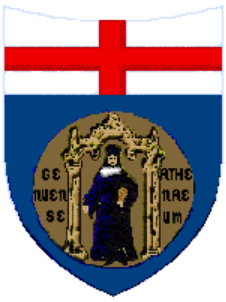
**Università degli studi di Genova**
**Facoltà di Scienze Matematiche Fisiche e Naturali**
**Corso di Laurea in Informatica**



Anno Accademico 2004/2005

Tesi di Laurea

# CooWS: Un'architettura per servizi web cooperativi

Candidato

**Luigi Bozzo**

Relatori
**Prof. Davide Ancona e**
**Prof.ssa Viviana Mascardi**

Correlatore
**Prof.ssa Elena Zucca**

*Alla mia famiglia*

# Ringraziamenti

```
10 PRINT "GRAZIE"
20 GOTO 10
```

*Luigi*

# Indice

# Introduzione

Riprendendo una definizione di M. Luck i sistemi ad agenti sono:

> *una delle più vibranti ed importanti aree di ricerca e sviluppo emerse nel campo dell'informatica durante gli anni 90, alla base delle più svariate applicazioni ed infrastrutture attualmente sul mercato.* (M. Luck, 2004)

Secondo la definizione di N. Jennings, K. Sycara, M. Wooldridge del 1998, un agente è un sistema in grado di agire *autonomamente* in un dato ambiente al fine di portare a termine i propri obiettivi. Un'altra nozione ben nota nel campo della ricerca è quella che pone l'agente come un'entità dotata di caratteristiche quali credenze, desideri, obiettivi e capacità decisionali (un "sistema intenzionale"). Negli ultimi quindici anni sono stati concepiti diversi modelli per lo sviluppo di sistemi intenzionali, uno tra questi è il modello BDI (acronimo di Belief, Desire, Intention - credenza, desiderio, intenzione) inizialmente proposto da A.S. Rao e M. Georgeff.

I linguaggi per lo sviluppo di sistemi BDI consentono la definizione di sistemi multiagenti (MAS, Multi Agent System) trascurando le caratteristiche specifiche dell'ambiente nel quale verranno collocati. Questo grazie alla mediazione tra agenti e ambiente esterno svolta da una coda eventi, in grado di accogliere i nuovi messaggi, e da un insieme di azioni per la modifica dell'ambiente stesso. L'insieme delle azioni esterne non è specificato a livello di linguaggio dato che esse variano da un dominio applicativo all'altro. Con queste caratteristiche i linguaggi BDI permettono di definire agenti che rispettino vincoli di *situatedness* ed allo stesso tempo mantengano la capacità di interazione con l'esterno. Inoltre questi linguaggi spesso supportano la socialità degli agenti fornendo loro la capacità di scambiarsi dati, in particolare i belief. In tal senso il modello Coo-BDI (Cooperative BDI) introduce

un'estensione della capacitá cooperativa degli agenti consentendo lo scambio di una conoscenza di tipo procedurale (i piani), realizzando di fatto una comunità di *learning agents*, agenti che imparano dalle proprie interazioni.

Il modello a multiagenti ben si presta quindi nella simulazione di sistemi economici, sociali, biologici ed in genere per la rappresentazione di tutti quegli ambienti reali dove si riscontra un elevato livello di complessità.
Allo stesso tempo, nonostante i diversi anni di studio ed i molti progressi compiuti in questo campo della ricerca, l'adozione di tecnologie ad agenti su larga scala è iniziata solo recentemente. La ragione di questo ritardo va ricercata nella mancanza di una struttura di supporto reale per la creazione di reti eterogenee di componenti e servizi, elemento fondamentale per lo sviluppo di sistemi ad agenti degni di nota.
La situazione è cambia solo recentemente con l'avvento di diversi middleware, piattaforme software concepite per l'integrazione di tecnologie infra-aziendali. Si spazia da protocolli wireless di basso livello come Bluetooth a tecnologie di più alto livello come Jini, CORBA, RMI ed i recenti Web Service. Questi protocolli supportano l'integrazione tra una vasta gamma di periferiche, a partire dai cellulari e PDA per finire alle workstation e server aziendali, fornendo così un ampio bacino di impiego. Tra questi la tecnologia dei Web Service si è recentemente distinta come valida alternativa nei confronti degli altri concorrenti grazie alla sua vocazione per l'implementazione di sistemi effettivamente "aperti". Inoltre l'architettura dei Web Service prevede dei ben definiti meccanismi per lo scambio dei messaggi, la descrizione delle interfacce, la pubblicazione e ricerca di servizi. Sono queste caratteristiche essenziali per il supporto di un'architettura basata sul paradigma ad agenti.

Il paradigma dei Web Service rappresenta oggi lo stato dell'arte dell'architettura orientata ai servizi (SOA - Service Oriented Architecture). Alla base di questa architettura, orientata in prima battura al B2B vi è il concetto di servizio, un componente con interfaccia accessibile via rete secondo i protocolli standard del Web (`http smtp ecc`. Questo componente è facilmente riutilizzabile ignorandone i dettagli di implementazione. Inoltre, essendo i protocolli Web indipendenti da piattaforme e linguaggi di programmazione, le applicazioni che ne risultano sono ben integrabili tra le differenti realtà aziendali fornendo allo stesso tempo una buona flessibilità per rispondere agilmente al cambiamento dei requisiti di design.

In questo lavoro di tesi viene proposto CooWS (Cooperative Web Service) , una implementazione del modello Coo-BDI sulla base degli standard dell'architettura dei Web Service. La piattaforma realizza un ambiente distribuito sul Web nel quale poter installare una comunità di agenti interoperanti secondo gli standard dell'architettura dei Web Service. L'ambiente è realizzato fisicamente da un'insieme di nodi CooWS pi‘ŭn nodo centrale con funzioni di registro per la catalogazione degli agenti. Ogni nodo della rete ospita un insieme di agenti ed ogni agente viene registrato sul catalogo insieme alle informazioni essenziali per la sua localizzazione da parte degli agenti partner. Ogni nodo espone una interfaccia secondo gli standard dettati dall'architettura dei Web Service e pertanto l'intero sistema CooWS risulta pubblicamente accessibile e di facile integrazione con i sistemi esterni. I piani di Coo-BDI vengono qui rappresentati come specifiche di processi BPEL installati ed eseguiti su ActiveBPEL, un motore BPEL esterno dedicato. Questa soluzione conferisce ancora una volta al sistema CooWS un elevato fattore di integrabilità con le risorse esterne pubblicate sul Web. Inoltre utilizzando i Web Service come azioni elementari viene di fatto superata una delle limitazioni del modello BDI che vuole la condivisione tra tutti gli agenti dell'insieme delle azioni per l'interazione con l'ambinete esterno.

Il contenuto di questo documento è così suddiviso:

Nel **Capitolo 1** viene presentato il modello Coo-BDI, un'estensione del modello BDI con la capacità di aggiornare la libreria attraverso un meccanismo cooperativo per lo scambio dei piani tra i diversi agenti. Dopo una breve introduzione al modello ad agenti BDI ed ad alcune sue problematiche irrisolte, vengono descritte le soluzioni proposte e sviluppate dal modello Coo-BDI. Si passa quindi alla vera e propria specifica strutturale dove si definiscono le strutture dati utilizzate tramite BNF. È quindi specificato in linguaggio Prolog un prototipo di interprete Coo-BDI. Infine si evidenziano i vantaggi portati da questa estensione attraverso l'analisi di un particolare esempio di applicazione.
Nel **Capitolo 2** vengono introdotti i concetti di Web Service, di modello orientato ai servizi e l'implementazione dell'architettura dei Web Service a partire dai differenti standard. Nella prima parte alcuni cenni storici illustrano l'evoluzione dal modello client-server a quello Web Service. Vengono quindi riprese alcune definizioni fondamentali dell'architettura Web Service

e degli standard sui quale essa poggia. Negli ultimi paragrafi vengono quindi passati in rassegna con maggiore dettaglio gli standard utilizzati all'interno di questo lavoro di tesi.
Nel **Capitolo 3** si affrontano le idee principali dell'architettura di CooWS, un'implementazione del modello Coo-BDI orientato ai Web Service. L'improntata data a questo capitolo vuole essere di descrizione a livello di progettazione per poi rimandare a maggiori dettagli di implementazione nel capitolo successivo. Nella prima parte vengono presentate le motivazioni, le idee e gli obiettivi alla base di questo lavoro. Segue un breve sunto della storia del processo di sviluppo. La sezione centrale del capitolo è occupata dalla descrizione delle problematiche affrontate e delle scelte impiegate per risolverle (tra queste in particolare l'utilizzo del linguaggio BPEL). Nella seconda parte viene illustrata l'architettura della piattaforma CooWS, partendo da uno sguardo d'insieme per poi scendere nei dettagli dei singoli moduli. Infine viene descritto il ciclo di esecuzione del motore ed il comportamento previsto nei differenti scenari.
Nel **Capitolo 4** viene presentata l'implementazione della piattaforma CooWS. Vengono subito presentati gli strumenti scelti per lo sviluppo e come questi siano impiegati in uno struttura che va a formare uno stack in cima al quale troviamo la piattaforma CooWS. Quindi vengono ripresi i principali moduli già definiti nel capitolo precedente e qui decorati con ulteriori informazioni di natura tecnica. Quindi i differenti componenti, disposti su un'architettura su tre livelli (presentazione, motore, archivio), vengono specificati con l'ausilio di stralci di codice.
In appendice è infine disponibile il codice sorgente relativo ai principali componenti citati all'interno del documento.

# Capitolo 1

# Coo-BDI: estensione del modello BDI con cooperatività

*Introdurremo qui il modello Cooperative BDI, un'estensione dell'architettura BDI con la capacità di modificare dinamicamente la libreria di piani tramite un meccanismo di scambio cooperativo di piani.*

## 1.1 Introduzione

Gli agenti intelligenti costituiscono una potente astrazione orientata alla modellazione di sistemi complesi in maniera semplice ed intuitiva. Per questo motivo negli ultimi anni sono stati proposti vari modelli ad agenti diluigitivi. Uno dei più famosi è quello Belief, Desire, Intention (BDI), la cui affermazione è testimoniata dal recente sviluppo della logica BDI, la definizione di linguaggi basati su tale logica (AgentTalk[1], 3APL[2], AgentSpeak(L)[3]) e la creazione di strumenti di sviluppo basati anch'essi sulle idee tipiche delle architetture BDI quali ad esempio JACK[4], PRS[5] e dMARS[6].
Nonostante questo largo consenso ricevuto, alcuni concetti e meccanismi BDI sono stati rappresentati ed implementati in modi non omogeneii e talvolta divergenti. Ad esempio, se consideriamo la nozione di "goal", esso è seplicemente costituito da un insieme di desideri. Molti autori ritengono che i goal debbano essere rappresentati esplicitamente in modo da poter ragionare sulle loro proprietà e garantirne la consistenza. In verità una rappresentazione esplicita è assente in linguaggi come dMARS, AgentSpeak(L) e JACK dove i goal, se rappresentati, lo sono solo transitoriamente come tipo di evento. Una situazione simile si ripete quando un evento (o un subgoal, dato che non viene fatta distinzione tra le due entità nei sistemi implementati) non può essere gestito a causa di mancanza di piani applicabili.
Quando si passa dalla specifica all'implementazione tale problema non può pi essere ignorato, ma i diversi sistemi lo affrontano in modo differente. Per i linguaggi esplicitamente basati sulle logiche BDI (quali dMARS, 3APL, JACK, AgentTalk e AgentSpeak(L)) ci si trova di fronte a due comportamenti: o si scarta l'evento privo di piani applicabili corrispondenti o si ricorre a un piano di default o a un comportamento alternativo che rimandi nel tempo la gestione dell'evento; per gli altri pare che l'approccio sia quello di fare fallire il piano per il quale non esistono istanze applicabili e proseguire con altre esecuzioni, "accettando/rassegnandosi" al fallimento. Questo evidenzia come il problema "nessun piano applicabile" sia risolto solitamente ad hoc. Un approccio al problema che tenta di risolvere la questione in un modo pi generale e che si fonda sulla capacità di cooperare degli agenti è quello introdotto da Ancona e Mascardi [7] che introduce il modello Cooperative BDI grazie ad un potente meccanismo di scambio piani finalizzato alla cooperatività tra agenti che consente l'aggiornamento dinamico della libreria di piani.
Una simile caratteristica risulta utile in svariati campi di applicazione quali:

