di riferimento per tale modalità di comunicazione asincrona.
- *Multicasting* - In questo schema c'è una conoscenza diretta dei riceventi da parte della sorgente della informazione. Tipicamente la diffusione multi-punto (*multicasting)* è fatta direttamente dal soggetto sorgente di informazioni, che si connette ai riceventi e fornisce i messaggi. La Figura 9 mostra lo schema a oggetti di riferimento per tale modalità di comunicazione.
- *Instance Based Routing* - E' una variante dello schema *Publish&Subscribe*, in cui i sottoscrittori non solo dichiarano l'interesse in qualche tipo di messaggi, ma hanno la possibilità di fornire criteri di selezione ulteriori, con cui il distributore determina se effettivamente consegnare certi messaggi. In sostanza mentre nello schema *Publish&Subscribe* si possono distinguere i messaggi solamente in base al loro tipo, in questo schema si ha la possibilità di raffinare ulteriormente le tipologie anche in base a sottocaratteristiche degli stessi.
- *Store & Forward* - Nello schema *Publish&Subscribe* il messaggio viene consegnato ai sottoscrittori solo se questi hanno una connessione attiva con il distributore; il messaggio infatti è consegnato immediatamente a tutti i sottoscrittori connessi al momento e quindi cancellato. In pratica il meccanismo è asincrono tra gli oggetti terminali (editore e sottoscrittore) ma richiede sincronicità tra distributore e sottoscrittore. Nello schema *Store&Forward*, invece, il messaggio viene memorizzato dal distributore non appena ricevuto dall'editore e viene mantenuto fino alla consegna a tutti i sottoscrittori. La consegna avviene al momento della connessione dei sottoscrittori al distributore o su iniziativa del distributore allo scadere di un intervallo di tempo prefissato.

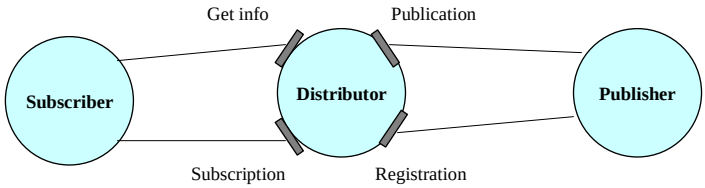

Figura 8: Schema a oggetti della comunicazione Publish&Subscribe

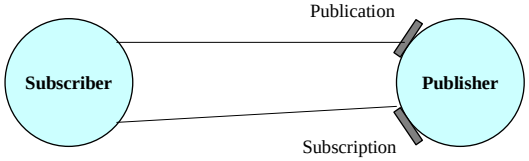

Figura 9: Schema a oggetti della comunicazione Multicasting

Al fine di mettere maggiormente in luce le caratteristiche di tali quattro modalità di comunicazione asincrona, si propone un esempio attinente la realtà della Pubblica Amministrazione, in particolare,

la problematica della diffusione di leggi e normative presso tutti i soggetti interessati, pubblici e privati.
Lo schema *Publish & Subscribe* è adatto per soddisfare le esigenze informative generali di una pluralità di soggetti (sottoscrittori), a fronte di una pluralità di enti normatori (editori). Un ente terzo o uno degli enti normatori è in tale caso chiamato a svolgere il ruolo di distributore. Ogni ente normatore inizialmente dichiara, mediante il servizio di registrazione, la propria volontà a pubblicare le normative prodotte che poi, mediante il servizio di pubblicazione, mette a disposizione. Ogni sottoscrittore inizialmente sottoscrive l'abbonamento al servizio, specificando gli enti della cui produzione normativa è interessato. In seguito, il distributore provvede a fornire a ogni sottoscrittore le norme nel frattempo prodotte, inviandogliele (servizio *push*) o fornendogliele su richiesta (servizio *pull*).
Lo schema *Multicasting* è adatto quanto l'ente normatore interessato alla pubblicazione sia uno solo. In tale caso, i sottoscrittori si rivolgono direttamente a lui, sia per l'abbonamento al servizio, sia per la fornitura delle normative prodotte. Il meccanismo *Multicasting* richiede, in generale, la conoscenza reciproca da parte degli utenti e del fornitore del servizio. Nel caso in cui il servizio informativo sia aperto al pubblico, si parla più propriamente di *Broadcasting*.
Lo schema *Instance Based Routing* permette, nel nostro esempio, di fornire un servizio informativo più diversificato, consentendo all'utente di selezionare le norme da ricevere non solo in base alla tipologia (per esempio, "leggi regionali") e all'ente normatore (per esempio una specifica Regione) ma anche in base al contenuto (per esempio, norme riguardanti le tematiche ambientali, ovvero norme nel cui testo ricorre il termine "documento informatico").
Lo schema *Store & Forward*, infine, si distingue per la flessibilità del collegamento richiesto fra l'ente distributore e i sottoscrittori che, a differenza dello schema *Publish & Subscribe*, non è necessario sia attivo al momento della diffusione delle informazioni richieste. Nel nostro esempio, è probabilmente consigliabile lo schema *Store & Forward* per la fornitura delle norme di interesse a utenti non istituzionali (per esempio, imprese private), per le quali non è realistico ipotizzare un collegamento perennemente attivo con il distributore, mentre per gli utenti istituzionali (gli enti pubblici) lo schema *Publish & Subscribe* è di naturale applicazione, anche alla luce della disponibilità della Rete Unitaria della Pubblica Amministrazione.
E' infine da sottolineare come l'uso del modello basato su oggetti per la rappresentazione unitaria di un sistema distribuito richieda implicitamente la definizione di un *dominio di concetti* sui quali il sistema opera, i cui formati di rappresentazione (*sintassi*) e i cui significati (*semantica*) siano concordati e validi per il complesso del sistema stesso. Questa esigenza è evidente nella necessità di definizione di interfacce fra oggetti non ambigue e che garantiscano l'ottenimento dei servizi attesi dagli oggetti cliente da parte degli oggetti servente. Nella definizione del dominio di concetti di interesse, gli approcci generali che si seguono sono due:

- Nel caso di sistemi distribuiti ad accoppiamento forte, il dominio di interesse è descritto da schemi dei dati unitari ottenuti per integrazione di schemi dei dati esistenti con eventuali aggiunte informative che siano necessarie. Con tale approccio metodologico, si mira a rimuovere le situazioni contraddittorie, arrivando a un progetto del patrimonio informativo complessivo unitario e coerente;
- Nel caso di sistemi distribuiti a accoppiamento debole (sistemi per la cooperazione), il dominio di interesse è ottenuto mediante la somma dei contributi informativi offerti dai diversi soggetti cooperanti. In tale scenario, si mira quindi a determinare congiuntamente il significato delle diverse classi di dati e le strutture logico-fisiche mediante le quali tali dati debbono essere messi a disposizione, demandando a ogni organizzazione partecipante il compito di individuare le modalità mediante le quali fornire i contributi informativi richiesti a partire dallo stato del proprio patrimonio informativo.

2.5. Il concetto di *middleware* nel *Distributed Object Computing*

In generale, con il termine *middleware* si indica un insieme di componenti software che realizzano

una *macchina virtuale* (ovvero un insieme di servizi fra loro coerenti e simulanti il comportamento di un unico elaboratore che fosse progettato per erogarli). La macchina virtuale è messa a disposizione delle applicazioni che la usano mediante chiamate ai servizi da questa offerti. Il *middleware* realizza la macchina virtuale usando servizi offerti da apparati *hardware* e *software* di livello più basso (Figura 10).
Generalmente si distingue fra due tipi di *middleware*: *middleware generalizzato* e *middleware orientato a specifici tipi di servizio.*
Il *middleware generalizzato* è il substrato della maggior parte delle interazioni tra componenti di un sistema informatico; include gli strumenti di comunicazione, i servizi di sicurezza, i servizi di indirizzamento, i meccanismi di sincronizzazione, i servizi di accodamento.
Fra i *middleware orientati a specifiche classi di servizio* ricordiamo:

- *middleware* per l'accesso a basi di dati, come *Open Data Base Connectivity – ODBC* e *Java Data Base Connectivity – JDBC*, che forniscono interfaccie di programmazione per l'accesso a basi di dati da parte di applicazioni *software* in modo indipendente dalle caratteristiche fisiche dei singoli sistemi di gestione dei dati;
- *middleware* per la gestione di transazioni, come quello previsto dal modello *Distributed Transaction Processing – DTP* del consorzio *X/Open*, che specifica le modalità mediante le quali processi diversi possono fra loro collaborare per attuare transazioni distribuite.

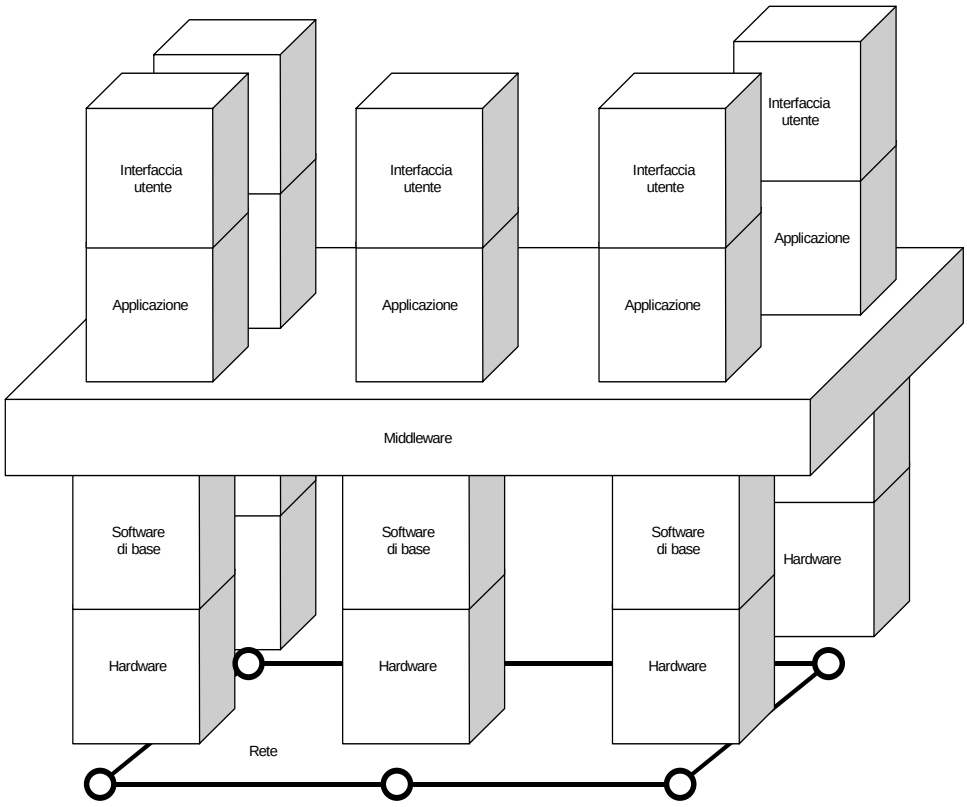

Figura 10 - Il concetto di middleware.

Nell'ambito del *Distributed Object Computing*, i *middleware* che permettono a un insieme di oggetti distribuiti di cooperare su una rete di calcolatori e che quindi costituiscono, nel loro complesso, una *infrastruttura a oggetti distribuiti*, affrontano quattro tematiche generali e distinte (Figura 11):

- il tema dell'accesso alle risorse informative e transazionali esistenti su sistemi *legacy*. I *middleware* che affrontano questo tema usano tecniche di *incapsulamento* di canali dati, di transazioni e di interfacce utente, al fine di presentare tali risorse come oggetti del tutto omogenei con quelli previsti dall'infrastruttura a oggetti distribuiti e fornenti servizi grazie ai

quali le risorse *legacy* sono disponibili nel nuovo contesto tecnologico;

- il tema della produzione, a partire da servizi elementari offerti da oggetti realizzati *ex novo* o da oggetti incapsulanti risorse *legacy*, di nuovi servizi a valore aggiunto, secondo logiche transazionali o procedurali. I *middleware* che affrontano questo tema mettono a disposizione ambienti di realizzazione a oggetti distribuiti completi e danno supporto alla realizzazione di transazioni distribuite o di procedure composte da più passi attivabili mediante scambio di messaggi, mascherando in modo più o meno completo e affidabile la distribuzione e l'eterogeneità della piattaforma hardware e software sottostante;
- il tema della diffusione dei servizi elementari e dei servizi a valore aggiunto a vaste e diversificate popolazioni di utenti. I *middleware* di maggiore popolarità che attualmente affrontano questa tematica sono costituiti dalle tecnologie basate su *Web*, che prevedono l'erogazione dei servizi all'utenza, dotata di stazioni di lavoro complete di *Web browser*, mediante nodi logici di diffusione (*siti Web*), raggiungibili tramite reti *Internet*, *Intranet* o *Extranet*;
- Il tema della rappresentazione e del trattamento dei dati di interesse in modo uniforme, flessibile e indipendente dai meccanismi logico-fisici con i quali le informazioni sono organizzate presso le diverse organizzazioni che collaborano nell'ambito del sistema. In tale contesto stanno assumendo sempre maggiore importanza le tecnologie basate sull'impiego di linguaggi standard di marcatura come *eXtensible Mark up Language – XML* definito dal consorzio *W3C*. Mediante tale approccio standard, è possibile fornire specifiche formali e elaborabili automaticamente delle classi di dati di interesse nell'ambito di interazioni fra oggetti cooperanti, etichettare le corrispondenti istanze e fornire direttive per la loro elaborazione e presentazione all'utenza. La rapida diffusione nell'adozione di questa famiglia di standard da parte di tutte le tecnologie di gestione di risorse, di elaborazione e di diffusione di informazioni e il fiorire di un ricchissimo mercato di strumenti e ambienti su questi basati, rendono *XML* una soluzione estremamente interessante per la realizzazione a costi limitati di sistemi flessibili e che richiedano basso accoppiamento fra le organizzazioni che cooperano per il loro esercizio.

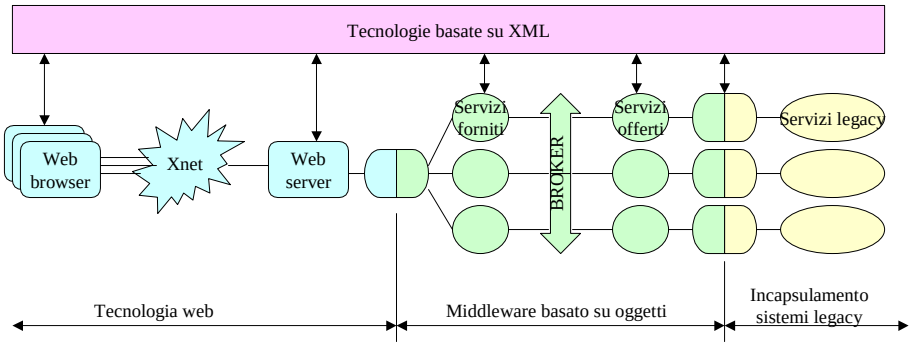

Figura 11: Le tecnologie a oggetti distribuiti

Il ruolo che le quattro classi tecnologiche hanno è quindi il seguente:

- i *middleware* generalizzati a oggetti realizzano la macchina virtuale che permette la progettazione e la realizzazione di un sistema distribuito come costituito da più oggetti applicativi fra loro cooperanti; nello sviluppo applicativo il sistema è modellato secondo un paradigma a oggetti, che promuove la modularità, la riusabilità e la manutenibilità e le problematiche proprie della distribuzione del sistema sulla rete di calcolatori sono affrontate dal *middleware* generalizzato a oggetti e da chi, a livello sistemistico, è chiamato a configurarlo e ottimizzarlo;
- le tecnologie basate su *Web* permettono la diffusione dei servizi informativi e transazionali offerti dagli oggetti di cui il sistema distribuito si compone su reti *Internet*, *Intranet* o *Extranet*. Tali tecnologie costituiscono il mezzo più moderno di diffusione dei servizi all'utenza, sia per la

praticità e la gradevolezza dell'interfaccia utente offerta, sia per l'economicità del supporto all'esercizio;

- le tecnologie di incapsulamento di sistemi *legacy* permettono di ottenere oggetti applicativi corrispondenti ai servizi transazionali (incapsulamento di transazioni host esistenti) o ai servizi informativi (incapsulamento di accesso a archivi di dati esistenti) offerti dai sistemi *legacy*; in questo modo i servizi offerti dai sistemi esistenti possono essere sfruttati nel nuovo contesto tecnologico, valorizzando gli investimenti pregressi e permettendo l'adozione di percorsi di migrazione graduali dalle vecchie alle nuove architetture;
- le tecnologie basate su linguaggi di marcatura consentono, da una parte, il disaccoppiamento del dominio dei concetti di interesse dal modo con il quale questo si materializza nei patrimoni informativi delle organizzazioni cooperanti e, dall'altra, la netta separazione fra i dati oggetto di elaborazione e le politiche con le quali i dati debbono essere presentati all'utenza, diminuendo quindi l'impatto che variazioni di carattere formale, non sostanziale, possono avere sul sistema distribuito stesso.

3. Tecnologie per sistemi a oggetti distribuiti

3.1. Elementi caratterizzanti il middleware in un ambiente di Distributed Object Computing

Nella pluralità e nella differenziazione dei diversi approcci e delle diverse tecnologie che possono essere usate per realizzare una infrastruttura a oggetti distribuiti, è comunque possibile identificare alcuni elementi chiave propri di un ambiente di *Distributed Object Computing* (Figura 12). I paragrafi che seguono ne forniscono una breve descrizione.

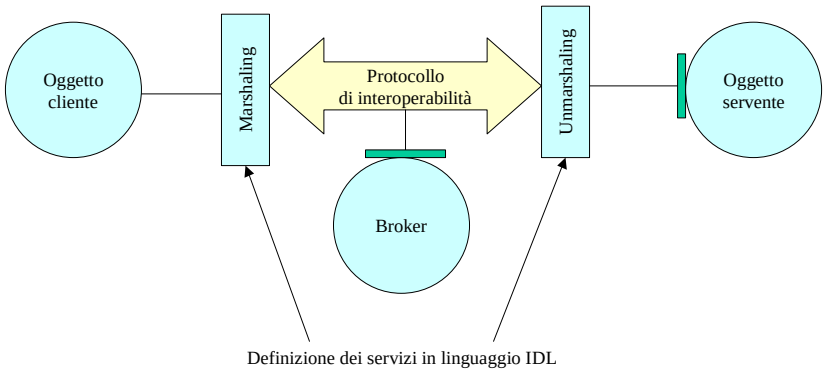

Figura 12 - Concetti fondamentali di una infrastruttura a oggetti distribuiti

3.1.1. Formalismi di definizione degli oggetti

Una infrastruttura a oggetti distribuiti deve mettere a disposizione un insieme completo di strumenti per definire gli oggetti componenti il sistema. In tale contesto, è di particolare importanza la disponibilità di un linguaggio di specifica delle interfacce degli oggetti (*Interface Definition Language - IDL*). *IDL* è un linguaggio dichiarativo per descrivere i servizi che ogni oggetto, tramite la propria interfaccia, mette a disposizione degli altri oggetti del sistema. Definita l'interfaccia, i servizi offerti da ogni oggetto possono essere chiamati ed usati da un qualsiasi altro oggetto, anche se questo ultimo è realizzato in un differente linguaggio di programmazione. Con *IDL* si stabilisce una sorta di contratto tra oggetto cliente e oggetto servente: il servente si impegna a fornire servizi coerenti con l'interfaccia definita, il cliente si impegna a invocare i servizi disponibili in modo conforme con quanto stabilito dall'interfaccia dell'oggetto servente.

3.1.2. Protocolli di interoperabilità

Con il linguaggio *IDL* il progettista definisce le interfacce esposte dagli oggetti al resto del sistema, ma non specifica il modo con cui le richieste e le relative risposte sono trasmesse sulla rete. La

trasmissione delle richieste di servizio e delle corrispondenti risposte è una funzione base di una infrastruttura a oggetti distribuiti ed è assicurata dalla presenza di un *protocollo di interoperabilità* che specifica i tipi di messaggi da scambiare e le sequenze ammissibili. La complessità del protocollo è direttamente legata al potere espressivo del linguaggio di specifica *IDL* e si riflette, in particolare, nei processi di trasformazione delle strutture dati definibili tramite *IDL* in messaggi trasferibili sulla rete (*marshaling)*, e nei processi inversi di trasformazione dei messaggi trasmessi in rete in strutture dati previste dal linguaggio *IDL* (*unmarshaling)*.

3.1.3. Meccanismi di intermediazione fra oggetti (object broker)

Una tematica importante in un ambiente di *Distributed Object Computing* riguarda i meccanismi con i quali gli oggetti clienti di servizi individuano gli oggetti serventi che li mettono a disposizione. Il meccanismo più semplice è quello dell'indirizzamento diretto e esplicito (*direct mail)*, secondo la quale ogni oggetto conosce l'indirizzo esatto (indirizzo *IP* nel caso che il protocollo di rete sia *TCP/IP*) di ogni oggetto al quale voglia inviare richieste di servizio. Tale tecnica è efficiente ma poco flessibile, in quanto la riallocazione di un oggetto servente comporta l'intervento su tutti gli oggetti che usufruiscono dei suoi servizi. La soluzione di tali difficoltà è costituita dall'introduzione di un oggetto intermediario (*Object Request Broker - ORB*), che mantiene un registro dei servizi disponibili e degli indirizzi degli oggetti che li mettono a disposizione. La disponibilità di un ORB presenta alcuni benefici fondamentali: in fase di progettazione è possibile separare la problematica dell'articolazione dell'applicazione in oggetti dalla problematica della distribuzione degli oggetti stessi sulla rete di elaboratori; nell'operatività del sistema è più semplice riconfigurare l'applicazione in caso di aumento del carico o di malfunzionamenti.

3.1.4. Gestione di transazioni

In generale per *transazione* si intende un insieme di operazioni la cui esecuzione fa passare il sistema da uno stato consistente a un nuovo stato consistente. Per esempio una operazione di pagamento di un'imposta comunale con addebito diretto su conto corrente bancario richiede che le operazioni di accredito della somma dovuta sul conto corrente d'appoggio del Comune, di addebito della medesima somma sul conto corrente del contribuente e di registrazione del pagamento nel sistema informatico comunale avvengano tutte con successo, pena il disallineamento dello stato del sistema informatico dalla realtà che esso rappresenta (pagamento avvenuto oppure non avvenuto).
In termini tecnici più precisi [Coulouris et al., 1994] [Goscinski, 1991] [Mullender, 1993] [Simon, 1996], una transazione di un sistema informatico è una unità indivisibile di lavoro che soddisfa le proprietà *ACID* (*Atomicity, Consistency, Isolation, Durability*), portando il sistema da uno stato consistente ad un altro stato consistente. Il significato delle proprietà è:

- *Atomicità*: garanzia che le operazioni previste dalla transazione vengano eseguite tutte, una sola volta, o che non ne venga eseguita nessuna;
- *Consistenza*: garanzia che la transazione porti la base informativa da uno stato consistente ad un nuovo stato consistente;
- *Isolamento*: garanzia che i dati coinvolti in una transazione non siano visibili o impiegabili da un'altra transazione, durante la sua esecuzione;
- *Persistenza*: garanzia che i dati coinvolti in una transazione rimangano disponibili per il tempo utile, anche in presenza di eventi catastrofici (quali la rottura di un disco).

I *sistemi transazionali* sono pertanto, in generale, sistemi di supporto all'esecuzione delle transazioni e hanno il compito di garantire le proprietà di atomicità, consistenza, isolamento e persistenza, la predicibilità dei tempi di risposta, la priorità di alcuni processi su altri e la riservatezza dei dati protetti. Quando le transazioni sono costituite essenzialmente da insiemi ordinati di accessi in lettura e scrittura a basi di dati, il compito di assicurare la loro corretta esecuzione è affidato ai sistemi di gestione di basi di dati. Quando si sia in presenza di logiche

applicative più complesse o quando i carichi elaborativi lo richiedano, si ricorre a componenti infrastrutturali specializzati, chiamati *Transaction Processing Monitor*.
Quando una transazione è decomposta in passi eseguiti da componenti architetturali diversi eventualmente allocati su elaboratori diversi, si è in presenza di una *transazione distribuita*. In tale scenario, è necessario identificare un protocollo con il quale i diversi nodi elaborativi comunichino fra di loro per arrivare alla medesima determinazione di eseguire la parte di lavoro elaborativo loro assegnato o di non eseguirlo per niente, dipendendo l'insuccesso nell'esecuzione della porzione di lavoro assegnato a ogni nodo sia da guasti che nel frattempo possono verificarsi sul nodo o sulle linee di comunicazione, sia dal risultato che ogni elaborazione parziale può produrre. Con riferimento all'esempio del pagamento dell'imposta comunale con addebito diretto in conto, potrebbe verificarsi la necessità di annullamento della transazione complessiva per tanti motivi, quali l'impossibilità momentanea di collegarsi alla banca d'appoggio del contribuente per un guasto sulla linea trasmissiva o l'impossibilità di mandare a buon fine il pagamento richiesto in quanto il numero di conto del contribuente sul quale effettuare l'addebito risulta chiuso, non esistente o bloccato. A livello teorico, si dimostra che non è possibile definire un protocollo di coordinamento non bloccante fra nodi partecipanti a una transazione distribuita che assicuri, in un numero *finito* di passi, l'atomicità della transazione stessa. Un buon compromesso pratico fra le esigenze di integrità del sistema e i carichi di comunicazione fra i partecipanti a una transazione distribuita è costituito dal *protocollo di commit a due fasi* (*2-Phase Commit - 2PC*), secondo il quale un componente svolge il ruolo di *coordinatore* (*Transaction Manager - TM*) mentre gli altri componenti svolgono il ruolo di *gestori di risorse* (*Resource Manager – RM*). Il protocollo prevede una *prima fase* di verifica della disponibilità di tutti i partecipanti a terminare positivamente la transazione. Il coordinatore, nel caso in cui riceva parere negativo da uno dei partecipanti o non riceva alcuna risposta da uno dei partecipanti entro un termine stabilito, invia un messaggio di fallimento della transazione (*rollback*) ai partecipanti residui. Altrimenti si passa alla *seconda fase*, che prevede l'invio da parte del coordinatore di un messaggio di conferma (*commit*) a tutti i partecipanti. Il protocollo *2PC*, le cui modalità di attivazione e uso sono standardizzate nell'ambito del modello *Distributed Transaction Processing – DTP* definito dal consorzio *X/Open*, si presta inoltre facilmente ad un meccanismo di ricorsione per cui la comunicazione tra i partecipanti assume una struttura ad albero, con la radice situata nel coordinatore e le foglie nei partecipanti che eseguono la parte di transazione di loro competenza senza coinvolgere altri sistemi.

Con riferimento all'esempio applicativo di pagamento di imposte comunali con addebito diretto in conto:

- la transazione distribuita è "il pagamento di una imposta comunale con addebito diretto sul conto corrente bancario del contribuente"
- le tre transazioni elementari da comporre sono l'accredito sul conto corrente bancario d'appoggio del Comune, l'addebito sul conto corrente bancario del contribuente, la registrazione nel sistema informatico del Comune del versamento effettuato;
- i tre *gestori di risorse* coinvolti sono, rispettivamente, il sistema informatico della banca d'appoggio del Comune, il sistema informatico della banca del contribuente, il sistema informatico del Comune beneficiario;
- il *coordinatore*, nel caso in oggetto, è collocato nell'ambito del sistema informatico del Comune beneficiario, se si ipotizza che la transazione distribuita descritta sia fornita da tale sistema mediante uno sportello telematico disponibile per i cittadini.

La Figura 13 mostra schematicamente la situazione illustrata.

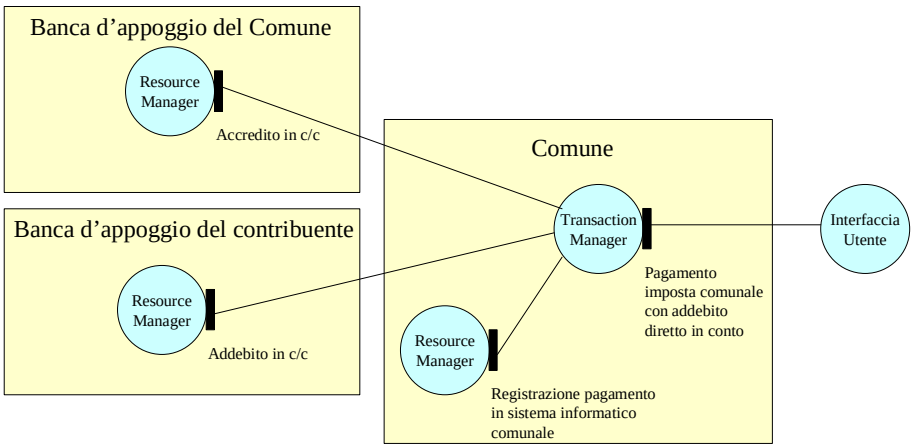

Figura 13 - Esempio di transazione distribuita per il pagamento di imposte comunali

Nelle infrastrutture a oggetti distribuiti, le tematiche di gestione di transazioni sono affrontate da una nuova classe di prodotti di base chiamati *Object Transaction Monitor – OTM*, che integrano le logiche di gestione di oggetti proprie degli *object broker* con i meccanismi di gestione di transazioni proprie dei *transaction monitor*. Attualmente è possibile distinguere fra:

- *Object Transaction Monitor integrati*, ottenuti per integrazione di *object broker* con potenti *Transaction Processing Monitor*;
- *Object Broker comunicanti con Transaction Processing Monitor* attraverso interfacce tipicamente proprietarie;
- *Object Broker con estensioni di gestione transazioni*; tale approccio è sicuramente innovativo ma ne vanno attentamente valutate la reale scalabilità e solidità.