*Personal Digital Assistants (PDA).* Le risorse hardware di un PDA solitamente sono limitate, motivo per cui l'integrazione con altre tecnologie ed il caricamento dinamico di codice sono caratteristiche fondamentali per tali dispositivi. Implementare un classico modello BDI per questo dominio applicativo non risulta di particolare efficacia. Il meccanismo di ricerca e caricamento di piani esterni, essenziale in questo contesto, non è infatti contemplato nel modello BDI originale. Il modello Coo-BDI invece, supportando nativamente la cooperazione tra agenti, meglio si adatta in questo contesto. La memorizzazione temporanea di un piano localmente ad un agente, proprietà supportata da Coo-BDI, viene inoltre incontro alle esigenze di spazio tipicamente limitate dei PDA.
*Agenti auto-riparanti.* Sono agenti in grado di aggiornare dinamicamente porzioni del loro codice in modo da mantenere un corretto funzionamento nel contesto di un ambiente in costante evoluzione. Quando un agente rileva una porzione di codice obsoleta la sostituisce con un aggiornamento fornito da un agente server, il tutto in maniera trasparente ovvero senza dover riavviare processi interni all'agente o relativi al sistema. Anche qui il modello Coo-BDI risulta essere vincente grazie alla possibiltà di aggiornare i piani locali con nuove versioni fornite dagli agenti partner.
*Maggiordomi digitali* Un maggiordomo digitale' è un agente che assiste l'utente in compiti specifici quali ad esempio la gestione dell'agenda degli appuntamenti, lo smistamento della posta ricevuta, la ricerca di informazioni sul Web. Caratteristica tipica di un maggiordomo digitale è la capacità di adattarsi dinamicamente alle esigenze dell'utente, sfruttando la cooperazione con altri maggiordomi digitali ed il controllo dell'utente stesso. Dal nostro punto di vista l'interazione dell'utente può essere vista come una sequenza di passi da applicare in determinate situazioni. Tale sequenza è vista dall'agente come un piano esterno, recuperato attraverso il meccanismo di cooperazione, da aggiungere alla propria libreria di piani. Anche in questo caso, quindi, l'utilizzo di Coo-BDI risulta più adatto di quello del modello BDI tradizionale, fornendo la flessibilità necessaria al maggiordomo digitale per apprendere le preferenze/abitudini dell'utente.
Più in generale Coo-BDI è adatto per modellare tutta quella serie di agenti che imparano dalle interazioni con altri agenti. Questo grazie alla principale introduzione del modello Coo-BDI, ovvero del meccanismo nativo per lo scambio di piani, utile quando un agente non ha la disponibilità di piani locali applicabili per soddisfare un particolare desiderio. Un simile approc-

cio rende necessaria una distinzione tra la nozione di *desiderio* e quella di *evento*. Ogni agente ha una propria coda contenente gli eventi ricevuti. Per ogni evento ricevuto viene generato un opportuno insieme di desideri. Questo meccanismo di generazione è strettamente locale all'agente (ovvero a questo livello non troviamo cooperazione) il quale decide quali desideri generare per un particolare evento senza alcuna interferenza da parte dell'esterno.
Se consideriamo per esempio l'evento "ricezione invito partecipazione al party a casa di Fabio alle 19", questo può generare i desideri "cancella tutti gli appuntamenti per questa sera" e "presentati a casa di Fabio alle 19" per un dato agente, mentre per un altro può generare il desiderio "telefona a Fabio per ringraziare e declinare l'invito". Questo esempio illustra la necessità di mantenere i desideri separati dagli eventi e di collezionarli in una esplicita struttura ad hoc.
In Coo-BDI la cooperatività tra gli agenti subentra successivamente alla generazione dei desideri nel momento in cui è necessario selezionare un piano appropriato per conseguire il desiderio corrente. Tornando all'esempio precedente, supponiamo che l'agente che desidera partecipare al party non sappia raggiungere la casa di Fabio, ovvero non abbia alcuna conoscenza procedurale (alcun piano) adatta allo scopo. In un sistema BDI classico l'agente fallirebbe o applicherebbe un piano di default. In Coo-BDI l'agente può invece rivolgersi ad altri agenti per ottenere le informazioni necessarie per raggiungere la casa di Fabio. Se uno degli agenti contattati sarà in grado di fornire tali informazioni le invierà all'agente richiedente permettendo quindi a quest'ultimo di soddisfare il desiderio di partecipare alla festa.

## 1.2 Panoramica su Coo-BDI

L'*estensione Cooperative BDI (Coo-BDI) per il modello BDI*, proposta da Ancona e Mascardi [8], promuove la *cooperazione*, cioé la capacità di un agente di aiutare altri agenti a soddisfare i loro desideri. In Coo-BDI la cooperazione consente agli agenti di scambiare i piani risolvendo così il problema "nessuna istanza di piano applicabile" anche se solo parzialmente.

Coo-BDI si basa sulla specifica di dMARS [6]. La prima estensione Coo-BDI prevede che eventi esterni e desideri principali siano tenuti separati. A tal fine ci sono due strutture: la "event queue" che contiene solo gli eventi esterni e il "desire set" che contiene solo desideri di tipo *achieve* generati dagli eventi. Si distingue tra *desideri principali (main desires)* che sono

tenuti nel desire set, e *desideri subalterni (subaltern desires)* che sono generati mentre si tenta di soddisfare un desiderio principale e restano impliciti nelle intenzioni dell'agente. Quando un desiderio principale non viene soddisfatto l'interprete Coo-BDI torna indietro (eseguendo il backtracking) e sceglie per esso un'istanza di piano non ancora provata. Se non ci sono piani non ancora provati per esso, il desiderio principale è rimosso dall'insieme. Il backtracking per i desideri subalterni non viene effettuato, sia per semplificare le strutture dati e l'interprete Coo-BDI, sia per seguire la stessa strategia adottata da dMARS.

La principale estensione di Coo-BDI, tuttavia, non riguarda l'introduzione dei desideri come oggetto distinto dagli eventi ma prevede

- l'introduzione della *cooperazione* tra agenti al fine di recuperare piani esterni per soddisfare desideri (principali e subalterni) e aggiornare dinamicamente la libreria dei piani;
- l'estensione di piani con *specificatori d'accesso*;
- l'introduzione dei piani di *default*;
- l'estensione delle *intenzioni* per tener conto del meccanismo di recupero delle istanze di piano esterne; e
- la modifica del Coo-BDI *engine* (interprete) per far fronte a tutti questi problemi.

*Strategia di cooperazione.* La strategia di cooperazione, o più semplicemente cooperazione, di un agente $A$ comprende l'insieme degli agenti con cui $A$ si aspetta di cooperare, la politica di recupero dei piani e la politica di acquisizione dei piani. La stategia di cooperazione può evolvere durante il tempo garantendo massima flessibilità e autonomia agli agenti.

*Piani.* I piani Coo-BDI sono classificati in *specifici* e *di default*. Come i piani BDI, entrambi i tipi di piano sono formati da un trigger, un precondizione, un corpo, un'invariante e due insiemi di azioni di successo e di fallimento. Oltre a queste componenti, tali piani hanno anche uno *specificatore d'accesso (access specifier)* che determina l'insieme degli agenti con cui il piano può essere condiviso. Esso può assumere tre valori: *private*, il piano non può essere condiviso, *public*, il piano può essere condiviso con qualunque agente e *only(TrustedAgents)*, il piano può essere condiviso solo con gli agenti contenuti nell'insieme *TrustedAgents*. Un *piano di default* è un piano in cui lo

specificatore d'accesso è *private*, il trigger è una variabile e la precondizione è la costante *true*; per definizione, un piano di default può essere sempre applicabile e non può mai essere scambiato. Ogni agente deve sempre fornire almeno un piano di default in modo che ogni desiderio possa essere gestito. Un *piano specifico* è un piano non di default.

*Intenzioni.* Le intenzioni Coo-BDI sono in relazione uno-a-uno con i desideri principali: ogni intenzione è creata quando un nuovo desiderio principale entra nell'insieme dei desideri, ed è cancellata quando il desiderio principale fallisce o è soddisfatto. Le intenzioni sono caratterizzate dalle componenti "standard" pi le componenti introdotte per gestire il meccanismo di recupero dei piani esterni. Se la politica di recupero piani lo prevede, i piani esterni sono recuperati, secondo la politica di recupero sia per i desideri principali che per quelli subalterni.

*Interprete Coo-BDI.* L'interprete Coo-BDI differisce dal classico interprete BDI per quanto concerne la generazione dei desideri ed il meccanismo di cooperazione.
Il suo funzionamento è riassumibile nei seguenti tre passi:

1. processare la event queue;
2. processare le intenzioni sospese;
3. processare le intenzioni attive.

Prima di scendere nei dettagli, descriviamo brevemente il meccanismo per il recupero dei piani rilevanti. Tale meccanismo si scompone a sua volta nei sequenti quattro passi .

1. L'intenzione corrente viene sospesa.
2. Si generano le istanze locali di piani rilevanti per il desiderio e le si associano all'intenzione.
3. Secondo la cooperazione, si definisce l'insieme $S$ degli agenti con cui si può interagire.
4. Si genera una richiesta di piano per il desiderio e la si spedisce a tutti gli agenti in $S$.

Per quanto concerne il primo passo dell'interprete, ovvero la gestione della coda degli eventi, esso viene affrontato in maniera diversa a seconda del tipo

di evento ricevuto. Distinguiamo due tipologie: gli eventi di *cooperazione*, e gli eventi *ordinari*. Gli eventi di cooperazione includono richieste di istanze di piani rilevanti per un desiderio e risposte a richieste di piano. Gli eventi ordinari comprendono almeno la ricezione di messaggi e la notifica di aggiornamenti fatti all'insieme dei belief dell'agente. Quando un agente riceve una richiesta da un altro agente $A$, esso manda ad $A$ l'insieme, anche vuoto, di tutte le istanze di piani locali rilevanti per il desiderio e visibili ad $A$ (i piani di default non sono visibili). Quando un agente riceve una risposta ad una richiesta di piano per un desiderio, controlla se la risposta è ancora valida e, in caso affermativo, aggiorna l'intenzione associata al desiderio in modo che includa le istanze di piano appena ottenute e memorizzi la risposta. Infine, se l'evento è ordinario, l'insieme dei desideri corrispondenti è generato ed aggiunto all'insieme dei desideri. Per ogni nuovo desiderio si genera una intenzione vuota e si innesca il meccanismo per il recupero dei piani rilevanti.

Il passo successivo consiste nel controllare se ci sono intenzioni sospese che possono essere ripristinate. Quando un'intenzione è ripristinata si genera l'insieme delle istanze di piano applicabili partendo dall'insieme delle istanze di piano rilevanti escludendo quelle già fallite, se ne sceglie una tra queste e si genera l'istanza di piano corrispondente da mandare in esecuzione. Se l'insieme delle istanze di piano applicabili è vuoto, il desiderio fallisce e viene cancellato dall'insieme dei desideri mentre l'intenzione è eliminata. In caso contrario, l'istanza di piano applicabile scelta è messa sullo stack dell'intenzione. Se il piano scelto è uno di quelli recuperati all'esterno, a seconda della politica di acquisizione può essere usato e scartato, oppure aggiunto alla libreria dei piani oppure ancora usato per sostituire i piani con un trigger che unifica con il trigger del piano recuperato.

Infine l'ultimo passo, la gestione delle intenzioni attive prevede che esse siano gestite come nell'architettura BDI eccetto per il meccanismo per il recupero di piani esterni rilevanti descritto in precedenza.