3.1.5. Meccanismi di intermediazione per lo scambio di messaggi fra oggetti (message broker)

Una ulteriore tematica importante in un ambiente di *Distributed Object Computing* riguarda i meccanismi con i quali gli oggetti si scambiano fra loro messaggi per richiedere servizi e per fornire i risultati dell'esecuzione dei servizi richiesti. Negli schemi di comunicazione sincrona fra oggetti, i meccanismi logici di interazione prevedono l'invio *diretto* di ogni messaggio dall'oggetto che lo genera all'oggetto destinatario. Negli schemi di comunicazione asincrona, che non richiedono la contemporanea attivazione dell'oggetto mittente e dell'oggetto destinatario, hanno un ruolo importante gli oggetti *intermediari per lo scambio di messaggi* (*message broker*). A differenza degli *object broker* che mediano fra oggetti clienti e oggetti fornitori di servizi, i *message broker* sono oggetti *gestori di code di messaggi* che assicurano l'instradamento e la consegna dei messaggi che ricevono da parte di oggetti mittenti verso gli oggetti destinatari. In particolare:

- I messaggi sono sequenze di caratteri costituite da una parte di *dati di controllo* e da una parte di *dati di interesse applicativo*. I dati di controllo sono utilizzati dai *message broker*, quelli applicativi dagli oggetti mittente e destinatario. I dati di controllo sono aggiunti ai dati applicativi per gestire la memorizzazione, l'instradamento, la consegna e la sicurezza di ogni messaggio;
- Le code di messaggi possono essere direttamente gestibili a livello applicativo o mascherate dall'infrastruttura, in funzione del modello di gestione di *message broker* messo a disposizione;
- L'infrastruttura mette a disposizione servizi di base per la creazione, la distruzione e la gestione dei message broker e delle code di messaggi corrispondenti.

I *message broker* sono alla base della realizzazione di sistemi a oggetti distribuiti basati su *logiche procedurali*, dove ogni servizio complesso è rappresentato come una *sequenza di passi* eseguiti da oggetti diversi e che viene attuata mediante messaggi di attivazione inviati a code di messaggi intermedie. La decomposizione di ogni procedura in una sequenza ordinata di passi governata da *message broker* assicura la tracciabilità e consente grande flessibilità da parte di tutti coloro che

partecipano al suo completamento.
Le infrastrutture tecnologiche specializzate nell'approccio a code di messaggi sono chiamate *Message Oriented Middleware – MOM* e sono promosse dalle associazioni *MOMA* (*MOM Association*) e *BQM* (*Business Quality Management*). La comunicazione tra gli oggetti mediata da una o più code di messaggi permette un minore accoppiamento fra gli oggetti cooperanti rispetto al caso dell'uso di comunicazioni dirette. Questa caratteristica sta favorendo la rapida diffusione delle tecnologie *MOM* specialmente in contesti cooperativi.

3.1.6. Servizi di supporto

Le piattaforme per il *Distributed Object Computing* più complete offrono *servizi di supporto* che rendono più semplice e veloce lo sviluppo di applicazioni distribuite.
Un primo insieme di servizi di supporto riguarda la *gestione del ciclo di vita* di un oggetto, dal momento in cui viene creato per la prima volta al momento in cui deve cessare di esistere.
Un ulteriore insieme di servizi di particolare importanza è costituito dai *servizi di trasparenza* che realizzano la macchina a oggetti virtuale mascherando ai progettisti e agli sviluppatori applicativi tutte o parte delle complessità indotte dalla distribuzione. I servizi di trasparenza principali che possono essere resi disponibili da una infrastruttura a oggetti distribuiti sono:

- *trasparenza di accesso*, ovvero reperimento e raggiungimento, da parte dell'oggetto cliente, di un oggetto che offre un servizio;
- *trasparenza di replica*, ovvero possibilità di configurare e variare le modalità di replica degli oggetti senza che ciò determini necessità di variazione nella realizzazione e nell'uso del sistema;
- *trasparenza di distribuzione*, ovvero invarianza delle modalità di realizzazione delle richieste di servizio, sia che queste siano offerte da oggetti locali o remoti;
- *trasparenza di migrazione*, ovvero possibilità di cambiare dinamicamente l'allocazione degli oggetti sulla rete;
- *tolleranza ai guasti*, ovvero continuità di erogazione dei servizi previsti, eventualmente secondo modalità degradate predefinite, anche in presenza di malfunzionamenti in uno o più nodi della rete e della conseguente indisponibilità degli oggetti;
- *trasparenza di linguaggio*, cioè possibilità di realizzare gli oggetti del sistema con diverse metodologie e diversi ambienti di sviluppo;
- *trasparenza rispetto all'eterogeneità tecnologica*, ovvero omogeneità applicativa su reti costituite anche da macchine eterogenee nell'hardware e nel software di base.

Altri esempi ancora sono rappresentati dai *servizi di sicurezza*, che permettono di definire e applicare politiche di sicurezza nell'uso del sistema e delle risorse da parte degli utenti, e dai *servizi di monitoraggio e gestione degli oggetti*, che permettono la supervisione dell'esercizio del sistema, la sua gestione, ottimizzazione e *tuning* attraverso il controllo e l'intervento su parametri di funzionamento degli oggetti legati ai servizi di trasparenza che l'infrastruttura offre.
Ai servizi di supporto di base, indipendenti dal contesto applicativo, si possono aggiungere servizi di supporto alla realizzazione di sistemi per specifici domini applicativi, quali la sanità, le banche, la grande distribuzione. Tale supporto è realizzato mediante la disponibilità di oggetti configurabili e rispondenti a logiche di *business* generali del dominio applicativo di interesse. L'insieme di tali oggetti, denominato spesso *framework applicativo*, può essere fornito in modo integrato con i servizi infrastrutturali di base, costituendo in tale caso un *Component Transaction Monitor – CTM*.
La ricchezza dell'insieme dei servizi di supporto offerti è evidentemente elemento critico per permettere il rapido sviluppo di applicazioni distribuite flessibili e estendibili.

3.2. Le proposte tecnologiche di mercato principali

Le aspettative che si ripongono in una infrastruttura ad oggetti distribuiti riguardano la disponibilità di una macchina virtuale unitaria che permetta l'esecuzione del sistema ad oggetti mascherando ai realizzatori e agli utenti tutte le problematiche riguardanti la distribuzione degli oggetti su una rete.

Attualmente sono disponibili sul mercato una serie di soluzioni architetturali e tecnologiche che si propongono come infrastrutture per il *Distributed Object Computing* e che rispondono in modo più o meno soddisfacente all'aspettativa sopra affermata [Umar2, 1997]. Nei paragrafi che seguono sono descritte sinteticamente le principali proposte tecnologiche presenti sul mercato:

- La proposta *Common Object Request Broker Architecture - CORBA* promossa da *Object Management Group - OMG*: *CORBA* è uno standard industriale per la realizzazione di infrastrutture a oggetti distribuiti operanti su reti e sistemi eterogenei;
- La proposta *Windows Distributed Network Architecture – DNA* di Microsoft: è la soluzione proposta da Microsoft per la comunicazione fra componenti software che operano in ambiente distribuito e intorno alla quale è maturata una vastissima offerta di soluzioni di mercato;
- La proposta *Enterprise Java Beans*: questa proposta identifica le caratteristiche di un *Component Transaction Monitor – CTM*, ovvero di una infrastruttura a oggetti distribuiti arricchita con un insieme normalizzato di oggetti specifici (componenti) che risolvono non solo problematiche di tipo generale, quali quelle che possono incontrarsi nella logica di presentazione di dati e di interazione con l'utenza, ma anche problematiche relative a logiche di *business* legate a specifici domini applicativi;
- La famiglia delle *tecnologie Web* e delle *tecnologie che sfruttano linguaggi di marcatura*: tale classe di tecnologie fa riferimento a un modello architetturale normalizzato secondo il quale l'utente, dotato di un *Web Browser* di caratteristiche *standard*, accede ai servizi messi a disposizione da un *Web Server* presentante una interfaccia *standard*, sia in termini di protocollo di comunicazione (tipicamente *Hyper Text Transfer Protocol – HTTP* su protocollo di trasporto *TCP/IP*), sia in termini di formato dei dati trasmessi (tipicamente un linguaggio di marcatura quale *Hyper Text Markup Language – HTML*).

3.3. OMG CORBA

Common Object Request Broker Architecture - CORBA è la specifica tecnica pubblica per lo scambio di messaggi tra oggetti su una rete di sistemi eterogenei promossa da *Object Management Group - OMG*, un consorzio che comprende istituti di ricerca, università, grandi utenti (banche, amministrazioni pubbliche, società di telecomunicazioni) e praticamente tutti i maggiori produttori di tecnologie informatiche. Questa specifica standard si inserisce nel contesto architetturale ad oggetti, mostrato in Figura 14, chiamato *Object Management Architecture - OMA*, basato sui seguenti componenti principali:

- Un *Object Request Broker - ORB*, che assicura i meccanismi di base necessari per la registrazione dei servizi offerti dai diversi oggetti e per la loro invocazione ed esecuzione; è il componente infrastrutturale che consente agli oggetti di fare richieste e ricevere risposte in modo trasparente in ambiente distribuito. Per tale motivo è anche visto come il canale (*bus*) attraverso il quale gli oggetti applicativi colloquiano;
- i *Servizi di base* (*Common Object Services - COS*), che assicurano le funzionalità necessarie per l'attivazione e la gestione operativa dei sistemi applicativi ad oggetti;
- i *Servizi Orizzontali*, ovvero servizi per risolvere problematiche ricorrenti e generali nelle applicazioni (ad esempio l'interfaccia utente);
- i *Servizi Verticali*, che facilitano nello sviluppo in particolari domini applicativi.

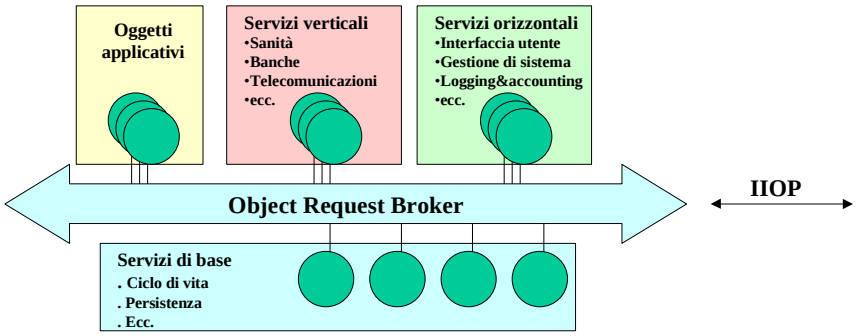

Figura 14 – Object Management Architecture

L'architettura proposta è disponibile su piattaforme hardware e software diverse. Gli oggetti possono essere realizzati con diversi linguaggi di sviluppo e dialogano fra loro e con il *broker* mediante interfacce definite nel linguaggio dichiarativo *IDL*. Il protocollo di interazione tra oggetti *Internet Inter Object Protocol - IIOP* permette il dialogo fra *broker* diversi.
Lo standard *CORBA*, che mira all'indipendenza da specifiche piattaforme e si basa sul concetto di *broker* come intermediario nel dialogo fra oggetti, si propone come soluzione nei casi in cui si debbano federare sistemi informatici di organizzazioni diverse, chiamati a cooperare mantenendo ognuno elevati livelli di autonomia. I suoi punti di forza sono la disponibilità di una architettura stabile e completamente definita, sia per quanto riguarda le caratteristiche dell'*object broker*, sia per quanto riguarda i servizi complementari, la sostanziale indipendenza della specifica dalle piattaforme, la disponibilità di realizzazioni commerciali della specifica per molte piattaforme hardware/software di mercato, la piena aderenza ai concetti dei modelli basati su oggetti, sia per gli approcci progettuali proposti, sia per i linguaggi di programmazione supportati. *CORBA* gode inoltre di un vasto supporto da parte dei numerosi produttori che aderiscono all'*OMG*.

3.3.1. Il modello a oggetti

Il paradigma di sviluppo implicitamente promosso da *CORBA* è quello *Plug&Play*: un oggetto, una volta che si sia dichiarato al *broker* (*plug*) è in grado di interoperare (*play*) con qualunque altro oggetto presente nel sistema. La realizzazione di un oggetto comporta la definizione, tramite il linguaggio IDL, dell'interfaccia di servizi che l'oggetto mette a disposizione del resto del sistema e la scrittura del codice che specifica il comportamento dell'oggetto in rapporto alle richieste di servizio che gli pervengano da altri oggetti.
Al concetto di richiesta di servizio che un oggetto può formulare sono associate due informazioni fondamentali per determinare l'oggetto servente cui richiedere il servizio e il servizio specifico richiesto. Ogni servizio ha una specifica (*signature*) che identifica i parametri richiesti, i risultati restituiti, le eccezioni nei casi di terminazione anomala.

3.3.2. Object Request Broker

L'*Object Request Broker - ORB* è l'oggetto infrastrutturale previsto dall'architettura *CORBA* per consentire il dialogo fra gli oggetti, rendendo trasparente al livello applicativo i meccanismi fisici impiegati per garantire tale comunicazione. La logica secondo la quale un *object request broker* opera è mostrata in Figura 15.

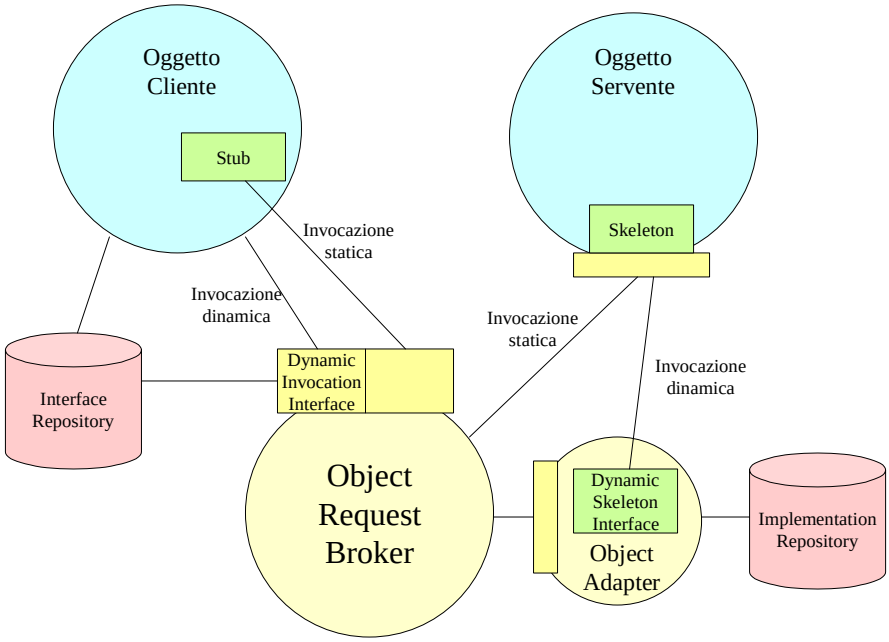

Figura 15 - I componenti di un broker CORBA

In fase realizzativa, i servizi che sono messi a disposizione da ogni oggetto al resto del sistema e che costituiscono la sua interfaccia sono specificati mediante il linguaggio dichiarativo *IDL*. La compilazione della specifica *IDL* dell'interfaccia di un oggetto produce uno scheletro (*skeleton*) dell'oggetto, nel linguaggio sorgente scelto (tipicamente *C*, *C++* o *Java*), che il programmatore è chiamato a arricchire con la codifica di ogni servizio. Inoltre, tale compilazione produce la registrazione della specifica dell'interfaccia in un archivio privato dell'*Object Request Broker* chiamato *Interface Repositor*y e la registrazione della disponibilità di ogni servizio offerto dall'interfaccia in un altro archivio chiamato *Implementation Repository*. La compilazione della specifica *IDL* produce infine un modulo (*stub*) che deve essere richiamato da ogni oggetto cliente intenzionato a usare i servizi definiti dalla specifica *IDL* stessa. I moduli *skeleton* e *stub* costituiscono quindi il meccanismo usato per rendere, a livello programmativo, la chiamata di un servizio *CORBA* del tutto simile alla chiamata di un modulo locale con un normale linguaggio di programmazione, nascondendo tutte le complessità legate all'ambiente distribuito.
Il meccanismo descritto di invocazione di un servizio è quello più semplice, in quanto prevede che gli oggetti cliente sappiano di dovere invocare specifici servizi offerti dall'infrastruttura *al tempo della progettazione*. Per tale motivo si parla di *meccanismo di invocazione statico*. L'infrastruttura *CORBA* mette a disposizione anche *meccanismi di invocazione dinamica*. Nel caso in cui un oggetto decida, *al tempo di esecuzione*, di avere bisogno di un servizio non previsto al tempo della progettazione, usa una *interfaccia di invocazione dinamica* (*Dynamic Invocation Interface - DII*) mediante la quale può consultare l'*Interface Repository*, contenente i servizi registrati presso l'*Object Request Broker*, e estrarre da esso tutti i metadati descriventi il servizio di interesse, essendo in fine in grado di invocarlo in modo corretto. *CORBA* consente inoltre a un oggetto servente di dichiarare i propri servizi non solo, staticamente, al tempo di compilazione delle specifiche IDL ma anche, dinamicamente, durante l'esecuzione del sistema. Il meccanismo disponibile per questo scopo è basato sulla disponibilità di una interfaccia dinamica per gli oggetti serventi (*Dynamic Skeleton Interface*) che consente la risoluzione delle richieste verso gli oggetti serventi che non hanno *skeleton* ottenuti tramite la compilazione di specifiche *IDL*. Il componente infrastrutturale che media il dialogo fra i gli oggetti serventi dinamici e l'*Object Request Broker* è chiamato *Object Adapter*, che quindi aggiorna l'*Implementation Repository* dei servizi disponibili in ogni istante nel sistema.

3.3.3. I Servizi CORBA

I servizi di sistema previsti dall'architettura *CORBA*, accessibili tramite interfacce definite in *IDL*, completano le funzionalità infrastrutturali assicurate dall'*Object Request Broker*. I servizi di sistema principali individuati dagli standard *OMG* riguardano:

- Il ciclo di vita degli oggetti (*Life Cycle Service*): forniscono strumenti per la creazione, la copia, la variazione e l'eliminazione di un oggetto;
- La persistenza degli oggetti (*Persistence Service*): forniscono strumenti per memorizzare in modo permanente lo stato degli oggetti, rendendo trasparente il meccanismo fisico di archiviazione usato;
- La gestione di uno spazio dei nomi logici associati agli oggetti (*Naming Service*); forniscono strumenti per l'associazione di nomi logici agli oggetti e per il loro uso in modo consistente in tutto il sistema;
- La gestione di eventi associati alle azioni compiute dagli oggetti (*Notification Service*): forniscono un insieme di strumenti per la comunicazione asincrona secondo il modello *Publish&Subscribe*;
- La gestione di transazioni distribuite (*Transaction Service*): mettono a disposizione i meccanismi propri del *protocollo di commit a due fasi* (*Two-Phase Commit)* fra oggetti;
- La disponibilità di un linguaggio di interrogazione sullo stato degli oggetti (*Query Service*): il linguaggio disponibile è un sovrainsieme del linguaggio *SQL* basato sulla specifica *SQL3* e sulla specifica *Object Query Language – OQL* definita dall'*Object Database Management Group - ODMG*;
- La gestione dell'uso degli oggetti (*Licensing Service*): mettono a disposizione meccanismi per la misurazione dell'uso degli oggetti, per il controllo delle licenze e per i relativi addebiti;
- La gestione del parametro tempo (*Time Service*): consentono la sincronizzazione temporale fra i diversi componenti di un sistema distribuito e la definizione e la gestione di eventi legati a precisi istanti temporali;
- La gestione della sicurezza (*Security Service*): mettono a disposizione una infrastruttura completa per l'accesso sicuro a oggetti distribuiti, usando diversi meccanismi di autenticazione, di riservatezza e di delega.

3.3.4. Interoperabiltà tra Object Request Broker

L'interoperabilità tra realizzazioni dell'architettura *CORBA* da parte di fornitori diversi è un elemento fondamentale del modello di riferimento *OMA*. Il meccanismo che, sulla carta, è chiamato a garantire tale interoperabilità è il protocollo di comunicazione fra *Object Request Broker* diversi denominato *Internet Inter-ORB Protocol – IIOP*. Il pieno supporto di questo protocollo da parte degli *Object Request Broker* di mercato permetterebbe a oggetti clienti di invocare trasparentemente servizi forniti sia da oggetti serventi registrati presso l'*ORB* locale, sia da oggetti serventi registrati presso *ORB* remoti ma in comunicazione, via *IIOP*, con l'*ORB* locale. Purtroppo la realtà di mercato non è pienamente in linea con quanto previsto dalla specifica dell'*OMG* per cui, se è vero che i prodotti disponibili sono nominalmente aderenti alle specifiche *CORBA* e quindi sono in grado di interoperare, di fatto i meccanismi di base capaci di usare in modo trasparente il protocollo *IIOP* sono molto limitati e cambiano da caso a caso. Capita, per esempio, che i meccanismi di invocazione dinamica di servizi non funzionino correttamente, che il nome con il quale denotare un oggetto remoto non sia ottenibile tramite i servizi forniti dall'infrastruttura o che alcune strutture dati non siano correttamente gestite nel passaggio da un *ORB* a un altro. In pratica quindi, a meno che l'interoperabilità tra *ORB* diversi non sia effettivamente provata per le casistiche di interesse, è consigliabile attualmente sviluppare le applicazioni distribuite utilizzando l'*ORB* di un unico fornitore.

3.4. Microsoft Windows DNA

Il modello di cooperazione fra oggetti proposto da Microsoft [Voth et al., 1998] è caratterizzato dall'integrazione in un unico quadro tecnologico (Figura 16) di soluzioni per la memorizzazione dei dati, per l'esecuzione di attività di gestione di risorse e per la fornitura di servizi agli utenti. Gli oggetti possono essere prodotti con svariati ambienti e linguaggi di sviluppo e comunicano tra loro mediante un protocollo proprietario Microsoft.

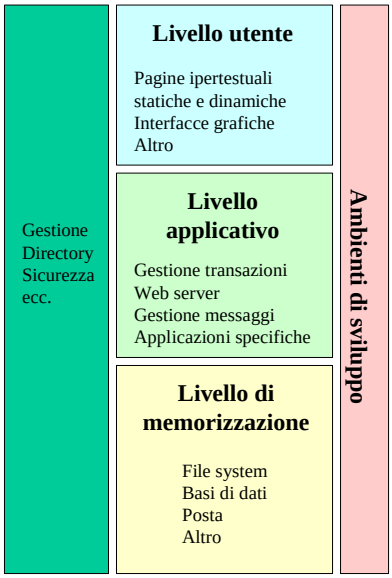

Figura 16 - Architettura Microsoft Windows DNA

Windows DNA è sia la specifica di un modello a componenti (*Component Object Model + - COM+*) sia la realizzazione di una infrastruttura composta da prodotti (la famiglia di prodotti *.NET Enterprise Server* basati sul nuovo sistema operativo *Microsoft Windows 2000*) e da ambienti per lo sviluppo di componenti che li sfruttino. Il risultato è la disponibilità di una soluzione infrastrutturale complessiva, di strumenti di sviluppo e di componenti riusabili fortemente integrati fra loro; tale disponibilità rende più semplice e meno costoso lo sviluppo di sistemi e prototipi a oggetti distribuiti rispetto a quanto è possibile ad oggi fare con l'approccio CORBA. Di contro, *Windows DNA* si presenta come una soluzione proprietaria e vincolante nelle scelte degli apparati *hardware* e *software* di base sui quali può operare.

3.4.1. Il modello a oggetti

In *COM+/Windows DNA* gli oggetti, detti anche *componenti*, comunicano tra loro e con l'infrastruttura attraverso insiemi di funzioni denominate *interfacce*; una interfaccia determina quindi il *contratto* tra il componente che la mette a disposizione ed i componenti clienti che la invocano. Un componente può offrire più interfacce e la stessa interfaccia può essere offerta da più componenti. Una *classe* è la *specifica di un insieme di interfacce* (definizione più pragmatica di quella classica dei modelli orientati agli oggetti per i quali una classe è *l'astrazione di un concetto*) e ogni oggetto (componente) è un'istanza di una specifica classe.
COM+/Windows DNA non supporta il costrutto di *ereditarietà* fra classi ma introduce meccanismi alternativi orientati al riuso. Con il meccanismo di *Contenimento* (Figura 17) un oggetto (*componente esterno*) che voglia riusare il codice di un altro oggetto (*componente interno*) si comporta a tutti gli effetti come un oggetto cliente. Di fatto, il componente interno non ha alcuna percezione della differenza che esiste tra l'essere invocato nell'ambito di una tecnica di contenimento e l'essere invocato in una qualunque altra situazione da un normale oggetto cliente. In questo scenario, il componente esterno realizza le proprie interfacce usando le interfacce del componente interno. Il componente esterno può anche esporre una interfaccia del tutto analoga a quella offerta dal componente interno, realizzando tutti i relativi servizi (chiamati anche *metodi*) con

la semplice invocazione dei corrispondenti metodi del componente interno. Il componente esterno può inoltre specializzare l'interfaccia aggiungendo logica applicativa prima e dopo le invocazioni ai metodi del componente interno.

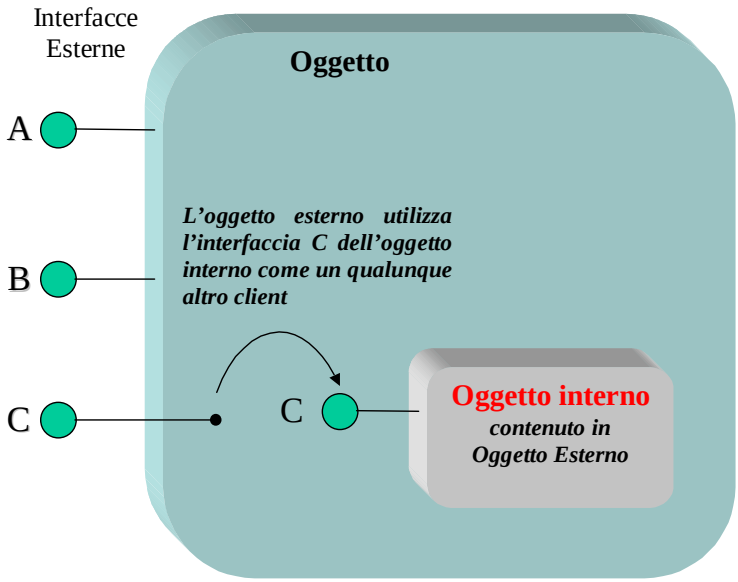

Figura 17 - Il contenimento in COM+/Windows DNA

Il meccanismo di *Aggregazione* (Figura 18) è una variante del meccanismo di *Contenimento* e consiste nel realizzare un oggetto aggregato che espone una interfaccia ottenuta pubblicando i metodi appartenenti all'interfaccia del componente interno. In tale modo, il componente esterno espone direttamente l'interfaccia del componente interno come se fosse propria. Gli oggetti clienti percepiscono l'interfaccia del componente esterno come nativa e non hanno alcuna conoscenza del fatto che essa è ottenuta per aggregazione, anche se le invocazioni dirette ai metodi di quell'interfaccia sono in realtà gestite direttamente dal componente interno. Il vantaggio fondamentale dell'aggregazione rispetto al contenimento consiste nel fatto che il componente esterno non è costretto a realizzare nuovamente un'interfaccia già fornita da un componente interno. Tuttavia, il componente esterno non è in grado di specializzare nessuno dei metodi di quell'interfaccia.

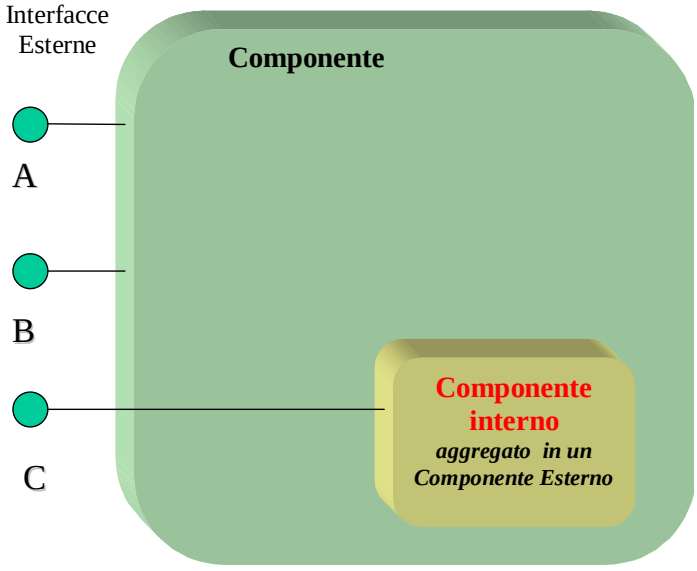

Figura 18 - L'aggregazione in COM+/Windows DNA

Un esempio schematico dell'uso di questi due meccanismi è il seguente. Si supponga di avere a disposizione la classe Persona dotata del metodo CalcoloReddito() e di dovere definire una classe Donna, sottoclasse di Persona. Usando un classico modello orientato agli oggetti, la classe Donna sarebbe una specializzazione della classe Persona e erediterebbe da questa tutti gli attributi e tutti i metodi. Con il modello *COM+/Windows DNA* è possibile applicare una delle soluzioni seguenti:

- Usando il meccanismo di *contenimento*, la classe Donna è dotata di un attributo privato di tipo Persona e di un metodo CalcolaReddito(); il codice del metodo CalcolaReddito() della classe Donna invoca il corrispondente metodo CalcolaReddito() dell'istanza privata della classe Persona referenziata dal valore dell'attributo privato di tipo Persona;
- Usando il meccanismo di *aggregazione*, la classe Donna aggrega la classe Persona; in questo modo ogni istanza della classe Donna contiene implicitamente una istanza della classe Persona ed ogni volta che viene invocato dall'esterno il metodo CalcolaReddito() su una istanza della classe Donna, è attivato trasparentemente il metodo CalcolaReddito() sull'istanza aggregata della classe Persona.
- E' infine da notare che, qualunque sia il meccanismo usato, il modello *COM+/Windows DNA* richiede l'allocazione di due oggetti, uno della classe *Donna* ed uno della classe *Persona*.

3.4.2. I servizi infrastrutturali

L'infrastruttura *COM+/Windows DNA* offre i seguenti servizi di base:

- Servizi di creazione di oggetti clienti e serventi: con particolare riguardo agli oggetti serventi, l'infrastruttura offre servizi per la pubblicazione delle interfacce fornite;
- Servizi di localizzazione degli oggetti serventi (*implementator locator)*, che permettono di determinare l'oggetto servente che fornisce l'interfaccia richiesta e la sua localizzazione;
- Gestione trasparente delle chiamate a servizi forniti da oggetti serventi remoti, secondo uno schema di invocazione sincrona (*Remote Procedure Call – RPC*) che prevede l'uso di componenti di supporto lato cliente (*proxy*) e lato servente (*stub*) generati automaticamente all'atto della compilazione della specifica delle interfacce in linguaggio *IDL*;
- Meccanismi di controllo dell'uso delle risorse di sistema da parte degli oggetti applicativi.