## 1.3 Specifica strutturale di Coo-BDI

La specifica strutturale di Coo-BDI è data mediante una BNF all'interno della quale si usano i non-terminali definiti informalmente come segue:

- `Variable`: un simbolo di variabile

- `Pred:` un simbolo di predicato

- `Term`: un termine
- `GroundTerm`: un termine privo di variabili (ground)
- `AgentId`: una stringa che rappresenta univocamente l'agente
- `RequestId`: una stringa che identifica una richiesta
- `IntentionId`: una stringa che identifica un'intenzione
- `Message`: ogni tipo di termine eventualmente contenente variabili libere
- `Substitution`: una corrispondenza tra variabili e termini
- Ogni non-terminale NT seguito dalla parola Set rappresenta un insieme di elementi in NT
- Ogni non-terminale NT seguito dalla parola Sequence rappresenta una sequenza di elementi in NT
- Ogni non-terminale NT seguito dalla parola Queue rappresenta una coda di elementi in NT
- Ogni non-terminale NT seguito dalla parola Stack rappresenta uno stack di elementi in NT.

Le sole differenze per gli ultimi quattro elementi consistono nelle primitive che saranno usate.

*Credenza, desiderio, interrogazioni e azione.* La formula che rappresenta una credenzaè un atomo o un atomo negato. Le credenze sono formule ground. I desideri sono della forma `achieve(BeliefFormula)`. Le interrogazioni sono denotate da `query(SituationFormula)` dove una situation formula è una belief formula, o una costante `true` o `false`, o una congiunzione o una disgiunzione di situation formula.

Le azioni possono essere interne o esterne. Le azioni interne sono aggiornamenti delle credenze dell'agente e possono essere eseguite solo se il loro argomento è ground. Le azioni esterne devono contemplare almeno le primitive per lo scambio di messaggi tra agenti.

```
BeliefFormula ::= Pred(Term, ...., Term) | not(Pred(Term, ...., Term))
Belief ::= Pred(GroundTerm, ...., GroundTerm) |
  not(Pred(GroundTerm, ...., GroundTerm))
```

```
SituationFormula ::= true | false | BeliefFormula |
  SituationFormula and SituationFormula |
  SituationFormula or SituationFormula
Desire ::= achieve(BeliefFormula)
Query  ::= query(SituationFormula)
InternalAction ::= add(BeliefFormula) | remove(BeliefFormula)
ExternalAction ::= send(AgentId, Message)
Action ::= InternalAction | ExternalAction
```

*Piani* Sono definiti da uno specificatore d'accesso (access specifier), un trigger, una precondizione, un corpo, un'invariante e due insiemi di azioni interne usati quando il piano fallisce o ha successo, rispettivamente. Sintatticamente non ci sono differenze tra piani specifici e piani di default. Gli specificatori d'accesso possono assumere i seguenti valori:
– `private` quando il piano non può essere fornito ad altri agenti;
– `public` quando il piano può essere fornito ad un agente qualsiasi;
– `only(TrustedAgents)` dove `TrustedAgents` è l'insieme degli identificatori degli agenti che indicano i soli agenti fidati che possono ricevere istanze del piano.
Il trigger di un piano specifico è il desiderio che il piano deve soddisfare. Precondizioni e invarianti sono situation formula. Il corpo di un piano è un albero non vuoto i cui nodi sono stati d'esecuzione e i cui archi sono etichettati da desideri, interrogazioni, o azioni interne/esterne (le azioni di successo e fallimento sono sequenze di azioni interne).

```
AccessSpecifier ::= private | public | only(AgentIdSet)
EdgeLabel ::= Desire | Query | Action
Body ::= state(EdgeBodySequence)
EdgeBody ::= (EdgeLabel, Body)
Plan ::= plan(AccessSpecifier, Desire, SituationFormula, Body,
SituationFormula, InternalActionSequence, InternalActionSequence)
```

*Istanze di piano* Un'istanza di piano è una coppia `(Plan, Substitution)` formata da un piano e una sostituzione. Un'istanza di piano è detta *rilevante* rispetto ad un desiderio se `Substitution` è l'unificatore pi generale tra il desiderio ed il trigger di `Plan`. Un'istanza di piano è detta *applicabile* se la formula ottenuta applicando `Substitution` alla precondizione di `Plan` è una conseguenza logica dei belief dell'agente. Si noti che le istanze di piano formate mediante un piano di default sono sempre rilevanti e applicabili.

L'esecuzione di un'istanza di piano è definita dall'istanza di piano con la sostituzione calcolata, con lo stato corrente del corpo del piano e l'insieme dei restanti fratelli dello stato corrente che non sono ancora stati eseguiti.

*Intezioni e richieste* Un'intenzione è composta da un identificatore univoco, uno stack per l'esecuzione delle istanze di piano, uno stato, un insieme di istanze di piano rilevanti, un insieme di identificatori di agente e un insieme di istanze di piano fallite. L'identificatore dell'intenzione è usato per modellare in modo conveniente le due relazioni `DesireIntention` e `IntentionRequest` che associano rispettivamente ogni desiderio principale all'intenzione corrispondente e ogni intenzione sospesa con la corrispondente richiesta di istanze di piano rilevanti.

Lo stack di esecuzione delle istanze di piano è simile allo stack di esecuzione dei programmi logici. Lo stato può essere `suspended` o `active`. Un'intenzione è sospesa se l'esecuzione dell'istanza di piano al top dello stack necessita la soddisfazione di un desiderio per cui non sono ancora state scelte istanze di piano; in questo caso l'insieme delle istanze rilevanti contiene le istanze di piano rilevanti che sono già state collezionate per il desiderio e l'insieme degli identificatori d'agente contiene tutti gli identificatori di quegli agenti che ci si aspetta che cooperino per soddisfare il desiderio.

Una richiesta di collaborazione è specificata da un identificatore unico, dall'identificatore dell'agente richiedente e dal desiderio da soddisfare.

Una relazione `relationIntentionRequest` associa ad ogni identificatore di intenzione sospesa la sua corrente richiesta di cooperazione (se c'è) ed ogni richiesta con l'intenzione sospesa che l'ha originata, se la richiesta è ancora valida

*Eventi.* Esistono due tipi di eventi: eventi di cooperazione e eventi ordinari. Un evento di cooperazione è del tipo `requested(request(RequestedId, ReqAgentId, Desire))` e rappresenta il fatto che l'agente identificato da `ReqAgentId` sta richiedendo un'istanza di piano rilevante per `Desire`, o `provide(AgentId, request(RequestedId, ReqAgentId, Desire), Instance)` che significa che l'agente identificato da `AgentId` ha cooperato rispondendo alla richiesta `request(RequestedId, ReqAgentId, Desire)` fornendo un insieme `Instances` di istanze di piano rilevanti per `Desire`. Gli eventi ordinari includono almeno quelli del tipo:
– `received(AgentId, Message)` attraverso il quale si rappresenta la ricezione del messaggio `Message` da parte dell'agente `AgentId`;
– `added(Belief)` quando un nuovo belief `Belief` è aggiunto alla base di conoscenza dell'agente;
– `removed(Belief)` quando il belief `Belief` è rimosso dalla base di conoscenza dell'agente.

Gli eventi percepiti dall'agente sono messi in una coda di priorità indicata come `eventQueue(EventQueue)`.

*Definizione dell'agente.* Un agente è definito mediante predicati asseriti nella sua stessa base di conoscenza:

- `agentId(AgentId)` che specifica l'identificatore unico dell'agente ;
- `eventQueue(EventQueue)` che indica la event queue corrente dell'agente;
- `isDesire(Desire)` che specifica l'insieme corrente dei desideri principali dell'agente;
- `isPlan(Plan)` che specifica l'insieme corrente dei piani (di default e specifici) dell'agente;
- `isIntention(Intention)` che specifica l'insieme corrente delle intenzioni dell'agente;
- i belief correnti dell'agente;
- tre predicati che specificano la *cooperazione* corrente per l'agente:
  - `trustedAgents(TrustedAgents)` che indica l'insieme degli identificatori di agenti attualmente fidati per l'agente;
  - `retrievalPolicy(Retrieval)` che specifica la politica di recupero attuale, dove `Retrieval ::= always | noLocal`;
  - `acquisitionPolicy(Acquisition)` che specifica la politica corrente di acquisizione dei piani dove `Acquisition ::= discard | add | replace`.
- `relationDesireIntention(Desire,IntentionId)` che specifica una relazione uno-a-uno tra il desiderio principale corrente per l'agente e gli identificatori di tutte le sue attuali intenzioni;
- `relationIntentionRequest(IntentionId,Request)` che specifica una relazione uno-a-uno tra gli identificatori di alcune delle intenzioni dell'agente attualmente sospese e alcune richieste effettuate dall'agente;
- `canResume(Request, AgentIdentifiers, PlanInstances)` che indica quando una certa intenzione sospesa, in attesa sulla richiesta di cooperazione `Request`, può essere ripristinata, fornendo l'insieme degli agenti che devono ancora rispondere alla richiesta in `AgentIdentifiers`, e l'insieme delle istanze di piano rilevanti raccolte finora in `PlanInstances`;
- `getDesires(OrdinaryEvent, Desires)` che specifica una funzione totale che associa ad ogni evento ordinario l'insieme (anche vuoto) di desideri principali che deve essere generato a partire da tale evento;

- `selectInstance(PlanInstances, SelectedPlanInstance)` che specifica una funzione totale che restituisce un elemento specifico `SelectedPlanInstance` di ogni insieme non vuoto di istanze di piano `PlanInstances`, tale che se `SelectedPlanInstance` è l'istanza di un piano di default, allora `PlanInstances` non contiene istanze di piani specifici;

- `selectIntention(Intentions, SelectedIntention)` che specifica una funzione totale che restituisce un elemento specifico `SelectedIntention` per ogni insieme non vuoto di intenzioni attive;

- `selectState(States, SelectedState)` che specifica una funzione totale che restituisce un elemento `SelectedState` per ogni insieme non vuoto di stati `States`.

Tra i predicati elencati sopra `agentId`, `canResume`, `getDesires`, `selectInstance`, `selectIntention` e `selectState` sono *statici* in quanto non possono essere modificati nell'arco della vita dell'agente, mentre tutti gli altri possono essere modificati dinamicamente.

## 1.4 Un prototipo per Coo-BDI

Il funzionamento dell'interprete di Coo-BDI è descritto usando Prolog [9] che è un linguaggio di programmazione logica. Prolog è stato scelto perché offre alcuni vantaggi nella specifica di un sistema multi-agente:

- *esecuzione del MAS*: poiché da un punto di vista astratto un linguaggio di programmazione logica è un linguaggio nondeterministico in cui la computazione avviene attraverso un processo di ricerca e l'evoluzione di un MAS è data da una successione nondeterministica di eventi;

- *capacità di meta-ragionamento*: la programmazione logica permette di vedere programmi come dati e gli agenti hanno necessità di poter modificare dinamicamente il loro comportamento per adattarlo ai cambiamenti nell'ambiente;

- *razionalità e reattività degli agenti* che sono collegate strettamente all'interpretazione *diluigitiva* e *operazionale* dei programmi logici in quanto un programma logico puro si può vedere come specifica della componente razionale di un agente e la visione operazionale dei programmi logici può essere usata per modellare il comportamento reattivo di un agente.

*Notazioni e assunzioni.* Le variabili iniziano con la lettera maiuscola. Si introducono:

- *predicati meta-logici* per testare l'identità di variabili e l'unificabilità di termini, per chiamare goal e per denotare congiunzioni di goal;
- *predicati extra-logici* per aggiornare lo stato dell'agente;
- *predicati del second'ordine* per recuperare un insieme di elementi che soddisfano una data condizione;
- *predicato di negazione* che è implementato usando `cut` e `fail`.