Nell'ambito di *COM+/Windows DNA*, *Microsoft Transaction Server* (*MTS*) è il componente dell'architettura distribuita proposta da Microsoft classificabile come *Component Transaction Monitor*. Infatti *MTS*:

- assolve alle funzioni di *Object Request Broker*;
- agisce come processo "surrogato" per gli oggetti *COM*;
- agisce come *Transaction Processing Monitor*.

MTS, come *broker*, consente la comunicazione trasparente tra oggetti, siano essi fra loro locali o remoti, evitando che l'oggetto cliente debba localizzare gli oggetti serventi ai quali chiedere i servizi necessari. Per tale scopo, *MTS* intercetta la creazione di oggetti *COM*, agendo da processo surrogato per gli oggetti stessi. In questo modo, *MTS* è in grado di collocarsi fra gli oggetti clienti e gli oggetti serventi e di controllare tutte le chiamate di servizio fatte dai primi ai secondi.

MTS, come *processo surrogato*, gestisce il contesto di esecuzione degli oggetti COM da lui mediati e gestisce le problematiche di attivazione di processi leggeri (thread) nel caso in cui gli oggetti svolgano funzioni di serventi chiamati concorrentemente da più oggetti clienti. Inoltre, MTS gestisce il ciclo di vita degli oggetti, le risorse elaborative e di memorizzazione usate dagli oggetti e la sicurezza delle chiamate effettuate

Infine *MTS*, come *Transaction Monitor*, gestisce l'esecuzione di transazioni distribuite fra più oggetti cooperanti.

COM+/WindowsDNA, nato con un approccio di comunicazione fra oggetti di tipo sincrono, si sta evolvendo verso un modello di comunicazione asincrono ottenuto attraverso:

- L'adozione di protocolli di comunicazione a messaggi tipo *HyperText Transfer Protocol - HTTP* e *Microsoft Messaging Queues - MSMQ* per le invocazioni di servizi remoti. Le infrastrutture

abilitanti questo tipo di dialogo sono quelle di gestione messaggi, in sostituzione di quelle basate sul modello di comunicazione sincrona *Remote Procedure Call – RPC*;

- L'adozione di *XML* come linguaggio primario per la definizione delle interfacce di servizi fornite dagli oggetti, in sostituzione di *IDL*.
- L'enfasi sulla cooperazione fra applicazioni attraverso la nuova proposta architetturale *BizTalk*. *BizTalk*, basato su *XML* come linguaggio standard per lo scambio di informazioni, si propone di realizzare la cooperazione fra applicazioni, viste come entità *lascamente accoppiate*, mediante lo scambio di messaggi in formato *XML*. Secondo questo nuovo approccio, le applicazioni che vogliano cooperare non debbono essere basate necessariamente su un modello ad oggetti comune, non debbono necessariamente concordare su uno stesso protocollo di comunicazione né debbono necessariamente essere scritte con il medesimo linguaggio di programmazione. Per consentire la cooperazione, è sufficiente che le applicazioni abbiano la capacità di leggere e produrre messaggi in formato *XML*, in modo nativo o attraverso *adattatori* (*adapter*), lasciando all'ambiente *BizTalk* la gestione delle eterogeneità di protocollo e di attivazione fra le applicazioni.

3.5. Enterprise Java Beans

3.5.1. Il linguaggio Java

Java è un linguaggio orientato agli oggetti originariamente nato per la realizzazione di semplici applicazioni di pilotaggio di apparati e di periferiche intelligenti [SUN, 1995]. La novità di *Java* consiste nel fatto che un programma non viene compilato in linguaggio macchina, ma in un codice virtuale, detto *Java bytecode*; tale codice virtuale è successivamente interpretato, a tempo di esecuzione, da una macchina virtuale, chiamata *Java Virtual Machine* - *JVM*, avente specifiche di funzionamento uguali per tutti i sistemi, che si occupa di effettuare la traduzione di *Java bytecode* nel codice macchina del sistema ospite. Rendendo disponibile per ogni tipo di sistema fisico una *Java Virtual Machine*, si ottiene un approccio architetturale indipendente dall'hardware e dal sistema operativo usati. In questo modo si ottiene la portabilità del software da una piattaforma ad un'altra, cioè la cosiddetta *WORA* (*Write Once, Run Anywhere*) *portability* che è alla base del successo di *Java* non solo come linguaggio, ma come vero e proprio approccio allo sviluppo di sistemi distribuiti.
Il modello *Java* prevede la realizzazione di oggetti eseguiti in modalità interpretata su *Java Virtual Machine* e che dialogano fra loro mediante protocollo standard *Java Remote Method Invocation – JRMI* (Figura 19).

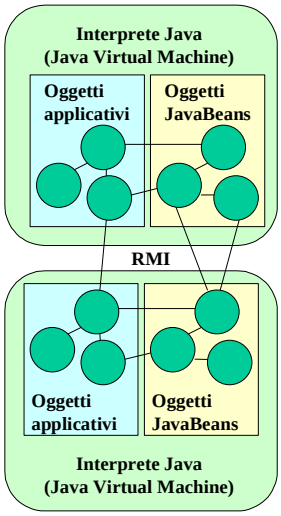

Figura 19 - Approccio Java

Gli oggetti hanno la capacità di *migrare* da una *Java Virtual Machine* ad un'altra collegata a questa in rete. La capacità di migrazione degli oggetti è alla base di soluzioni architetturali estremamente dinamiche, che possono prevedere la visita di reti di grandi dimensioni come *Internet* da parte di *agenti* che svolgono autonomamente particolari funzioni (per esempio l'indicizzazione dei contenuti informativi presenti sui siti Web), l'arricchimento funzionale dei nodi elaborativi in funzione delle richieste dell'utenza, il bilanciamento dei carichi elaborativi sui nodi fisici disponibili. Questo modello appare quindi particolarmente adatto in scenari di distribuzione su reti *Internet*, *Intranet* o *Extranet* di servizi a insiemi aperti di utenti non predefiniti, per i quali non è possibile imporre vincoli sulle piattaforme usate.

3.5.2. Gli oggetti Java

Java introduce un nuovo approccio architetturale particolarmente adatto alla distribuzione di applicazioni sul *Web*. Questo approccio si basa sulla realizzazione di piccoli oggetti *software* chiamati *applet* che possono essere scaricati dalla Rete per essere eseguiti da un *Web browser* che includa, come attualmente è la norma, una *Java Virtual Machine*. In questo modo è possibile distribuire sulla Rete oggetti *software* eseguibili sulle stazioni di lavoro degli utenti e eventualmente capaci di dialogare, attraverso la Rete, con siti *Web* appositamente progettati. Gli applet sono in grado di gestire una interfaccia utente con la quale possono presentare informazioni e possono acquisire dati da usare o ai fini di una migliore gestione delle risorse di elaborazione locali o ai fini dell'innesco di servizi da parte di siti *Web* raggiungibili con la Rete.
E' da sottolineare che concetti sostanzialmente analoghi a quelli legati agli *applet Java* si ritrovano nell'offerta Microsoft con la tecnologia *ActiveX*, che richiede che i *Web browser* usati siano compatibili con l'approccio *Windows DNA*.

3.5.3. Enterprise Java Beans

Enterprise Java Beans - EJB è una specifica, promossa da Sun Microsystems con il supporto di altri fornitori di tecnologie, che definisce quali debbano essere i servizi da fornire nell'ambito di tematiche di interesse generale (come la gestione di transazioni, la gestione di code di messaggi, la gestione dell'interazione con l'utente, l'accesso ai dati) e come tali servizi possano essere invocati dagli oggetti clienti. *Enterprise Java Beans* introduce il concetto di *riuso* di oggetti applicativi nell'ambito dell'approccio *Java* e mira a limitare la complessità e l'impegno economico e temporale necessario per realizzare l'insieme degli oggetti serventi di cui ha bisogno un qualunque sistema. L'effettivo successo e utilità di questo approccio dipende dal concreto impegno, da parte dei fornitori, a realizzare gli oggetti serventi in accordo con le specifiche *Enterprise Java Beans* concordate. La Figura 20 elenca a titolo esemplificativo alcuni tipi di servizio previsti.

Tipo di servizio	Descrizione
Remote Method Interface - RMI	Crea interfacce per la chiamata a metodi remoti
Java Interface Definition Language - JIDL	Crea interfacce per il supporto di architetture CORBA
Java Messaging Service - JMS	Fornisce supporto a comunicazioni asincrone con diversi meccanismi, quali il Publish&Subscribe
Java Transaction Service - JTS	Fornisce supporto alla gestione di transazioni distribuite
Java Data Base Access - JDBC	Fornisce una interfaccia unica per l'accesso a basi di dati relazionali

Figura 20 - Esempi di servizi Enterprise Java Beans

L'organizzazione della specifica *Enterprise Java Beans* si basa su alcuni concetti fondamentali:

- *Componente*: *software* riusabile che può essere combinato con altri componenti per produrre sistemi complessi e specializzati; si distinguono essenzialmente due tipi di componenti, clienti e serventi, eseguiti entrambi all'interno di un *contenitore* (*container*); l'invocazione dei servizi, l'accesso e la manipolazione dei dati sono realizzati mediante *interfacce normalizzate* rispondenti alla specifica *Enterprise Java Beans*;

- *Contenitore* (*Container*): *software* che fornisce un contesto applicativo per uno o più componenti, garantendo i servizi di gestione e di controllo degli stessi; quando un cliente invoca il servizio di un componente servente, il *contenitore* alloca le risorse necessarie alla sua esecuzione (memoria, spazio di processo), inizializza il componente servente e gestisce tutte le sue interazioni con l'esterno; un componente cliente normalmente è eseguito all'interno di un contenitore di tipo visuale (una *form*, una pagina *web*), mentre un componente servente è eseguito all'interno di un contenitore fornito da un oggetto servente applicativo;
- *Java Bean*: è una classe *Java* specializzata che può essere usata e modificata durante lo sviluppo di applicazioni. Caratteristica importante dei *bean* è la possibilità di personalizzarne alcuni aspetti usando attributi particolari dei *bean* stessi che funzionano da *parametri di configurazione*; questa tecnica permette di sviluppare componenti altamente riusabili, il cui comportamento specifico in un particolare sistema può essere determinato non modificandone la codifica ma variando i valori dei parametri di configurazione;
- *Enterprise Java Beans*: concettualmente sono analoghi ai semplici *JavaBean* e ne possiedono tutte le caratteristiche di componibilità e di configurabilità. Con *JavaBean* si indica normalmente un componente usato nell'ambito di oggetti clienti, mentre con *Enterprise JavaBean* si indica un componente usato nell'ambito di oggetti serventi.

La specifica *Enterprise Java Beans* descrive un ambiente applicativo integrato che semplifica il processo di sviluppo di sistemi a oggetti distribuiti. Un *Enterprise Java Beans server* fornisce un ambiente per l'esecuzione di applicazioni sviluppate utilizzando la specifica *Enterprise Java Beans*, gestendo l'allocazione delle risorse necessarie. In particolare, un *Enterprise Java Beans server* fornisce un insieme di servizi infrastrutturali tipici riguardanti:

- *Il ciclo di vita degli oggetti*: sono gestite la creazione del processo nell'ambito del quale ogni oggetto è eseguito, nonché l'attivazione e la distruzione degli oggetti;
- *La gestione dello stato e la gestione di transazioni*: è assicurata la persistenza dello stato degli oggetti e, nel caso di attività transazionali, la persistenza del risultato finale raggiunto da una transazione andata a buon fine ovvero, nel caso di transazione fallita, il ripristino dello stato originario;
- *La sicurezza*: è gestito il processo di riconoscimento e di autenticazione degli utenti e la verifica dei livelli di autorizzazione necessari per l'attivazione di specifici servizi;
- *La distribuzione degli oggetti*: l'insieme dei servizi forniti da *Remote Method Invocation - RMI* rende l'allocazione degli oggetti serventi trasparente agli oggetti clienti. *RMI* supporta più protocolli di comunicazione fra cui il protocollo *IIOP*, con il quale si permette la cooperazione fra ambienti *CORBA* e ambienti *EJB*.

3.6. Tecnologie *Web* e *XML*

3.6.1. Introduzione

La tecnologia ipertestuale del *Web* è nata nel 1989 presso i laboratori del *CERN*, costituendo per alcuni anni la modalità innovativa di reperimento di informazioni su *Internet*. La facilità d'uso e la gradevolezza di presentazione hanno costituito motivi determinanti del successo di Internet presso il grande pubblico e della nascita della Rete *World Wide Web - WWW*. *WWW* è una rete mondiale di informazioni organizzate in *pagine ipertestuali*. Ogni pagina contiene informazioni di natura testuale o multimediale e *riferimenti*, espressi mediante indirizzi univoci denominati *Universal Resource Locator - URL*, a altre pagine contenenti informazioni correlate. L'insieme delle pagine *Web* può essere quindi visto come un vastissimo insieme di archivi distribuiti e fra loro collegati, nell'ambito del quale l'utente può *navigare* usando in modo diretto gli indirizzi *URL* delle pagine di suo interesse, se sono a sua conoscenza, o i riferimenti che incontra nelle pagine che attraversa.

3.6.2. Principi architetturali generali

Gli elementi architetturali principali alla base della tecnologia *Web* sono:

- I *Web server*, che costituiscono i *siti* ai quali gli utenti si collegano, tramite rete *Internet*, *Intranet* o *Extranet*, per usufruire dei servizi;
- I *Web browser*, sulle macchine utente, che gestiscono sia la presentazione delle informazioni in formato standard *HyperText Markup Language – HTML*, sia l'interazione con l'utente, comunicando con i *web server* mediante il protocollo standard *HyperText Markup Protocol - HTTP*.

Le prime tecnologie *web* consentivano la sola lettura di documentazione multimediale da parte dell'utente; negli ultimi anni la tecnologia è evoluta per permettere l'erogazione non solo di servizi informativi di tipo *statico* (lettura di documenti pubblicati), ma anche di servizi informativi *dinamici* (presentazione di risultati estratti da basi di dati) e di servizi transazionali veri e propri (si pensi, ad esempio, al settore in forte espansione del commercio elettronico). L'evoluzione tecnologica ha visto prima la nascita, nel 1995, del protocollo *Common Gateway Interface - CGI* che permette al *Web server* di attivare servizi forniti da applicazioni a lui esterne, restituendo il risultato delle elaborazioni al *web browser* che le ha richieste. In tempi più recenti la tecnologia *Web* è stata arricchita con un approccio a oggetti, introdotto grazie alla possibilità di includere nelle pagine di presentazione delle informazioni *riferimenti* a oggetti (*applet Java* o *ActiveX Microsoft*) capaci di eseguire operazioni di accesso e di manipolazione di informazioni e capaci di migrare dai *web server* ai *web browser* sfruttando il collegamento di rete (Figura 22).