Si definisce un insieme di primitive standard per le strutture dati usate comunemente:

- *primitive per code*;
- *primitive per insiemi e relazioni*;
- *primitive per stacks*;
- *primitive per alberi*;
- *primitive per sequenze.*

Si forniscono inoltre delle primitive specifiche di Coo-BDI per lo scambio di piani e la socialità:

- `multicastRequestOp(AgentIdSet, Request)`: a partire da un insieme di identificatori di agenti `AgentIdSet` e una richiesta `Request` inserisce una `requested(Request)` nella event queue di ogni agente nell'insieme `AgentIdSet`;
- `provideOp(ProviderId, request(RequestId, AgentId, Desire), Instances)`: prende un identificatore d'agente `ProviderId`, una richiesta `request(RequestId, AgentId, Desire)` e un insieme di istanze `Instances` ed inserisce `provided(ProviderId,-request(RequestId, AgentId, Desire), Instances)` nella event queue di `AgentId`;
- `sendOp(SenderId, ReceiverId, Message)`: prende due identificatori d'agente, `SenderId` e `ReceiverId`, e un messaggio `Message` ed inserisce `received(SenderId, Message)` nella event queue di `ReceiverId`. `Message` può essere un qualsiasi tipo di termine contenente eventualmente variabili libere che implementano le informazioni da passare in quanto per il momento non ci si conforma ad alcun linguaggio di comunicazione per agenti specifico.

Infine, sono disponibili le primitive:

- `newId(Identity)` per generare una nuova identità;

- `ground(Literal)`: che ha valore vero se `Literal` (`Atom` o `not(Atom)`) è ground;

- `apply(Theta, Term, TermTheta)`: prende una sostituzione `Theta` e un termine `Term` e restituisce il termine ottenuto da `Term` rimpiazzando ogni variabile `X` con `Theta(X)`;

- `compose(Sigma, Theta, SigmaTheta)`: prende due sostituzioni `Sigma` e `Theta` e restituisce la loro composizione;

- `mgu(Expr1, Expr2, Mgu)`: prende due espressioni e restituisce il loro most general unifier.

*Interprete Coo-BDI.* Come detto in precedenza, differisce dall'interprete classico del modello BDI in quanto tiene conto anche della generazione dei desideri e della cooperazione attraverso la gestione della event queue, delle intenzioni sospese e di quelle attive.

```
cooBDIengine :- processEventQueue, processSuspendedIntentions,
  processActiveIntentions.
```

*Gestione eventi.* Se la coda degli eventi è vuota non si fa nulla, altrimenti si prende dalla coda un evento `Event`, lo si elabora e si aggiorna la coda degli eventi dell'agente.

```
processEventQueue :- eventQueue(EventQueue), empty(EventQueue).

processEventQueue :- eventQueue(EventQueue), not(empty(EventQueue)),
  get(Event, EventQueue, RemainingEventQueue), manageEvent(Event),
  retract(eventQueue(EventQueue)),
  assert(eventQueue(RemainingEventQueue)).
```

La gestione degli eventi può condurre a tre situazioni:

1) Se `Event` è di tipo `requested(request(RequestId, RequestingAgentId, Desire))`, allora il predicato `getRelInstance(Instances, request(RequestId, RequestingAgentId, Desire))` recupera tutte le istanze di piani specifici rilevanti per `Desire` il cui specificatore d'accesso è `public` o `only(TrustedAgents)`, e `TrustedAgents` include `RequestingAgentId`. L'insieme `Instances` recuperato è messo nella coda degli eventi di `RequestingAgentId` chiamando `provideOp(AgentId, request(RequestId, RequestingAgentId, Desire), Instances)`, in cui `AgentId` è l'identificatore dell'agente che porta a termine l'azione `provideOp`.

```
manageEvent(requested(request(ReqId, ReqAgentId, Desire))) :-
  agentId(AgentId),
  getRelInstances(Instances, request(ReqId, ReqAgentId, Desire)),
  provideOp(AgentId, request(ReqId, ReqAgentId, Desire), Instances).
```

2) L'`Event` è di tipo `providedOp(ProvidingAgentId, request(ReqId, AgentId, Desire), RetrievedInstances)`. Se esiste un'intenzione `Intention` tale che l'identificatore dell'intenzione e la richiesta `request(ReqId, AgentId, Desire)` appartengono alla relazione `relationIntentionRequest` si avvia il recupero delle istanze di piano rilevanti per `Desire`. `Intention` è aggiornata aggiungendo le `Instances` recuperate alle istanze rilevanti e rimuovendo l'identificatore dell'agente che ha risposto `(ProvidingAgentId)` dall'insieme `WaitingOnAgents`. L'intenzione aggiornata sostituisce `Intention`. Altrimenti, se non c'è `Intention` tale che l'identificatore dell'intenzione e la richiesta appartengono a `relationIntentionRequest`, l'evento è ignorato.

```
manageEvent(provided(ProvidingAgentId, request(ReqId, AgentId,
        Desire), Instances)) :-
 relationIntentionRequest(IntentionId, request(ReqId, AgentId, Desire),
 isIntention(intention(IntentionId, Stack, Status, RelevantInstances,
   WaitingOnAgents, FailedInstances)),
 singleton(ProvidingAgentId, SingletonProvidingAgentId),
 setDifference(WaitingOnAgents, SingletonProvidingAgentId,
   UpdatedWaitingOnAgents),
 setUnion(Instances, RelevantInstances, UpdatedRelevantInstances),
 retract(isIntention(intention(IntentionId, Stack, Status,
   RelevantInstances, WaitingOnAgents, FailedInstances))),
 assert(isIntention(intention(IntentionId, Stack, Status,
   UpdatedRelevantInstances, UpdatedWaitingOnAgents, FailedInstances))).
```

3) Se `Event` è un evento ordinario, si recupera l'insieme dei desideri corrispondenti mediante `getDesires(OrdinaryEvent, DesireSet)` e l'insieme dei desideri principali viene aggiornato in modo da contenerli. Per ogni desiderio generato che non è ancora nell'insieme dei desideri principali si crea e si inizializza un nuovo stack delle intenzioni `createIntention`, l'insieme delle intenzioni e la relazione `DesireIntention` sono aggiornate per tener conto delle nuove intenzioni create e il recupero di istanze di piano rilevanti per l'intenzione e il desiderio viene avviata (mediante la chiamata a `retrieveRelevantInstance(Intention, Desire)` all'interno di `createOneIntentionForOneDesire(Desire)` richiamata a sua volta da `createIntentionsForDesires(DesireSet)`).

```
manageEvent(Event) :- Event \= provided(_,_,_), Event \= requested(_),
  getDesires(Event, DesireSet), createIntentionsForDesires(DesireSet).
```

Quando si crea una nuova intenzione le sue componenti `RelevantInstances`, `WaitingOnAgents` e `FailedInstances` assumono come valore l'insieme vuoto e il suo stato ha valore `suspended`.

```
createIntention(IntentionId) :-
  assert(isIntention(intention(IntentionId, emptyStack, suspended,
  emptySet, emptySet, emptySet))).
```

Aggiornare un'intenzione recuperando le istanze di piano rilevanti per il desiderio che l'intenzione sta attualmento tentando di soddisfare (sia esso principale o subalterno) significa assegnare all'intenzione il valore `suspended`, recuperare le istanze di piano locali rilevanti per il desiderio chiamando `getLocalRelevantInstances(Desire, LocalRelevantInstances)`, assegnare l'insieme restituito alla componente `RelevantInstances` e aggiornare la componente `WaitingOnAgents` secondo la politica di recupero dei piani. Se la politica di recupero è `always` oppure non si sono trovate istanze locali rilevanti, la componente `WaitingOnAgents` è aggiornata con il valore dell'insieme degli agenti fidati; si invia allora una richiesta piani per il desiderio ad ogni agente fidato (`multicastRequestOp(TrustedAgents, Request)`) e si aggiorna la relazione `relationIntentionRequest`. In caso contrario, l'insieme `WaitingOnAgents` è inizializzato all'insieme vuoto e non si inoltrano richieste di piano.

```
retrieveRelevantInstances(IntentionId, Desire) :- agentId(AgentId),
  getLocalRelevantInstances(Desire, LocalRelInst),
  (retrievalPolicy(always); specificPlans(LocalRelInst, emptySet)),
  trustedAgents(TrustedAgents), newId(RequestId),
  multicastRequestOp(AgentId, TrustedAgents,
    request(RequestId, AgentId, Desire)),
  retract(isIntention(intention(IntentionId, Stack, _Status,
    _RelevantInstances, _WaitingOnAgents, FailedInstances))),
  assert(isIntention(intention(IntentionId, Stack, suspended,
    LocalRelInst, TrustedAgents, FailedInstances))),
  assert(relationIntentionRequest(IntentionId,
    request(RequestId, AgentId, Desire))).

retrieveRelevantInstances(IntentionId, Desire) :-
```

```
  getLocalRelevantInstances(Desire, LocalRelevantInstances),
  retrievalPolicy(noLocal),
  not(specificPlans(LocalRelevantInstances, emptySet)),
  retract(isIntention(intention(IntentionId, Stack, _Status,
    _RelevantInstances, _WaitingOnAgents, FailedInstances))),
  assert(isIntention(intention(IntentionId, Stack, suspended,
     LocalRelevantInstances, emptySet, FailedInstances))).
```

*Gestione delle intenzioni sospese.* Durante questo passo l'interprete controlla se ci sono intenzioni sospese che possono essere ripristinate in quanto la richiesta `Request` associata, la componente `WaitingOnAgents` e la componente `RelevantInstances` soddisfano la condizione `canResume`. Se non si trova un'intenzione ripristinabile, `processSuspendedIntentions` procede senza fare nulla.

```
processSuspendedIntentions :-
  isIntention(intention(IntentionId, _Stack, _Status, RelevantInstances,
  WaitingOnAgents, _FailedInstances)),
  relationIntentionRequest(IntentionId, Request),
  canResume(Request, WaitingOnAgents, RelevantInstances),
  resume(IntentionId, Request).
```

Se si trova un'intenzione ripristinabile, si possono verificare due casi:

1) Lo stack dell'intenzione è vuoto: il desiderio per cui si sono raccolte le istanze di piano rilevanti è un desiderio principale. Per eseguire il meccanismo di backtracking sulle istanze di piano che possono essere usate per soddisfare il desiderio principale, non si riprovano le istanze di piano per le quali si è già verificato un fallimento. Le istanze di piano applicabili sono quindi determinate a partire dalla collezione di istanze rilevanti `RelevantInst` meno le istanze fallite.

```
resume(IntentionId, Request) :-
  isIntention(intention(IntentionId, emptyStack, _Status,
  RelevantInst, _WaitingOnAgents, FailedInst)),
  setDifference(RelevantInst, FailedInst, NotAttemptedInst),
  getApplInstances(ApplicableInstSet, NotAttemptedInst),
  manageApplicableInstances(ApplicableInstSet, IntentionId, Request).
```

2) Lo stack dell'intenzione non è vuoto: il desiderio per cui si sono raccolte le istanze di piano rilevanti è un desiderio subalterno e per esso non viene eseguito alcun meccanismo di backtracking. Le istanze di piano applicabili sono determinate a partire da `RelevantInst`.

```
resume(IntentionId, Request) :-
  isIntention(intention(IntentionId, Stack, _Status, RelevantInst,
  _WaitingOnAgents, _FailedInst)), not(empty(Stack)),
  getApplInstances(ApplicableInstSet, RelevantInst),
  manageApplicableInstances(ApplicableInstSet, IntentionId, Request).
```

L'atomo `getApplInstances(ApplicableInstSet, RelevantInst)` unifica `ApplicableInstancesSet` con le coppie `(Plan, Substitution)` ottenute da `RelevantInst` in modo che la precondizione del piano istanziata con `Substitution` sia una conseguenza logica ground dei belief dell'agente. Per definizione, per ogni desiderio esiste almeno un'istanza di piano rilevante (quella originata da un piano di default) e questa istanza è anche applicabile (la precondizione è sempre vera): `getApplInstances` applicata a un insieme non vuoto di istanze di piano rilevanti restituisce sempre almeno un elemento. `ApplicableInstanceSet` può essere vuoto solo se tutte le istanze rilevanti per il desiderio principale sono fallite.