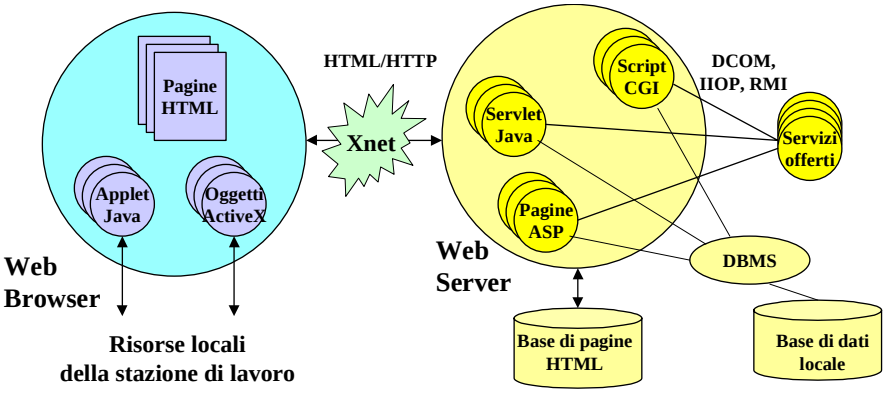

Figura 22 - La tecnologia Web

3.6.3. SGML, HTML, XML

Le pagine ipertestuali lette dai *web browser* sono descritte usando il linguaggio *HTML*, che usa un insieme di *marche standard* (dette *tag*) per identificare porzioni caratteristiche del testo stesso, quali il titolo, una frase da sottolineare, il riferimento a un'altra pagina. Ogni *tag* si riconosce dal testo generico in quanto costituita da un testo standard compreso fra le parentesi angolari "<" e ">". Lo standard *HTML* stabilisce un *insieme non estensibile* di *tag* che permettono di specificare caratteristiche di struttura, di formato e di presentazione delle pagine che il *web browser* mostra all'utente. Per tale motivo, *HTML* è un linguaggio specificatamente concepito per la presentazione di pagine ipertestuali. Un esempio di pagina HTML relativa a una tabella riguardante la vendita di prodotti nei tre quadrimestri del 2000 è mostrato nella figura seguente.

```
<TABLE border=”2” frame=”hsides”>
       <CAPTION>VENDITE QUADRIMESTRALI NEL 2000</CAPTION>
       <COLGROUP align=”left”>
       <COLGROUP align=”left”>
       <COLGROUP align=”right”>
       <COLGROUP align=”right”>
       <COLGROUP align=”right”>
       <THEAD valign=”top”>
              <TR><TH>Codice<TH>Descrizione<TH>Q1<TH>Q2<TH>Q3
       <TBODY>
              <TR><TD>20101<TD>Cancelleria minuta<TD>34,4<TD>36,7<TD>29,7
              <TR><TD>20102<TD>Fascicoli e buste<TD>76,3<TD>89,6<TD>92,8
              <TR><TD>20103<TD>Strumenti tecnici<TD>156,8<TD>192,7<TD>206,4
              <TR><TD>20104<TD>Materiale per imballaggi<TD>21,1<TD>22,5<TD>27,8
              <TR><TD>20105<TD>Macchine per ufficio<TD>567,8<TD>671,4<TD>765,9
</TABLE>
```

Figura 23 – Esempio di pagina HTML

Standard Generalized Markup Language – SGML è un meta-linguaggio, originariamente definito da *IBM* negli anni ’70 e diventato standard *ISO* nel 1986, con il quale è possibile specificare le caratteristiche di linguaggi di marcatura di documenti, quali *HTML*. *SGML* è estendibile, in quanto permette la definizione di insiemi personalizzati di *tag* che possono essere scelti dal progettista di applicazioni per descrivere le caratteristiche e l’organizzazione dei contenuti dei tipi di documento di interesse. Tali caratteristiche sono descritte, per ogni tipo di documento, mediante un *Document Type Definition – DTD*, la cui sintassi deve rispettare lo standard previsto dal linguaggio *SGML* stesso. Con *SGML*, quindi, le tipologie di documenti riconosciute nel contesto applicativo di interesse sono descritte mediante corrispondenti *DTD*. Il rispetto delle regole strutturali definite dai *DTD* è verificabile mediante programmi, scritti generalmente in linguaggi quali *C*, *C++* e *Java*, chiamati *parser*. L’estrema generalità e la difficoltà, da parte dei *web browser* attuali, di supportarne tutte le caratteristiche hanno impedito fino a oggi l’uso diffuso di *SGML* nel contesto industriale. Per tale motivo il *World Wide Web Consortium – W3C*, consorzio che promuove la standardizzazione della tecnologia web, ha definito nel 1998 *eXtensible Markup Language – XML*. *HTML* e *XML* sono linguaggi di marcatura derivanti entrambi da *SGML* ma in modo diverso; infatti, mentre *HTML* è una *istanza* specifica di *SGML*, in quanto prevede un insieme ben definito di *tag* concepiti specificatamente per la visualizzazione di pagine ipertestuali, *XML* è un *sottoinsieme* di *SGML*, rimanendo comunque come *SGML* un *meta-linguaggio* di specifica di linguaggi di marcatura.
XML permette di caratterizzare la struttura e il contenuto dei documenti, definendo corrispondenti meccanismi di validazione. Non sono oggetto di specifica con *XML* gli aspetti relativi alla caratterizzazione di legami ipertestuali, né quelli di presentazione delle pagine all’utente. Un esempio di documento XML riguardante un frammento di informazione relativo a contratti di locazione è mostrato in Figura 24.

```
<Locatore SessoLocatore="F">
    <NomeLocatore>Patrizia</NomeLocatore>
       <CognomeLocatore>Gentili</CognomeLocatore>
       nata il
       <DataNascitaLocatore>28041961</DataNascitaLocatore>
       a
       <ComuneNascitaLocatore>Roma</ComuneNascitaLocatore>
       provincia di
       <ProvinciaNascitaLocatore> ROMA </ProvinciaNascitaLocatore>
       codice fiscale
       <CodiceFiscaleLocatore>GNTGDU62D10D773D </CodiceFiscaleLocatore>
       e residente in via
       <IndirizzoLocatore> Augusto Vera</IndirizzoLocatore>
       numero
       <CivicoLocatore> 45 </CivicoLocatore>
       <ComuneLocatore> Roma </ComuneLocatore>
       provincia di
       <ProvinciaLocatore>Roma</ProvinciaLocatore>
</Locatore>
```

Figura 24: Esempio di documento XML

Al fine di permettere la caratterizzazione dei legami ipertestuali all'interno dei documenti e la caratterizzazione dei criteri di presentazione dei documenti all'utente, sono disponibili i *linguaggi di supporto* a XML *eXtensible Linking Language – XLL* e *eXtensible Stylesheet Language – XSL*.
Ad oggi, la famiglia degli standard relativi a *XML* è molto ampia. La Tabella 25 fornisce un elenco dei principali linguaggi disponibili o in corso di elaborazione.

XSL – Extensible Stylesheet Language	Linguaggio di specifica per la presentazione di documenti *XML*
SMIL – Standard Multimedia Integration Language	Linguaggio *XML* per la specifica di documenti multimediali
XML Signature	Linguaggio *XML* per la gestione di meccanismi di firma elettronica
XQuery	Linguaggio per effettuare ricerche evolute in raccolte di documenti *XML*
WIDL – Web Interface Definition Language	Consente di definire interfaccie funzionali all'interno dei documenti, richiamabili da oggetti remoti

Figura 25 – Linguaggi di marcatura definiti in ambito XML

Un documento organizzato usando il linguaggio *XML* è messo a disposizione delle applicazioni attraverso un processo di analisi del suo contenuto effettuato da un particolare programma chiamato *parser*. Il compito del *parser* è eseguire il controllo formale del documento, gestire gli eventuali errori riscontrati ed estrarre i dati di interesse in modo che l'applicazione che lo usa sia in grado di elaborarli in base agli scopi per i quali è stata progettata (visualizzazione, innesco di procedure transazionali, archiviazione, trasformazionee simili). Esistono in generale due tipi di *parser*:

- *parser non validanti*, che si limitano a controllare che il documento risponda alle specifiche generali del linguaggio *XML*;
- *parser validanti*, che controllano che il documento sia valido, ossia ben formato e conforme al *DTD* relativo al suo tipo; questo tipo di controllo è molto importante nel caso di scambio di documenti tra applicazioni diverse che usino *DTD* concordati.

La caratteristica fondamentale dei *parser XML* è che, grazie alla struttura sintattica comune a tutti i linguaggi derivati da *XML*, non è necessario scrivere un *parser* per ogni linguaggio: un solo *parser* scritto per *XML* può essere usato per qualunque linguaggio derivante da questo; il progettista dovrà solo scegliere tra i *parser* disponibili, *validanti* e *non validanti*, disponibili come strumenti gratuiti scaricabili da *Internet* o come prodotti commerciali di mercato.

Un contesto applicativo particolarmente interessante per *XML* e per i linguaggi *standard* ad esso correlati è quello dello scambio di informazioni tra sistemi informativi eterogenei appartenenti a organizzazioni diverse. *XML* è un linguaggio per rappresentare i dati in modo indipendente dall'architettura *hardware* e *software* del sistema e dai linguaggi di programmazione usati. È quindi possibile definire, nell'ambito di un sistema cooperativo, l'*universo del discorso* mediante la definizione di *DTD* concordati dalle diverse organizzazioni. Lo scambio informativo fra i diversi contesti è a questo punto ottenibile mediante documenti *XML* rispondenti ai *DTD* concordati. I canali di comunicazione a tale fine usati possono essere i più disparati e, fra questi, sono da ricordare in particolare quelli basati sui protocolli *HTTP* e *FTP*. Ogni organizzazione cooperante che riceva un documento *XML* ha la possibilità di validarne il contenuto usando un *parser* che analizzi il documento alla luce del *DTD* corrispondente. La presentazione finale di ogni documento, infine, potrà avvenire in forme diverse nelle diverse organizzazioni, sulla base di specifiche di formattazione *XSL* rispondenti alle necessità applicative locali.

3.7. Le tecnologie di incapsulamento di sistemi legacy

Il *Distributed Object Computing* promette di diventare l'approccio principale di realizzazione di sistemi informatici in realtà complesse, ma non sarebbe completo se non fornisse meccanismi con i quali valorizzare il patrimonio di informazioni e di transazioni accumulato nel corso degli ultimi decenni dalle organizzazioni di tutto il mondo. A tale esigenza di recupero e di valorizzazione di investimenti pregressi rispondono le *tecnologie di incapsulamento* che permettono l'*accesso trasparente*, da parte di un sistema a oggetti, a dati e transazioni ospitati in sistemi *legacy*. Le tecnologie di incapsulamento offrono interfacce di servizi invocabili da oggetti o usabili direttamente dagli utenti che ricadono in una delle seguenti tre categorie (Figura 26):

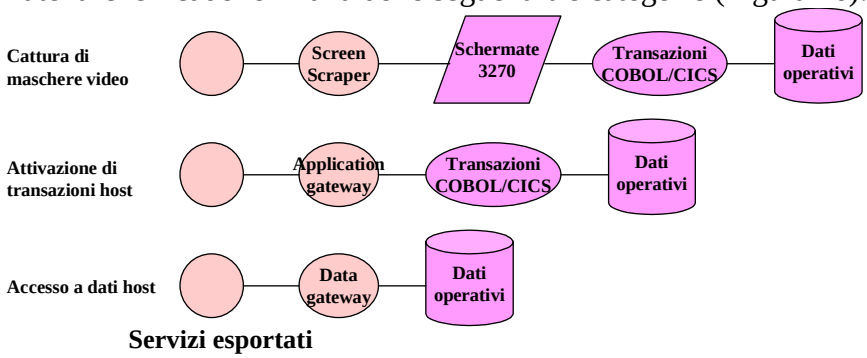

Figura 26 - Tecnologie di incapsulamento di sistemi legacy

- Incapsulamento delle interfacce utente *legacy* (*screen scraping*): questa classe di tecnologie consente, mediante l'uso di un processo chiamato *screen scraper* o *terminalista virtuale*, di mascherare il dialogo che avviene fra l'utente e il sistema *legacy* dotato delle tradizionali maschere video a caratteri. Il processo di mascheramento fornisce una interfaccia di servizi corrispondenti aventi ognuno un nome, un insieme di parametri di ingresso e un insieme di parametri in uscita. L'interfaccia di servizi è usata direttamente da altri oggetti di una architettura a oggetti distribuiti o per produrre una interfaccia utente innovativa di tipo *web*;
- Incapsulamento della logica applicativa di transazioni *legacy* (*application gateway*): questa classe di tecnologie interagisce con il sistema non a livello di interfaccia utente ma a livello di transazioni interne, offrendo al resto del sistema una interfaccia di servizi corrispondenti alle transazioni incapsulate. Questa tecnica è più *invasiva* della tecnica precedente, in quanto richiede in generale l'installazione di *software* specifico all'interno del sistema *legacy* e l'interazione con il suo *Transaction Monitor*;
- incapsulamento dell'accesso ad archivi *legacy* (*data gateway*): questa tecnica permette l'attivazione diretta di interrogazioni o di operazioni di aggiornamento sui dati del sistema *legacy*. Questa tecnica è ancora più invasiva e complessa delle tecniche precedenti, sia perché

induce sul sistema *legacy* carichi elaborativi di tipo completamente nuovo, sia perché richiede una profonda conoscenza sul significato dei dati contenuti negli archivi che spesso è difficile da ottenere.

4. Architetture dei sistemi distribuiti e scenari applicativi

4.1. Architetture

L'architettura di un sistema distribuito può essere, in termini generali, molto complessa. È possibile però, nella maggioranza dei casi, riconoscere tre livelli logici, in letteratura spesso indicati come *application layer*, che trovano corrispondenza in altrettanti strati *software* nei quali vengono partizionate le applicazioni:

1. Il *layer* che si occupa dell'interazione con l'utente (*logica di presentazione*);
2. Il *layer* che si occupa delle funzioni da mettere a disposizione dell'utente (*logica di business o logica applicativa*);
3. Il *layer* che si occupa dei dati (*logica d'accesso ai dati*).

Nei sistemi distribuiti i *layer* sono installati su un numero, anche differente, di livelli *hardware* (detti *tier*), dove un livello rappresenta in generale una macchina di diversa potenza elaborativa. Da questo secondo punto di vista, una applicazione può essere configurata con:

- *Un livello di distribuzione (Single Tiered)*: tutti e tre i livelli applicativi vengono assegnati a un'unica macchina: questa configurazione è quella classica *terminale-host*.
- *Due livelli di distribuzione (Two Tiered)*: i livelli applicativi sono divisi tra la stazione di lavoro dell'utente (tipicamente un *personal computer*) e la macchina *server* che ospita i dati (di tipo *mainframe* o *mini computer*); sulla stazione di lavoro è realizzata la logica di presentazione e sulla macchina *server* quella d'accesso ai dati; differenti scelte sono possibili per l'allocazione della logica applicativa.
- *Tre livelli di distribuzione (Three Tiered)*: i tre livelli applicativi sono suddivisi tra altrettante macchine: una stazione di lavoro di tipo *personal computer*, un *server* intermedio di tipo *mini computer* e un *server* di gestione dei dati di tipo *mini computer* o di tipo *mainframe*.

La prima configurazione, pur rappresentando il vecchio modello centralizzato, mantiene ancora una sua validità. Infatti i sistemi *mainframe, Single Tiered* per eccellenza, garantiscono un livello di prestazioni, affidabilità e sicurezza che le moderne soluzioni distribuite non hanno ancora pienamente eguagliato.

Relativamente alla seconda configurazione, in base a come sono allocati i tre *layer* applicativi, sono possibili cinque differenti architetture (Figura 27), aventi caratteristiche differenti in termini di traffico di rete, di prestazioni con alti carichi transazionali, di compiti eseguiti dalla stazione di lavoro (secondo configurazioni *fat client* o *thin client*) che comportano livelli diversi di flessibilità e di facilità di modifica. Tra queste quella attualmente più comune, perchè indotta dai più diffusi ambienti di sviluppo rapido (*Rapid Application Development – RAD*) e dai sistemi di gestione di basi di dati commerciali, è la *Gestione Dati Remota* che può avere impredicibili effetti sul traffico di rete e porta allo sviluppo di oggetti clienti ricchi di funzionalità (*fat client*) e quindi difficili da manutenere.

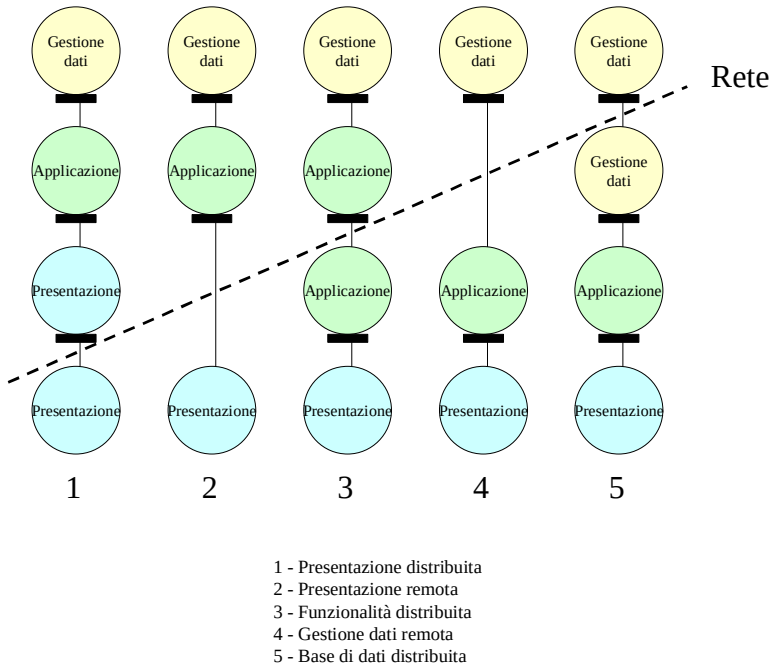

Figura 27 - Classificazione delle architetture 2-tier (secondo Gartner Group).

La tabella di Figura 28 mostra la possibile ripartizione dei tre *layer* architetturali su tre *tier* hardware in una configurazione *three tiered*: P, A e D indicano rispettivamente i *layer* di Presentazione, di Gestione della logica applicativa, di Gestione dati.

Tier/Configurazione	1	2	3	4	5	6	7	8	9	10	11	12	13
Tier 3	D	D	D	D	D,A	D	D	D,A	D	D	D,A	D,A	D, A,P
Tier 2	D	D	D,A	A	A	D,A	A	A	D, A,P	A,P	A,P	P	P
Tier 1	D, A,P	A,P	A,P	A,P	A,P	P	P	P	P	P	P	P	P

Figura 28: Classificazione delle architetture 3-tiered

Con riferimento alla Figura:

- Le configurazioni da 1 a 5 riguardano situazioni *fat client*, maggiormente indicate per realtà interne dove l'elevato supporto richiesto dall'utenza controbilancia i maggiori oneri di supporto all'esercizio;
- Le configuazioni da 9 a 13 riguardano situazioni *thin client*, adatte alla distribuzione di servizi a un insieme aperto di utenti;
- Le configurazioni 4, 5, 7, 8 e da 10 a 13 realizzano la separazione fisica fra utenti (*tier* 1) e dati (ospitati unicamente nel *tier* 3), per il tramite di una macchina intermedia (*tier* 2). Tali configurazioni sono qundi particolarmente sicure riguardo all'accesso ai dati da parte degli utenti;
- La tecnologia *Web*, inizialmente nata per fornire informazioni su *Web browser* (*tier* 1) mediante un *Web server* (*tier* 2) (configurazioni da 6 a 13), si è negli ultimi tempi evoluta per permettere l'esecuzione di logica applicativa sul *Web browser* (configurazioni da 2 a 5) o anche per accedere alle risorse locali del *tier* 1 (configurazione 1);
- La tecnologia di incapsulamento di risorse *legacy* (*tier* 3) permette di mettere a disposizione degli altri livelli architetturali servizi informativi (configurazioni da 1 a 4, 6, 7, 9, 10) o servizi transazionali (configurazioni 5, 8, da 11 a 13) arrivando, grazie alla sua integrazione con un approccio *Web*, a offrire le classiche transazioni *mainframe* anche a una utenza che usi reti *Internet*, *Intranet* o *Extranet* per l'accesso (configurazione 13).

Con riferimento alle tre classi di tecnologie principali che compongono le moderne infrastrutture a oggetti distribuiti, ovvero i *middleware* generalizzati a oggetti, le tecnologie di diffusione di servizi

tramite *Web* e le tecnologie di incapsulamento di sistemi *legacy*, è possibile ipotizzare due naturali distribuzioni delle stesse sui tre *tier* (Figura 29).

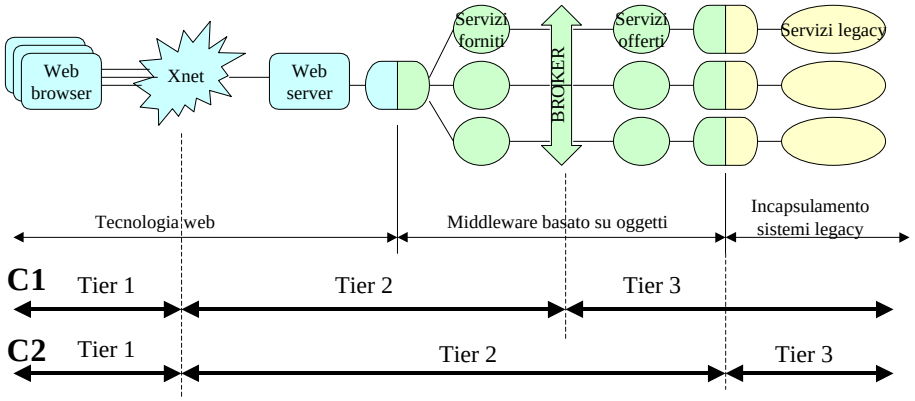

Figura 29: Distribuzione delle tre classi di tecnologie a oggetti sui tre tier

La configurazione contrassegnata con C1 è propria di situazioni nelle quali, sul *tier* 3 (tipicamente un *mainframe*) sono installati strumenti di incapsulamento di transazioni esistenti che forniscono una vista a oggetti delle stesse. L'approccio è invasivo rispetto al sistema *legacy* esistente ma permette una integrazione forte del mondo *mainframe* con il resto del sistema distribuito.
La configurazione contrassegnata con C2 è propria di situazioni nelle quali i servizi offerti dal *tier* 3 sono integrati nel sistema distribuito in modo non invasivo (per esempio con tecniche di cattura automatica delle informazioni presentate dalle interfacce utente native - *screen scraping*), applicabili quando la ripartizione del sistema in più *tier* corrisponda a una ripartizione organizzativa di ruoli fra diversi attori (federazione di sistemi).
E' infine da notare come la suddivisione di una architettura in livelli hardware è indipendente da quella in livelli software. Spesso infatti può capitare che applicazioni distribuite (e quindi in presenza di più livelli hardware) non presentino una chiara stratificazione in livelli software. Tale caratteristica si rivela spesso molto problematica nel momento in cui si debba modificare il sistema. Si può quindi affermare che la stratificazione *software* (*layer*) è concettualmente più importante di quella *hardware* (*tier*), anche perché senza la prima risulta assai difficile riuscire a pervenire alla seconda.

4.2. Scenari applicativi

Nel paragrafo precedente si è avuto modo di richiamare le tre grandi famiglie tecnologiche: le tecnologie basate sul *Web*, particolarmente adatte per la diffusione dei servizi; i *middleware* a oggetti distribuiti, per organizzare una molteplicità di oggetti, spesso forniti da organizzazioni diverse, in un contesto tecnologico unitario; le tecnologie di incapsulamento di sistemi *legacy*, per fornire informazioni e servizi *legacy* in modo innovativo.
Queste tre tipologie di tecnologie possono essere usate singolarmente o in modo concatenato, fornendo così scenari architetturali adatti per la risoluzione di diverse problematiche applicative. Si può affermare che l'uso combinato di due o più tipi di tecnologie è in generale attualmente fattibile, per via della disponibilità dei protocolli e dei formati *standard* di interscambio; inoltre ogni combinazione tecnica è in generale soluzione di problemi applicativi significativi, ovvero base per sviluppi organizzativi e applicativi innovativi. La disponibilità, inoltre, dei linguaggi di marcatura basati su *XML* e di strumenti software in grado di interpretarli è elemento abilitante per la realizzazione di sistemi cooperativi, in particolare in scenari di basso accoppiamento fra le organizzazioni che cooperano per la loro realizzazione.
La Figura 30 ripresenta per comodità la simbologia grafica adottata per ciascuna famiglia.

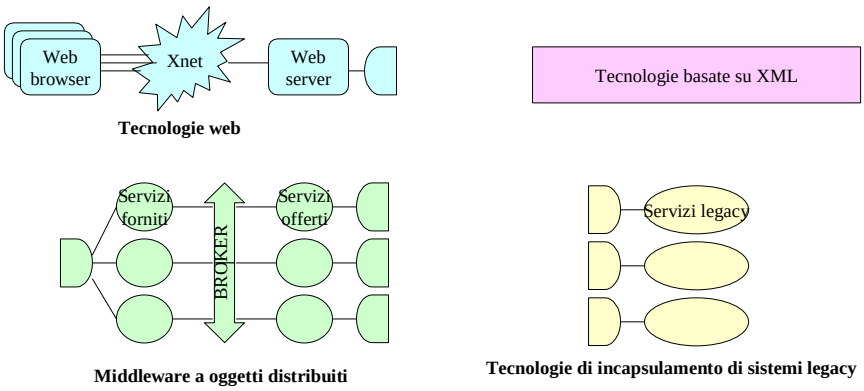

Figura 30: le tecnologie del Distributed Object Computing

Al fine di esplorare con sistematicità lo spazio delle diverse possibilità d'impiego applicativo delle classi tecnologiche individuate, nel seguito applicheremo una *strategia guidata dalle esigenze applicative* secondo la quale, data una classe di scenari applicativi, determineremo la combinazione architetturale più idonea a soddisfarli. La strategia è mostrata in Figura 31.

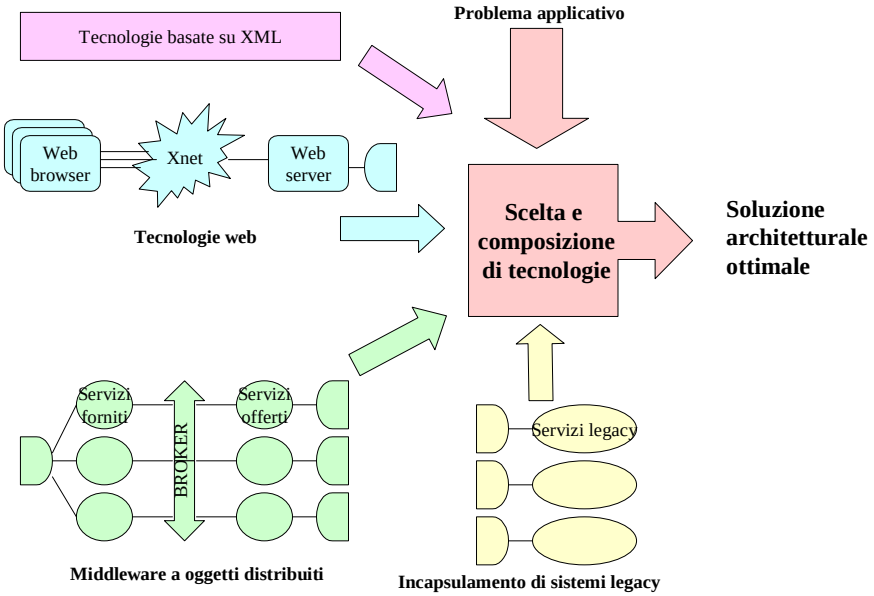

Figura 31: Strategia guidata dalle esigenze applicative

Per ogni problema applicativo significativo individuiamo le classi tecnologiche che, singolarmente o in composizione, offrono la soluzione architetturale ottimale.
Al fine di esplorare in modo sistematico almeno un sottoinsieme significativo di tutte le possibilità applicative che possono presentarsi nella realtà, analizziamo la casistica che si presenta al variare dei seguenti quattro parametri significativi:

- Fornitura del servizio da parte di un'unica organizzazione o da parte di più organizzazioni;
- Fornitura di servizi informativi o transazionali;
- Fornitura del servizio a utenti interni o esterni;
- Necessità di incapsulare sistemi *legacy* nell'ambito del servizio offerto.

La matrice delle possibilità è presentata in figura 32. La matrice presenta, per ogni incrocio, l'insieme delle tecnologie che appaiono essere più idonee per l'ottenimento del servizio. In particolare:

- Le tecnologie di incapsulamento (indicate con I) sono presenti laddove siano presenti sistemi *legacy* da valorizzare;

- Le tecnologie *Web* (indicate con W) sono presenti laddove siano da servire utenti esterni all'organizzazione (*Internet*, *Extranet*) e sono consigliabili per servire utenti interni (*Intranet*);
- Le tecnologie di *middleware* a oggetti distribuiti (indicate con M) sono presenti laddove esista una pluralità di fornitori di servizi da coordinare e possono inoltre apparire nell'ambito della fornitura di servizi transazionali innovativi, come gestori di transazioni;
- Le tecnologie *XML* (indicate con X) sono presenti laddove si debba coordinare il contributo informativo e transazionale di più organizzazioni cooperanti e, comunque, nell'ambito dell'erogazione di servizi su piattaforme *web*.

		Una unità organizzativa		Più unità organizzative	
		Presenza di legacy	Assenza di legacy	Presenza di legacy	Assenza di legacy
Servizio interno	S. informativo	I + W + X	W + X	I + M +W + X	M + W + X
	S. transazionale	I (+ M) (+ W + X)	(M) (+W + X)	I + M (+ W) + X	M (+ W) + X
Servizio esterno	S. informativo	I + W + X	W + X	I + M + W + X	M + W + X
	S. transazionale	I (+ M) + W + X	(M +) W + X	I + M + W + X	M + W + X

Figura 32: Uso delle tecnologie di Distributed Object Computing nei diversi ambiti applicativi

Si illustrano nel seguito alcune esigenze applicative particolarmente significative e frequenti.

4.2.1. Diffusione di servizi esistenti verso una pluralità di utenti

Nella realtà applicativa è frequente il caso in cui si desideri mettere a disposizione di una utenza diffusa servizi originariamente concepiti per una utenza ristretta e specializzata. Si pensi, per esempio, a tutte le nuove iniziative di *e-procurement* che mirano a snellire il processo di acquisto di prodotti e servizi da parte di grandi organizzazioni. In tale scenario applicativo, parte dei servizi informatici precedentemente usati dal personale interno dell'ufficio acquisti (quali, ad esempio, le procedure di acquisizione dei dati riguardanti un fornitore o un prodotto) possono essere usati direttamente dai fornitori stessi, attraverso un collegamento in rete *Extranet*. In questo caso l'approccio architetturale naturale è costituito dalla combinazione di tecnologie *web* con tecnologie di incapsulamento di sistemi *legacy* (Figura 33).