Se l'insieme delle istanze applicabili è vuoto il desiderio fallisce: desiderio ed intenzione vengono rimossi dai corrispondenti insiemi e le relazioni che li coinvolgono vengono aggiornate.

```
manageApplicableInstances(emptySet, IntId, request(ReqId, AgId, Des)) :-
  isDesire(Des), isIntention(intention(IntId, _Stack, _Status,
  _RelevantInstances, _WaitingOnAgents, _FailedInstances)),
  relationDesireIntention(Des, IntId),
  relationIntentionRequest(IntId, request(ReqId, AgId, Des)),
  retract(isDesire(Des)),
  retract(isIntention(intention(IntId, _Stack, _Status,
  _RelevantInstances, _WaitingOnAgents, _FailedInstances))),
  retract(relationDesireIntention(Des, IntId)),
  retract(relationIntentionRequest(IntId, request(ReqId, AgId, Des))).
```

Se l'insieme delle istanze applicabili non è vuoto si preleva un'istanza e se ne avvia il processo di esecuzione mediante una serie di inizializzazioni. assegnando il suo identificatore al campo istanza il valore dell'istanza di piano scelta, al campo sostituzione il contenuto di `Substitution`, impostando come stato corrente la radice del corpo del piano selezionato e come prossimo stato il figlio dello stato corrente. L'intenzione, ora pronta per l'esecuzione, è aggiunta in cima allo stack delle intenzioni con stato `active`. Si aggiorna infine la libreria dei piani con il piano selezionato seguendo la politica di acquisizione dell'agente. Il piano può essere aggiunto alla libreria oppure scartato o ancora utilizzato in sostituzione di tutti i piani aventi il trigger unificante con il suo.

```
manageApplicableInstances(ApplicableInstancesSet, IntentionId,
  _Request) :- ApplicableInstancesSet \= emptySet,
  selectInstance(ApplicableInstancesSet, (Plan, Substitution)),
  createExecution((Plan, Substitution), Execution),
  isIntention(intention(IntentionId, Stack, Status, RelevantInstances,
  WaitingOnAgents, FailedInstances)),
  push(Execution, Stack, UpdatedStack), acquirePlan(Plan),
  retract(isIntention(intention(IntentionId, Stack, Status,
  RelevantInstances, WaitingOnAgents, FailedInstances))),
  assert(isIntention(intention(IntentionId, UpdatedStack, active,
  RelevantInstances, WaitingOnAgents, FailedInstances))).
```

*Processing Active Intentions.* Le intenzioni il cui stato è `active` sono processate come in dMARS. Se non ci sono intenzioni attive non si fa nulla, altrimenti se ne sceglie una e si manda in esecuzione il piano al top del suo stack. Si possono verificare tre casi.

1. Lo stato corrente è una foglia: l'istanza di piano conclude la propria esecuzione con successo e viene rimossa dallo stack dell'intenzione. La sostituzione associata all'intenzione per l'esecuzione viene applicata alla sequenza di azioni interne che sarà quindi eseguita. Se non ci sono altri elementi sullo stack, l'intenzione si è conclusa con successo e può essere rimossa insieme al desiderio principale che l'ha generata (si deve aggiornare la relazione `relationDesireIntention`). In caso contrario l'esecuzione dell'intenzione deve procedere componendo la propria sostituzione con la sostituzione dell'istanza appena terminata..

2. L'istanza in esecuzione al top dello stack non verifica l'invariante o lo stato corrente non è una foglia e non ci sono stati da esso raggiungili: l'istanza di piano fallisce. La sostituzione associata per l'esecuzione viene applicata alle azioni per il fallimento del piano che sono quindi eseguite; infine, si aggiorna l'insieme delle istanze di piano fallite con l'istanza alla base dello stack dell'intenzione e si rimuovono tutti gli elementi dello stack. Si provano tutte le alternative per il soddisfacimento del desiderio principale associato all'intenzione.

3. L'istanza in esecuzione al top dello stack non fallisce né ha successo: si recupera l'azione da compiere che etichetta l'arco tra lo stato corrente e lo stato successivo scelto. L'azione può essere

   (a) `add(BeliefFormula)` o `remove(BeliefFormula)`: `BeliefFormula`, che al momento della chiamata dell'operazione deve essere ground, è ag-

giunta a (rimossa da) l'insieme dei belief dell'agente. L'agente manda a se stesso un messaggio per notificare che l'insieme dei belief è cambiato.

(b) `send(ReceiverAgentId, Message)`: viene chimaato `agentId(AgentId)` per trovare l'identità del mittente e si chiama `sendOp(ReceiverAgentId, AgentId, Message)` per spedire il messaggio.

(c) `query(SituationFormula)`: se `SituationFormula` istanziata con la sostituzione per l'esecuzione corrente può essere resa ground, e l'istanza ground è una conseguenza logica dei belief dell'agente, allora la sostituzione per l'esecuzione è aggiornata componendola con la sostituzione di base. Altrimenti si seleziona lo stato successivo dall'insieme degli stati raggiungibili dallo stato corrente.

(d) `achieve(Desire)`: se `Desire` instanziato con la sostituzione per l'esecuzione corrente può essere reso ground, e l'istanza ground è una conseguenza logica dei belief dell'agente, allora la sostitutuzione per l'esecuzione è aggiornata componendola con la sostituzione di base. Altrimenti si attiva il meccanismo per il recupero di istanze per il desiderio istanziato con la sostituzione per l'esecuzione corrente.

Al termine dell'esecuzione di un'azione, si aggiorna l'istanza di piano correntemente in esecuzione.

## 1.5 Vantaggi dell'estensione Coo-BDI

Coo-BDI è particolarmente adatto per modellare "learning agent", agenti che apprendono dall'interazione con altri agenti, con la facoltà di aggiornare dinamicamente la libreria dei piani. L'estensione Coo-BDI consente di modellare il comportamento umano e questo implica sicuramente dei vantaggi che possono essere evidenziati con l'analisi dell'esempio seguente.

Si vuole modellare questa situazione:

1. Viviana e Davide ricevono una e-mail da Paolo che li invita a raggiungerlo a casa di Luigi per una festa alle 19.

2. Viviana vorrebbe andare alla festa: desidera cancellare tutti gli appuntamenti presi in precedenza per questa sera e raggiungere la casa di Luigi alle 19. Invence, a Davide non piacciono le feste: desidera telefonare a casa di Luigi per declinare l'invito.

3. Viviana sa come concellare gli appuntamenti precedenti.

4. Viviana non sa come raggiungele la casa di Luigi. Davide non sa come telefonare a casa di Luigi.

Per ognuna delle affermazioni sopra si deve mostrare come possono essere modellate in BDI e in Coo-BDI per poi discutere il comportamento degli interpreti di entrambi i modelli sul modello dato per gli agenti.

*Viviana e Davide ricevono un e-mail da Paolo.* Sia in BDI, sia in Coo-BDI, la ricezione di un e-mail da Paolo si modella con un evento

```
received(paolo, message(by(e-mail),
     content(join_party, luigi_home, today, 7pm)))
```

entro le event queue di Viviana e di Davide.

*Viviana desidera cancellare gli appuntamenti presi in precedenza e raggiungere casa di Luigi. Davide desidera telefonare a Luigi e declinare l'invito.* In BDI non c'è un modo chiaro di modellare una corrispondenza tra eventi esterni e relativi desideri. Questa relazione può essere modellata in modo implicito definendo un piano il cui trigger sia l'evento esterno e il cui corpo contenga il desiderio da esaudire. I piani BDI sono rappresentati da `plan(Trigger, Precondition, Body, Invariant, SuccessActionSe quence, FailureActionSequence)`. Per una miglior leggibilità i corpi dei piano sono rappresentati graficamente come alberi. Il piano BDI per Viviana potrebbe essere:

```
plan(received(Sender, message(by(CommunicationMeans),
     content(join_party, Where, Day, Hour))),    ⇐ Trigger
   true,                                         ⇐ Precondition
   s0 •
     ↓ achieve(del_appointments(Day, Hour))
   s1 •                                          ⇐ Body
     ↓ achieve(go_to(Where, Day, Hour))
   s2 •,
   true, emptySeq, emptySeq)                     ⇐ Inv., Succ. and Fail.
```

Un piano simile, con corpo differente, può essere usato per simulare la generazione di desideri per Davide.

In Coo-BDI la relazione tra eventi esterni e desideri è rappresentata esplicitamente dal predicato `getDesire(OrdinaryEvent, DesireSet)` che ogni agente deve definire. La specifica dell'agente Viviana include:

```
getDesires(received(Sender,message(by(CommunicationMeans),
   content(join_party, Where, Day, Hour))),
   {achieve(del_appointments(Day, Hour),
   achieve(go_to(Where, Day, Hour))})
```

mentre quella dell'agente Davide include:

```
getDesires(received(Sender,message(by(CommunicationMeans),
   content(join_party, Where, Day, Hour))),
   {achieve(phone(Where, content(decline))})
```

*Viviana sa come cancellare gli appuntamenti precedenti.* In entrambi i modelli, il fatto che Viviana sappia come cancellare gli appuntamenti presi in precedenza significa che nella libreria dei piani di Viviana c'è almeno un piano attivato da `achieve(del_appointments(today, 7pm))`. Per esempio il piano mostrato nel seguito mostra come cancellare una lista di appuntamenti richieda il recupero della lista di persone e l'invio di un messaggio a ogni persona della lista. Un piano per gestire `achieve(get_appointment_list(Day, Hour, List))` dovrebbe appartenere alla libreria dei piani di Viviana e l'azione esterna `multicast(List, content(delete_appointment))` dovrebbe essere definita in termini di `send`.

Il piano Coo-BDI appare esattamente come quello BDI pi gli specificatori d'accesso.

```
plan(achieve(del_appointments(Day, Hour)),   ⇐ Trigger
   true,                                     ⇐ Precondition
   s0 •
     ↓ achieve(get_appointment_list(Day, Hour, List))
   s1 •                                                ⇐ Body
     ↓ multicast(List, content(delete_appointment))
   s2 •,
   true, emptySeq, emptySeq)                 ⇐ Inv., Succ. and Fail.
```

*Viviana non sa come raggiungere casa di Luigi. Davide non sa come telefonare a casa di Luigi.* In entrambi i modelli questa situazione è modellata dall'assenza, nella libreria dei piani di Viviana, di piani attivati dal desiderio `achieve(go_to(luigi_home, today, 7pm))` e dall'assenza, nella libreria dei piani di Davide, di piani attivati dal desiderio `achieve(phone(Where, content(decline))`.

La sostanziale differenza tra Coo-BDI e BDI sta nel modo in cui è gestita la mancanza di piani rilevanti per il soddisfacimento di tali desideri.

Si consideri prima il caso di Viviana. In molti sistemi BDI, Viviana può scartare questo desiderio o adottare un piano di default. In nessun sistema di nostra conoscenza è previsto il recupero di pian dall'esterno. Scartare il desiderio non

modella correttamente il comportamento umano consueto: nessuno rinuncerebbe ad andare ad una festa perché non sa come raggiungere il luogo dove si tiene la festa (in genere...). Usare un piano di default potrebbe essere una soluzione ma solitamente i piani di default sono progettati per essere utilizzabili in tutte le situazioni. Questo richiede molta intuizione per prevedere tutte queste situazioni in anticipo. Per esempio, un piano di default che dice "cerca nelle pagine bianche/gialle" potrebbe essere utile in molte situazioni, compresa quella dell'esempio, ma è completamente inutile per esaudire desideri del tipo "prepara una torta" o "supera un esame".

Estendere il modello BDI con la cooperazione aiuta a superare la mancanza di piani rilevanti in modo simile al comportamento umano. Si supponga che la strategia di cooperazione di Viviana sia caratterizzata da:

```
trustedAgents({paolo, davide, sonia})
retrievalPolicy(noLocal)
acquisitionPolicy(add)
```

Poiché Viviana non ha piani locali rilevanti per il desiderio `achieve(go_to(luigi_home, today, 7pm))`, il recupero di piani esterni viene avviato:

- si crea una richiesta `request(go_to_luigi_request, valentina, achieve(go_to(Luigi_home, today, 7pm)))` per ottenere piani rilevanti per il desiderio.
- la richiesta è inviata ad ogni agente dell'insieme {`paolo, davide, sonia`} per mezzo della primitiva `multicastRequestOp`.
- alla componente `WaitingOnAgent` dell'intenzione `go_to_luigi_intention` è assegnato l'insieme {`paolo, davide, sonia`}.
- l'intenzione `go_to_luigi_intention` viene sospesa.