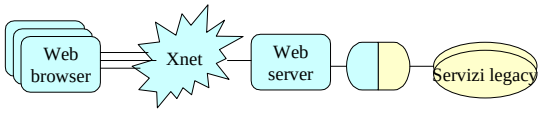

Figura 33: Diffusione di servizi esistenti a una pluralità di utenti

4.2.2. Erogazione di servizi da parte di una federazione di organizzazioni

La crescente specializzazione delle organizzazioni, la loro focalizzazione sulla missione e, in parallelo, il concepimento di prodotti e servizi a valore aggiunto e innovativi rendono sempre più attuali, sia nel contesto pubblico, sia nel contesto privato, scenari nei quali il bene di consumo è frutto della cooperazione di più organizzazioni fra loro autonome. In questo ambito è particolarmente rilevante la combinazione di *middleware* ad oggetti distribuiti, di tecnologie di incapsulamento di sistemi *legacy* e di tecniche di formalizzazione delle informazioni di interesse tramite *XML*. L'insieme delle tecnologie adottate può essere ulteriormente arricchito dalla famiglia delle tecnologie *web* per permettere l'accesso ai servizi da reti *Internet*, *Intranet* o *Extranet* (Figura 34).

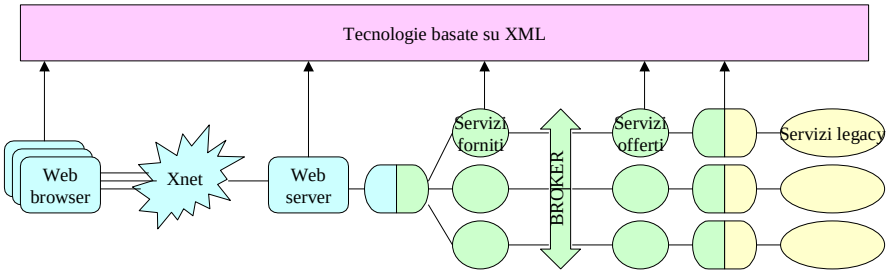

Figura 34: Fornitura di servizi da parte di una federazione di organizzazioni diverse

4.2.3. Adeguamento dell'interfaccia utente di applicazioni esistenti

L'adeguamento dell'interfaccia utente di un sistema tradizionale a paradigmi di interazione moderni è spesso il primo passo compiuto da una organizzazione per fornire alla propria utenza un servizio al passo con i tempi, senza che ciò comporti nell'immediato elevati investimenti di rifacimento e di migrazione verso nuove tecnologie. L'uso di tecnologie di incapsulamento di sistemi *legacy*, eventualmente combinato con le tecnologie *web*, permette di rendere disponibili i servizi offerti da un sistema *legacy* con paradigmi di interazione grafici più moderni rispetto alle classiche interfacce a carattere (Figura 35).

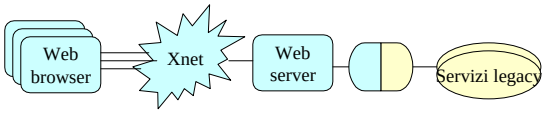

Figura 35: Adeguamento di sistemi legacy con interfacce utente moderne

4.2.4. Realizzazione di servizi a valore aggiunto

Sono numerosi gli scenari applicativi nei quali è di attualità la realizzazione di servizi a valore aggiunto a partire da servizi elementari esistenti. A titolo di esempio, si pensi ai servizi di investimento finanziario offerti da realtà bancarie e assicurative. In tale contesto si assiste alla tendenza di portare il servizio *presso* il cliente, semplificandone la logica di fruizione e potenziandone il livello di competitività rispetto a servizi simili offerti dalla concorrenza. L'esigenza fondamentale da soddisfare riguarda la semplificazione massima possibile del processo che il cliente deve compiere per scegliere fra alternative di investimento e per ordinare conseguentemente un acquisto o una vendita di titoli. Questo processo, che in uno scenario tradizionale richiede l'intervento umano dello specialista finanziario, nello scenario innovativo prevede la realizzazione di un ambiente virtuale per l'utente che metta a disposizione un insieme completo di servizi informativi sulle opportunità di investimento presenti e un processo integrato per ordinare compravendite di titoli. Le tecnologie abilitanti di questo scenario sono quindi i *middleware* ad oggetti distribuiti, le tecnologie di incapsulamento di sistemi legacy e le tecnologie *web* per l'accesso ai servizi da rete Internet (Figura 36).

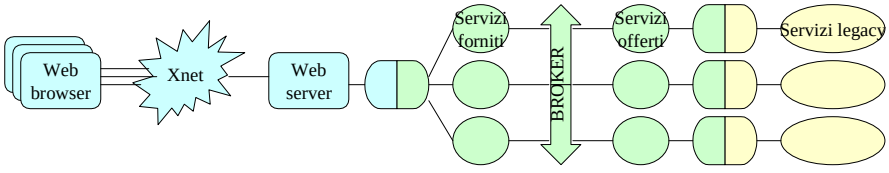

Figura 36: Realizzazione di nuovi servizi integrando servizi esistenti

4.2.5. Interoperabilità fra organizzazioni diverse

Sono numerosi gli scenari applicativi nei quali è necessario realizzare l'interoperabilità fra organizzazioni diverse. In ambito privato, è particolarmente importante l'insieme delle esigenze relative al supporto della catena del valore (*value chain management*) attraverso la quale le capacità di organizzazioni diverse sono messe a fattore comune al fine della vendita sul mercato di beni e servizi di consumo. In ambito pubblico, l'interoperabilità fra organizzazioni diverse è alla base dell'automazione dei processi di servizio inter-amministrativi. Per tali scenari applicativi, sono da considerare tecnologie abilitanti i *middleware* ad oggetti distribuiti basati su *broker* di oggetti o su broker di messaggi, i protocolli di comunicazione fra *broker* tipo *IIOP* e le tecnologie *XML* per la determinazione di formati comuni dei dati da scambiare (Figura 37).

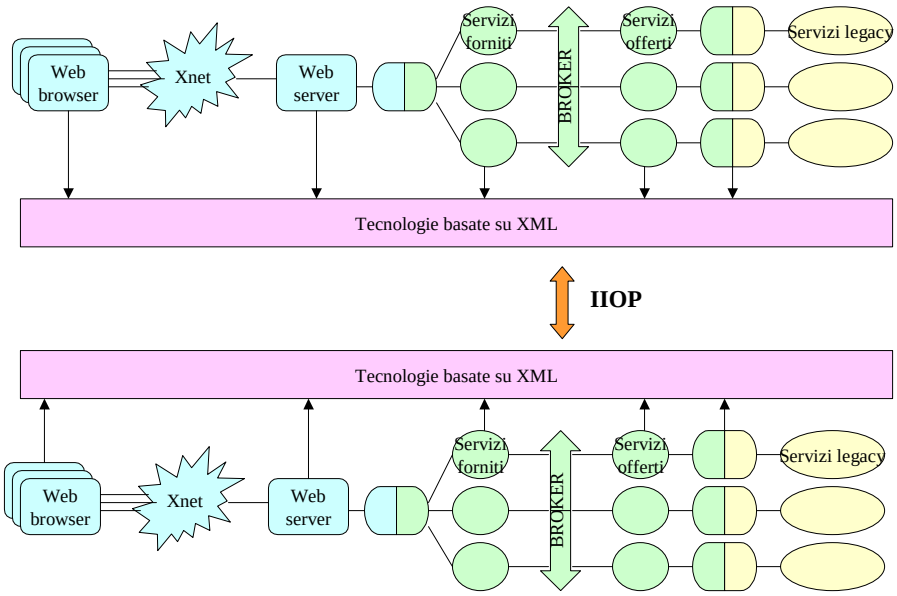

Figura 37: Interoperabilità fra organizzazioni mediante comunicazione fra broker

5. Conclusioni

Lo sviluppo delle metodologie e delle tecnologie di *Distributed Object Computing* cui si sta assistendo nel corso di questi ultimi anni e che è stato illustrato in questo contributo promette di fornire soluzioni tecniche valide per diversi scenari applicativi, alcuni dei quali particolarmente importanti per il miglioramento dei servizi offerti e per la razionalizzazione dell'organizzazione interna di realtà pubbliche e private. Le soluzioni tecnologiche disponibili hanno o stanno per raggiungere, in generale, un livello di maturità accettabile per consentirne l'impiego in contesti pratici di rilievo. Nel prossimo futuro si attende:

- un'ulteriore maturazione sui fronti della scalabilità, della robustezza e dell'ingegnerizzazione delle interfacce d'uso;
- un aumento dei livelli di integrazione e di supporto di piattaforme tecnologiche diverse, grazie al consolidarsi di *standard*;
- lo sviluppo di ambienti di sviluppo e di supporto all'esercizio dei sistemi che facilitino ulteriormente il compito degli specialisti;
- la maturazione sul fronte di tematiche specifiche, quali la sicurezza, che attualmente richiedono studi e verifiche di volta in volta.

Dal punto di vista metodologico, l'approccio a oggetti, originariamente concepito per essere applicato *in piccolo*, a livello di singola applicazione, è oggi chiamato a governare la complessità dei grandi sistemi informativi distribuiti e interoperanti. Le problematiche che si incontrano riguardano:

- la non completa maturità del metodo in rapporto alla complessità e alla varietà delle tematiche architetturali e tecnologiche legate al *Distributed Object Computing*;
- la complessità delle tematiche specialistiche interessate, di natura progettuale, realizzativa e di gestione, che rende lungo e complesso il processo di apprendimento.

Da questo quadro emerge la necessità, per le organizzazioni, di pianificare attentamente la strategia di applicazione di questo nuovo paradigma architetturale; è necessario muoversi con piccoli passi incrementali, per ognuno dei quali siano presenti le fondamentali attività d'istruzione, apprendimento sul campo e riflessione sull'esperienza condotta. La transizione verso il *Distributed Object Computing* non è semplicemente un cambiamento di architetture o di tecnologie. Si deve avere la consapevolezza che sarà necessario un certo tempo per attuarla e che potrà essere convenientemente condotta eseguendo dapprima alcuni progetti pilota mirati, che forniscano l'esperienza necessaria per la progressiva diffusione a tutti gli ambiti interessati.

6. Riferimenti bibliografici

[AA.VV. 1997] - Autori vari - Object-Oriented Application Frameworks - Numero monografico su Communications of the ACM, ottobre 1997

[Batini, Mecella, 2000] – C. Batini, M. Mecella – Cooperation of Heterogeneous Legacy Information Systems: a Methodological Framework – Proceedings of the 4th International Enterprise Distributed Object Computing Conference, Makuhari, Giappone, 2000

[Birrell, Nelson, 1984] – A. D. Birrell, B. J. Nelson – Implementing Remote Procedure Calls – ACM Transactions on Computer Systems, febbraio 1984

[Booch, 1994] – G. Booch – Object-Oriented Analysis and Design with Applications – Benjamin/Cummings, 1994

[Booch et al., 1998] – G. Booch, J. Rumbaugh, I. Jacobson – The Unified Modeling Language User Guide – Addison Wesley, 1998

[Brodie, 1998] – M. L. Brodie – The Cooperative Computing Initiative: a Contribution to the Middleware and Software Technologies – The Cooperative Computing Initiative, gennaio 1998

[Coad, Yourdon1, 1991] - P. Coad, E. Yourdon – Object-Oriented Analysis – Yourdon Press, 1991

[Coad, Yourdon2, 1991] - P. Coad, E. Yourdon – Object-Oriented Design – Yourdon Press, 1991

[Coleman et al., 1993] – D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes – Object-Oriented Development: the Fusion Method – Prentice Hall, 1993

[Coulouris et al., 1994] - G. Coulouris, J. Dollimore, T. Kindberg – Distributed Systems, Concepts and Design – Addison Wesley, 1994

[D'Souza, Wills, 1998] - D.F. D'Souza, A.C. Wills - Objects, Components and Frameworks with UML: The Catalysis Approach - Addison Wesley, 1999

[Duchessi, Chengalur-Smith, 1998] - P. Duchessi, I. Chengalur-Smith - Client/Server Benefits, Problems, Best Practices - Communications of the ACM, maggio 1998

[Gamma et al., 1997] - E. Gamma, R. Helm, R. Johnson, J. Vlissides - Design Patterns: Elements of Reusable Object-Oriented Software - Addison Wesley, 1997

[Goscinski, 1991] – A. Goscinski – Distributed Operating Systems, the Logical Design – Addison Wesley, 1991

[Graham et al., 1997] - I. Graham, B. Henderson-Sellers, H. Younessi - The OPEN Process Specification - Addison Wesley, 1997

[Hutt1, 1994] – A. Hutt – Object Analysis and Design, Description of Methods – OMG, 1994

[Hutt2, 1994] – A. Hutt – Object Analysis and Design, Comparison of Methods – OMG, 1994

[Jacobson, 1995] – I. Jacobson – The Object Advantage: Business Process Reengineering with Object Technology – Addison Wesley, 1995

[Jacobson et al., 1999] – I. Jacobson, G. Booch, J. Rumbaugh - The Unified Software Development Process - Addison Wesley, 1999

[Krieger, Adler, 1998] - D. Krieger, R.M. Adler - The Emergence of Distributed Component Platforms - IEEE Computer, marzo 1998

[Laufmann et al., 1995] – S. Laufmann, S. Spaccapietra, T. Yokoi - Foreword – Proceedings of the Third International Conference on Cooperative Information Systems, Vienna, 1995

[Lewandowski, 1998] - S.M. Lewandowski - Frameworks for Component-Based Client/Server Computing - ACM Computing Surveys, marzo 1998

[Mylopoulos, Papazoglou, 1997] – J. Mylopoulos, M. Papazoglou – Cooperative Information Systems – IEEE Expert, 1997

[Mullender, 1993] – S. Mullender – Distributed Systems – Addison Wesley, 1993

[OMG, 1991] - OMG - The Common Object Request Broker: Architecture and Specification - OMG, 1991

[Rumbaugh et al., 1991] – Rumbaugh, Blaha, Premerlani, Eddy, Lorensen – Object Oriented Modeling and Design - Prentice Hall, 1991

[Shlaer, Mellor, 1988] – S. Shlaer, S. Mellor – Object-Oriented Systems Analysis: Modeling the World in Data – Prentice Hall, 1988

[Simon, 1996] – E. Simon – Distributed Information Systems, from Client/Server to Distributed Multimedia – McGraw Hill, 1996

[SUN, 1995] - Sun Microsystems - The Java Language Environment: a White Paper – 1995

[Tepfenhart, Cusick, 1997] - W.M. Tepfenhart, J.J. Cusick - A Unified Object Topology - IEEE Software, gennaio 1997

[Umar1, 1997] - A. Umar - Application (Re)engineering: Building Web-Based Applications and Dealing with Legacies - Prentice Hall, 1997

[Umar2, 1997] - A. Umar - Object-Oriented, Client-Server, Internet Environments - Prentice Hall, 1997

[Voth et al., 1998] - G. Voth e altri - Distributed Application Development for Three-Tier Architectures: Microsoft on Windows DNA - IEEE Internet Computing, marzo 1998