Il risultato del processo cooperativo di recupero piani dipende dal contenuto delle librerie dei piani di Paolo, Davide e Luca. Paolo ha solo il piano rilevante pubblico mostrato sotto. Quando riceve la richiesta di Viviana egli manda l'istanza di piano corrispondente a Viviana.

```
plan(public,                                    ⇐ Access specifier
   achieve(go_to(luigi_home, Day, Hour)),       ⇐ Trigger
   true,                                        ⇐ Precondition
   s0 •
     ↓ achieve(drive_to(collins_street, Day, (Hour-30min)))
```

```
s1 •                                                    ⇐ Body
   ↓ achieve(park_near(main_church))
s2 •,
true, emptySeq, emptySeq)                    ⇐ Inv., Succ. and Fail.
```

Davide ha solo un piano privato per raggiungere casa di Luigi e di conseguenza manda a Viviana un insieme di istanze vuoto. Infine, Luca ha un unico piano rilevante mostrato sotto. Poiché Viviana è inclusa tra le persone autorizzate a ricevere il piano, Luca manda l'istanza corrispondente ad esso a Viviana.

```
plan(only({valentina, alberto}),             ⇐ Access specifier
   achieve(go_to(luigi_home, Day, Hour)),    ⇐ Trigger
   true,                                     ⇐ Precondition
   s0 •
      ↓ achieve(take_a_bus(number(168), Day, (Hour-40min)))
   s1 •                                                 ⇐ Body
      ↓ achieve(get_off(collins))
   s2 •,
   true, emptySeq, emptySeq)                 ⇐ Inv., Succ. and Fail.
```

Quando Viviana riceve tutte e tre le risposte, procede come segue:

- realizza che l'intenzione `go_to_luigi_intention` può essere ripristinata.
- genera tutti i piani applicabili dall'insieme di quelli rilevanti (i due piani ricevuti da Paolo e Luca pi tutte le istanze di piano di default). In questo semplice esempio, tutti i piani rilevanti sono anche applicabili.
- seleziona uno dei piani specifici applicabili associati all'intenzione.
- se il piano scelto è uno di quelli esterni (i soli piani specifici applicabili sono quelli ricevuti da Paolo e Luca) lo memorizza.

Probabilmente, un umano si comporterebbe in modo simile: contatterebbe alcuni amici chiedendo consigli, sceglierebbe le istruzioni pi convenienti e memorizzerebbe le istruzioni per ritilizzarle in futuro.

Va osservato che questo comportamento introdotto da Coo-BDI non può essere simulabile nei classici sistemi BDI. Questo avvalora ulteriormente l'idea dell'estensione cooperativa. Tornando all'esempio, davide si comporterà come Viviana per ottenere il numero telefonico di casa di Luigi e declinare l'invito. Se non si separano gli eventi dai desideri, l'attitudine completamente diversa di Viviana e Davide non può essere catturata facilmente. Supponendo che la ricezione dell'evento `received(paolo, message(by(e-mail), content(join_party, luigi_home, today, 7pm)))` attivi un piano come nel caso BDI e che né Viviana né Davide posseggano

un piano rilevante per esso. La strategia cooperativa implementata in Coo-BDI porterebbe alla richiesta di piani per la gestione di tale evento a tutti gli agenti fidati. Questo però ha poco senso in quanto i piani rilevanti recuperati rappresenterebbero suggerimenti veramente eterogenei quali declinare l'invito, raggiungere la festa, chiedere di posticipare la festa o inventare rapidamente una scusa. Come fa un agente a scegliere il piano corretto tra questi se non conosce quale stato vuole raggiungere? Associando gli eventi esterni con i desideri generati internamente e cooperando per esaudire i desideri si risolve pertanto il problema.

# Capitolo 2

# I Web Service

*Introduciamo qui il concetto di Web service. Vengono presentati l'architettura Web service ed i principali standard impiegati nello sviluppo di CooWS*

## 2.1 Introduzione

### 2.1.1 Cenni storici

Un breve cenno alla storia delle architetture software chiarirà meglio le motivazioni ed il ruolo dei Web service.

#### Database e client-server application (anni '80)

Le prime applicazioni client-server, risalenti ai primi anni 80, si basavano su un'architettura client-database (fig. 2.1). In tale struttura l'applicazione client interagisce con il database utilizzando un linguaggio ad-hoc (tipicamente SQL [10]). Con questo approccio, di tipo two-tier (due livelli), gli sviluppatori demandano al database tutti i compiti relativi all'archiviazione dati. Una configurazione tipica prevede un server sul quale è installato il database ed un insieme di pc client sui quali gira la client-application. La connessione client-server avviene tramite linea dedicata (Local Area Network, abbreviato con LAN).

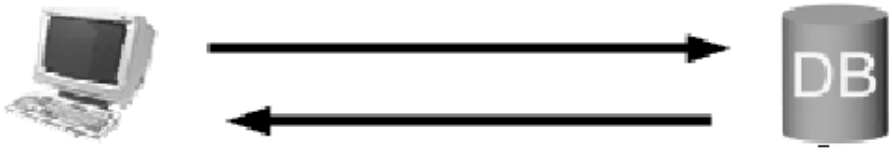



Figura 2.1: Architectura Client-Server

#### Web application server (anni '90)

È negli anni novanta che Internet prende piede e con lei tutta una serie di applicazioni e servizi orientati al Web. Una delle prime architetture Web adottate è stata la CGI (Common Gateway Interface). Qui l'applicativo client è il Web browser mentre sul server è installato un Web server con supporto CGI (fig. 2.2). La grande differenza tra questo modello ed il precedente è la tipologia di interconnessione. In quest'ultimo le applicazioni Web comunicano mediante Internet - un'enorme rete pubblica basata su protocolli standard aperti quali ad esempio HTTP [11] su TCP/IP [12].

Mentre le architetture client-server di prima generazione potevano gestire centinaia/migliaia di connessioni simultanee, e ciò risultava essere un buon inidice i contesti di applicazione (LAN), nel caso delle applicazioni Web tale limite massimo viene amplificato dall'enorme dimensione della rete Internet. In certi frangenti un

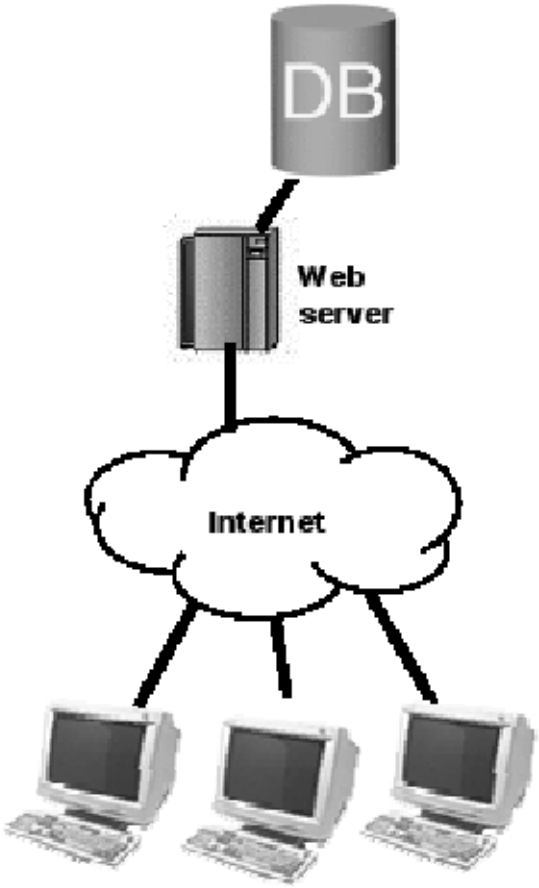



Figura 2.2: Architettura Web Server

server può ricevere centinaia di migliaia sino a milioni di connessioni simultanee. Di consenguenza un Web server deve fornire buone prestazioni in termini di scalabilità, bilanciamento del carico e gestione delle transazioni (laddove è presente un database). Per soddisfare queste nuove esigenze sono nati gli "Application Server" (AS, fig. 2.3). Uno degli AS piu noti, utilizzato in questo lavoro di tesi, è Java 2 Enterprise Edition (J2EE [13]) il quale fornisce un ambiente di sviluppo completo orientato al Web.

### L'avvento dei Web service

L'architettura sofware dei Web service è il risultato della convergenza tra due tecnologie consolidate.
Da una parte il Web con standard de facto quali HTTP come procollo di comu-

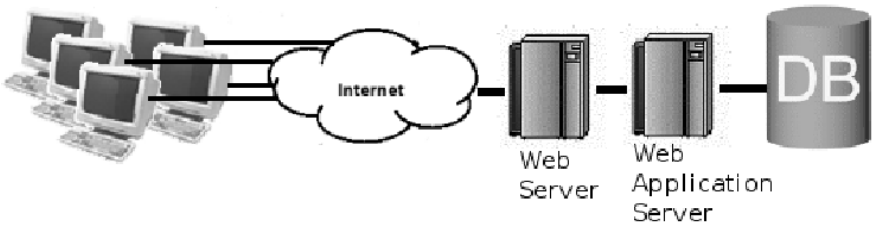



Figura 2.3: Architettura Web Application Server

nicazione e XML [14] come linguaggio per la rappresentazione dell'informazione, oggigiorno strumenti essenziali per comunicazioni indipendenti dalla piattaforma. Dall'altra l'evoluzione del Service Oriented Computing (SOC), dove dati e servizi sono forniti attraverso ben definite interfacce. Principale obiettivo del SOC è la comunicazione applicatione-applicazione (senza intervento umano). Mentre le architetture precedenti trovano applicazione in contesti "Business To Consumer" (B2C), con il SOC si viene incontro a problematiche tipiche del "Business To Business" (B2B), quali ad esempio l'"Enterprise Application Integration" (EAI) dove la finalità ultima è l'integrazione tra differenti componenti software interni ed esterni ad una organizzazione. Tecnologie orientate a tale integrazione sono DCOM[15], DCE[16], CORBA e IDL[17], RPC[18], RMI[19], MOM[20]. Nonostante l'importanza che tali tecnologie hanno avuto per far decollare il paradigma "service oriented", esse mostrano gravi limitazioni per quanto concerne l'interoperabilità. La natura "verticale" di tali piattaforme vanifica infatti uno degli aspetti più importanti del SOC, l'integrazione appunto.
Ed è qui che subentra l'architettura Web service.

## 2.2 Web service: concetti fondamentali

### 2.2.1 Alcune definizioni

> *"Dal mio punto di vista un Web service è un insieme arbitrario di risorse mirate all'elaborazione automatica (senza intervento umano), così come un sito Web rappresenta un insieme arbitrario di risorse organizzato per una consultazione umana"* (Paul Prescod [21])

> *"Un Web service è un interfaccia che descrive un insieme di operazioni accessibili via rete mediante messaggi XML standardizzati. Un Web service è descritto utilizzando una notazione standard basata su*

*XML, chiamata "descrizione del servizio". Essa descrive tutti i dettagli necessari per utilizzare un servizio, incluso il formato di codifica dei messaggi, il protocollo di trasporto e le locazioni (dove si trovano le istanze del servizio). L'interfaccia nasconde i dettagli di implementazione del servizio, consentendo un utilizzo indipendente dalla piattaforma hardware e software che ospita il servizio e dal linguaggio di programmazione nel quale è scritto. ... I Web service possono adempiere ad un particolare o ad un insieme di compiti. Possono essere utilizzati sia singolarmente sia collettivamente creando aggregazioni complesse o transazioni commerciali"*

(Heather Kreger)

*"Un Web service è un sistema software progettato per automatizzare interazioni via rete. La sua interfaccia è descritta con un formato comprensibile ad una macchina (in particolare WSDL). Altri sistemi possono interagire con il Web service in maniera compatibile con tale descrizione, utilizzando messaggi SOAP, tipicamente spediti tramite protocollo HTTP, serializzati in XML, congiuntamente ad altri standard basati sul Web."*

(WS Gloassary - [22])

*"Il termine Web service descrive una sistema standard per l'integrazione di applicazioni basate sul Web, tramite l'adozione di standard aperti quali XML, SOAP, WSDL e UDDI. XML è utilizzato come sistema di descrizione per i dati, SOAP è utilizzato per trasferire i dati, WSDL è utilizzato per descrivere i servizi disponibili ed infine UDDI è utilizzato per elencare quali servizi sono disponibili. Utilizzati principalmente dalle aziende per integrare i propri processi con altre aziende e con i propri clienti, i Web service consentono fondamentalmente alle organizzazioni di scambiarsi informazioni senza necessariamente conoscere l'infrastruttura tecnologica che sta dietro al firewall di ognuna di esse."*

(Webopedia - [23])

### 2.2.2 Caratteristiche principali della WSA

Alla luce delle precedenti definizioni, le caratteristiche principali dell'architettura Web service è riassumibile nei seguenti punti:

- Un Web service è un servizio accessibile tramite Web. Le comunicazioni tra Web service avvengono in maniera indipendente dalla piattaforma e dal linguaggio di implementazione dei singoli servizi. Queste caratteristiche sono garanzia per una facile integrazione tra contesti applicativi eterogeneii.

- Un Web service fornisce una interfaccia che può essere invocata da un altro programma, sia esso applicazione client o altro Web service, realizzando così l'integrazione tra il Web e le diverse tecnologie che implementano i servizi.

- Un Web service è registrato presso un Web service Registry. Tramite questo registro i fruitori del servizio ("service consumer") possono localizzare dinamicamente i fornitori dei servizi ("service provider") a loro necessari.

- I Web service implementano un protocollo di comunicazione basato su scambio di messaggi fortemente disaccoppiato. Tale proprietà conferisce al modello flessibilità e adattabilità .

### 2.2.3 SOA e WSA

La Web service Architecture (WSA) [24] rappresenta lo stato dell'arte dell'implementazione di una Service Oriented Architecture (SOA) [25] (fig. 2.4)

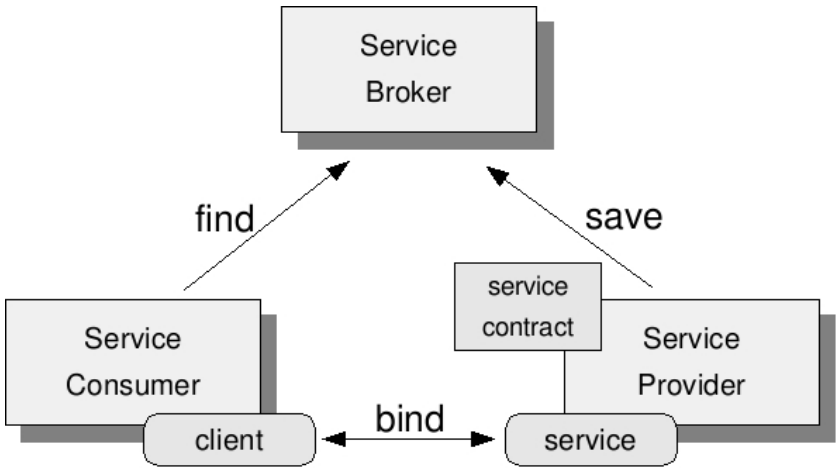



Figura 2.4: Architettura Service Oriented

L'architettura SOA è costituita da tre ruoli:

**Service Provider** Il service provider è l'applicativo che fornisce il servizio publicandone interfaccia e contratto presso un catalogo di servizi ("service

broker"). Da un punto di vista commerciale rappresenta il proprietario del servizio. Dal punto di vista dell'architettura rappresenta la piattaforma che ne fornisce l'accesso.

**Service Requestor** Il service requestor (o service consumer) è l'applicativo che invoca un servizio, previa localizzazione presso un service broker. Da un punto di vista commerciale è un cliente che necessita di particolari funzionalità. Dal punto di vista dell'architettura è una applicazione che ricerca o interagisce con un Web service. Il ruolo di service requestor può essere rivestito ad esempio da un web browser oppure da un programma senza interfaccia utente, quale ad esempio un altro servizio.

**Catalogo di servizi** (Service Broker) Il service broker implementa un registro per la catalogazione dei servizi. I service provider vi pubblicano le informazioni sui servizi forniti. I service requestor effettuano ricerche per localizzare servizi a loro congeniali.

Affinché i suddetti ruoli possano svolgere funzionalmente i loro compiti la SOA specifica inoltre tre componenti chiave:

**Trasporto** (Transport) Tale componente rappresenta i formati ed i protocolli utilizzati per comunicare con un servizio.

**Descrizione** (Description) Rappresenta il linguaggio utilizzato per descrivere i servizi. In tale linguaggio è possibile dichiarare le operazioni supportate dal servizio nonché i parametri ed il formato dei messaggi scambiati. Questo linguaggio deve avere un formato comprensibile ad una macchina in modo da consentire la generazione automatica di codice relativa a componenti quali ad esempio proxy per i client, skeleton, stubs[1], ecc. ecc. ....

**Scoperta** (Discovery) Implementa il meccanismo di pubblicazione e ricerca di un servizio.

L'architettura Web service (fig. 2.5) integra i concetti chiave della Service Oriented Architecture con il Web. Il risultato è un architettura che risolve molti dei limiti imposti da tecnologie verticali quali DCOM, DCE, CORBA o RMI, tutti middleware orientati al servizio ma di difficile integrazione. L'architettura Web service può quindi essere considerata una forma di Internet middleware, con

[1] proxy, skeleton e stubs sono componenti software che forniscono una interfaccia locale per comunicazioni con oggetti remoti nascondendo così al programmatore i dettagli relativi ai protocolli di basso livello

il vantaggio di non introdurre nuovi protocolli o codifiche proprietarie. Tutti i componenti funzionali della SOA vengono infatti implementati nella Web service Architecture (WSA) tramite SOAP (per il trasporto), WSDL (per la descrizione) e UDDI (per la scoperta), tutti linguaggi e formati basati su XML che saranno affrontati nella sezione seguente.

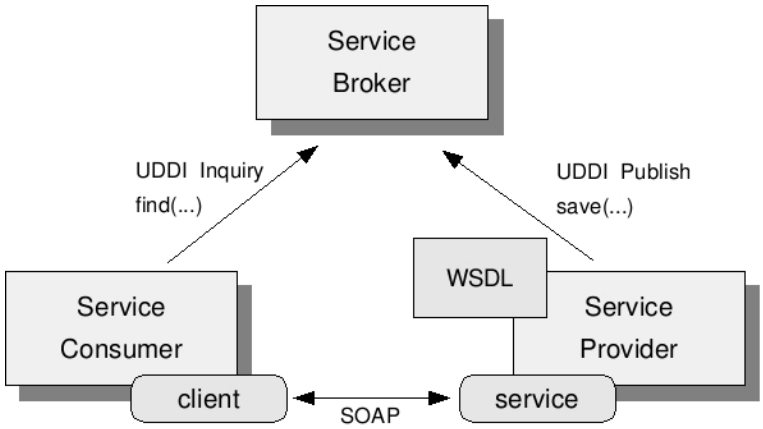



Figura 2.5: Architettura Web service

## 2.3 Tecnologie

### 2.3.1 Lo stack dei Web service

Le tecnologie impiegate nella Web service Architecture creano lo stack di fig. 2.6

Nel seguito vengono presentate le tecnologie più recenti e che hanno ricevuto maggiore consenso.

### 2.3.2 XML

XML [26] (eXtensible Markup Language) rappresenta il mattone fondamentale sul quale è costruito il modello dei Web service. In XML sono codificate le chiamate di procedura remota (con SOAP), sono definite le interfacce dei singoli servizi (con WSDL), le interfacce dei cataloghi di Web service (con UDDI) ed infine sono descritti i processi di integrazione tra diversi Web service (con BPEL). Grazie ad XML quindi si risolvono tutte le problematiche tipiche dettate dall'integrazione tra sistemi eterogenei.

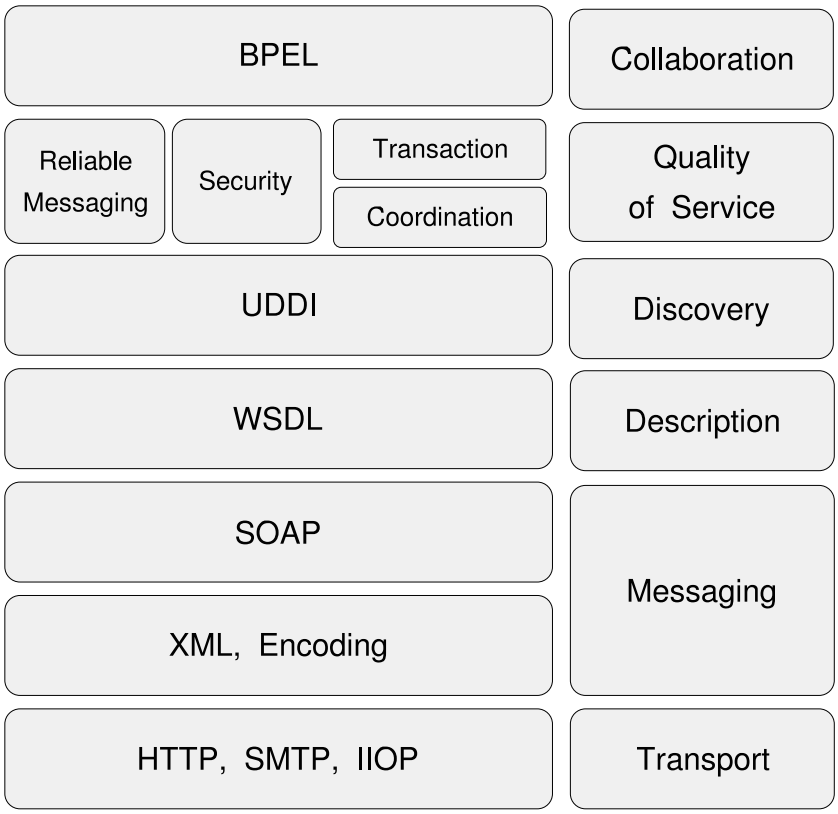



Figura 2.6: Lo stack dei Web service

### 2.3.3 SOAP

SOAP [27] è un protocollo per lo scambio di messaggi basato su XML. Un messaggio codificato in SOAP contiene, oltre all'informazione trasportata, ulteriori metainformazioni necessarie alla sua decodifica. Fondamentalmente SOAP supporta una comunicazione *peer-to-per* unidirezionale della forma mittente-destinatario. È tramite la combinazione di più messaggi SOAP che si implementano protocolli più complessi quali richiesta/risposta, notifica, ecc ecc.
La specifica di SOAP è suddivisa fondamentalmente in quattro sezioni **??**:

**SOAP Envelope** L'*envelope* fornisce un meccanismo per identificare e decodificare il contenuto di un messaggio. Un envelope SOAP è costituito da un *header* ed un *body*. Nell'*header* è possibile trovare informazioni relative alla sicurezza, all'instradamento, all'autenticazione, ecc ecc.

**SOAP Body** Contiene l'informazione vera e propria, oggetto della comunicazione

**SOAP Transport Binding Framework** La specifica di SOAP prevede il binding con protocolli per il trasporto quali *HTTP* ed *SMTP*. Il ricorso a tali protocolli di trasporto consente di mantenere una politica di integrazione con standard aperti e consolidati.

**SOAP Serialization Framework** Tramite la serializzazione i tipi di dato semplici, quali interi o stringhe, e complessi, quali array e strutture, di un qualsiasi linguaggio (ad esempio Java) vengono mappati in codice XML all'atto della spedizione per poi essere "ricostruiti" tramite il processo inverso di deserializzazione. SOAP prevede un insieme di regole per la codifica divenute col tempo uno standard de facto.

SOAP supporta due modalità per lo scambio di messaggi:

**SOAP Document** Messaggi che trasportano un documento da elaborare in remoto. È indicato per contesti dove è necessario un accoppiamento lasco tra le applicazioni coinvolte. Il mittente invia un documento senza preoccuparsi del formato di codifica (si veda "SOAP serialization framework") mentre sarà carico del destinatario elaborarlo opportunamente.

**SOAP RPC** Messaggi che trasportano comandi e parametri per l'invocazione di procedure remote, ad esempio secondo il modello RPC (Remote Procedure Call) [28]. Questa forma di messaggistica è invece orientata per comunicazioni ad accoppiamento stretto. La specifica SOAP per messaggi RPC stabilisce una convenzione per la codifica di chiamate e risposte RPC. Una chiamata RPC codificata in SOAP conterrà quindi un elemento XML "top-level" che rappresenterà il nome della procedura invocata (ad esempio `MiaProcedura`). Nella gerarchia del documento XML sarà poi possibile specificare gli eventuali parametri della procedura come elementi figli. La risposta RPC codificata in SOAP similarmente conterrà un elemento XML in riferimento al nome di procedura invocata (ad esempio `MiaProceduraResponse`) con gli eventuali elementi figli contenenti i parametri di ritorno.

Nel particolare contesto dei Web service, SOAP viene impiegato per la codifica delle informazioni necessarie a operazioni di *publish, find, bind* ed *invoke* di Web service.
Riassumiamo questo paragrafo con un semplice esempio. Consideriamo una procedura con la seguente interfaccia:

```
void dispatchEvent(String agentTo, EventRequested anEvent)
```

dove la classe EventRequested è così definita:

```
class EventRequested  {
    String mAgentID; // the sender
    String mRequestID;
    String desire;
}
```

Una sequenza di esempio di chiamata-risposta RPC codificata in SOAP è illustrata dal codice seguente:

Richiesta codificata in SOAP: dichiarazione dell'envelope

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

Richiesta codificata in SOAP: inizio della sezione body contenente l'invocazione della procedura dispatchEvent

```
 <soapenv:Body>
  <ns1:dispatchEvent
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="CooWS">
```

Richiesta codificata in SOAP: i parametri della chiamata

```
   <agentTo xsi:type="xsd:string">agent0</agentTo>
   <ns2:EventRequested href="#id0" xmlns:ns2="urn:coows"/>
  </ns1:dispatchEvent>
  <multiRef id="id0" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xsi:type="ns3:EventRequested"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns3="urn:coows">
   <MAgentID xsi:type="soapenc:string">agent0</MAgentID>
   <MDes href="#id1"/>
   <MReqID xsi:type="soapenc:string">req1</MReqID>
  </multiRef>
  <multiRef id="id1"  soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xsi:type="ns4:Desire"  xmlns:ns4="urn:coows"
```

```
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
   <MDes xsi:type="soapenc:string">desire0</MDes>
  </multiRef>
```

Richiesta codificata in SOAP: chiusura delle sezioni body ed envelope

```
 </soapenv:Body>
</soapenv:Envelope>
```

La corrispondente risposta RPC codificata in SOAP

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:dispatchEventResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="CooWS"/>
 </soapenv:Body>
</soapenv:Envelope>
```

### 2.3.4 WSDL

WSDL (Web service Description Language) è un linguaggio basato su XML per la descrizione delle interfacce di Web service. Un documento WSDL descrive in maniera strutturata *quali* funzionalità offre un *WS*, *come* devono essere accedute e *dove* sono accessibili.
Un documento WSDL è suddiviso in cinque sezioni fondamentali:

**Tipi** Tramite l'elemento `<types>` vengono definiti i tipi di dato che sono poi utilizzati per la definizione dei messaggi. WSDL utilizza di default il sistema dei tipi di dato definito nella *W3C XML Schema Part2: Data types Recommendation [26]*. È comunque possibile utilizzare qualsiasi altra codifica così come è possibile mappare qualsiasi codifica con i tipi di dato di un particolare linguaggio di programmazione. Gli strumenti che fanno uso di SOAP codificano e decodificano tramite queste informazioni.

**Messaggi** I messaggi vengono definiti tramite l'elemento `<message>`. Un messaggio può essere costituito da una o più sezioni ognuna delle quali associata ad un particolare tipo di dato (eventualmente definito in una precedente sezione `<types>`). I messaggi qui definiti saranno poi utilizzati come

parametri di input/output per le operazioni. Quando si utilizza il modello di comunicazione RPC la singola parte di un messaggio rappresenta un parametro della procedura.

**Tipi di porta** (Port Type) L'elemento `<portType>` definisce un insieme di operazioni. Ogni operazione contiene la specifica dei tipi di dato dei parametri di input/output. Nel modello RPC ogni operazione rappresenta un metodo.

**Associazioni** (Binding) L'elemento `<binding>` associa le operazioni precedentemente definite ad un particolare protocollo e formato di codifica dei dati. Ad esempio, è possibile associare un portType ad una specifica interfaccia RPC via SOAP utilizzando HTTP come protollo di trasporto e la codifica SOAP per la codifica dei tipi di dato.

**Servizio** (Service) L'elemento `<service>` infine raggruppa logicamente un insieme di porte. Un elemento `<port>` associa un binding alla locazione (specificata da un URL) di un particolare Web service.

Possiamo quindi definire un servizio come un insieme di *porte*. Ogni porta è definita astrattamente da un *tipo di porta* il quale supporta un insieme di operazioni. Ogni operazione può processare un insieme di messaggi. Un *binding* associa un *portType* ad un particolare protocollo. Una porta corrisponde all'istanza di un *portType* e di un *binding* associati ad un particolare indirizzo di rete.
Riprendendo l'esempio del paragrafo precedente, vediamo ora il documento WSDL associato all'interfaccia di un Web service che espone l'operazione *dispatchEvent (anAgentID, anEvent)*

WSDL: tipi di dato

```
<wsdl:definitions
targetNamespace="urn:coows"
xmlns:impl="urn:coows"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <wsdl:types>
  <schema targetNamespace="urn:coows"
xmlns="http://www.w3.org/2001/XMLSchema">
   <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
   <complexType abstract="true" name="Event">
    <sequence/>
   </complexType>
   <complexType name="Desire">
    <sequence>
     <element name="MDes" nillable="true" type="xsd:string"/>
    </sequence>
   </complexType>
   <complexType name="EventRequested">
    <complexContent>
     <extension base="impl:Event">
      <sequence>
       <element name="MAgentID" nillable="true" type="xsd:string"/>
       <element name="MDes" nillable="true" type="impl:Desire"/>
       <element name="MReqID" nillable="true" type="xsd:string"/>
      </sequence>
     </extension>
    </complexContent>
   </complexType>
  </schema>
</wsdl:types>
```

WSDL: messaggi e portType

```
<wsdl:message name="dispatchEventRequest">
      <wsdl:part name="agentTo" type="soapenc:string"/>
      <wsdl:part name="anEvent" type="impl:Event"/>
```

```
</wsdl:message>

<wsdl:message name="dispatchEventResponse">
</wsdl:message>

<wsdl:portType name="CooWS">
  <wsdl:operation name="dispatchEvent"
parameterOrder="agentTo anEvent">
    <wsdl:input message="impl:dispatchEventRequest"
name="dispatchEventRequest"/>
    <wsdl:output message="impl:dispatchEventResponse"
name="dispatchEventResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

WSDL: binding e service

```
<wsdl:binding name="CooWSSoapBinding" type="impl:CooWS">
  <wsdlsoap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="dispatchEvent">
       <wsdlsoap:operation soapAction=""/>
       <wsdl:input name="dispatchEventRequest">
         <wsdlsoap:body
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="urn:coows" use="encoded"/>
       </wsdl:input>
       <wsdl:output name="dispatchEventResponse">
          <wsdlsoap:body
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="urn:coows" use="encoded"/>
       </wsdl:output>
    </wsdl:operation>
</wsdl:binding>

<wsdl:service name="CooWSService">
  <wsdl:port binding="impl:CooWSSoapBinding" name="CooWS">
     <wsdlsoap:address
      location="http://localhost:8080/active-bpel/services/CooWS"/>
  </wsdl:port>
```

```
</wsdl:service>

</wsdl:definitions>
```

### 2.3.5 UDDI

UDDI (Universal Description, Discovery and Integration) rappresenta lo standard per la pubblicazione e ricerca di Web service. La specifica di UDDI si fonda su standard consolidati quali HTTP, XML, XML Schema [26], SOAP e WSDL. Tramite un registro UDDI è possibile mantenere traccia di informazioni e metainformazioni dei Web service, classificarli e catalogarli secondo i criteri più efficaci per il business di una azienda. Un registro UDDI è in verità a sua volta un Web service le cui operazioni consentono di cercare service provider, conoscere quali Web service essi forniscono ed infine recuperare le informazioni tecniche per una corretta invocazione di tali servizi. Uno scenario tipico vedrà quindi un client effettuare una ricerca (inquiry) sul registro per localizzare servizi corrispondenti ai propri criteri. Una volta ottenuta una lista di servizi, il client potrà effettuare la bind ed infine l'invocazione vera e propria per ognuno di essi.
UDDI gioca quindi un ruolo determinante nella realizzazione di una connessione fortemente disaccoppiata tra Web service.

### 2.3.6 BPEL

BPEL (Business Process Execution Language [29], si legge come la parola inglese "beeple") é il linguaggio divenuto standard de facto per l'"orchestrazione" dei Web service. Con il termine orchestrazione si indica un processo di coordinamento automatico mirato all'integrazione di processi preesistenti in un unico nuovo processo a valore aggiunto. Le applicazioni software classiche per la gestione di flussi di lavoro sono i cosidetti sistemi di Work Flow Management. In tali sistemi è possibile gestire contemporaneamente più istanze di processi regolamentandone il flusso esecutivo. Nella realtà del mercato attuale tali sistemi spesso sono confinati all'interno dei processi presenti nel contesto della singola organizzazione. Grazie alla regolamentazione degli standard di comunicazione introdotta con l'architettura Web service è facile estendere tali concetti a contesti più ampi coinvolgendo servizi forniti da diverse organizzazioni. BPEL gioca pertanto un ruolo di primo piano nella assimilazione della tecnologia Web service sul mercato.
BPEL definisce una grammatica basata su XML per la descrizione della logica di controllo per l'integrazione di Web service. In un documento BPEL si specifica un processo come una serie di attività e relazioni tra esse, con criteri per stabilire inizio e terminazione del processo. Un documento BPEL descrive quindi un vero e

proprio processo eseguibile. Tramite un motore BPEL [attualmente sono presenti sul mercato svariate implementazioni] è possibile interpretare, e quindi eseguire, documenti BPEL. Ogni processo BPEL è dotato di una interfaccia WSDL nella quale sono descritti i partner esterni coinvolti ed il nome del servizio associato al processo. Pertanto una volta installato su un motore BPEL sarà possibile invocare tale processo come un normale Web service.
Vediamo ora nello specifico la struttura di un documento BPEL:

```
<process ...>
    <partnerLinks>
        <partnerLink.../>
        <partnerLink.../>
              ...
    </partnerLinks>
    <variables>
        <variable.../>
        <variable.../>
             ...
    </variables>
    <faultHandlers>
 ...
    </faultHandlers>
    <activities ..../>*
</process>
```

Nella sezione *partnerLinks* sono dichiarati i Web service che vengono coordinati all'interno del processo. Nella successiva sezione *variables* si dichiarano i messaggi che saranno utilizzati come parametro di I/O durante le varie chiamate. Con i *faultHandlers* è invece possibile specificare i comportamenti del processo in caso di failure. I singoli faultHandler qui dichiarati possono essere richiamati all'interno delle *activities*. È nelle *activities* che il corpo del processo BPEL viene infine propriamente definito.
Le attività presenti all'interno di un processo BPEL appartengono a due categorie:

**Semplici**: il livello più semplice di interazione con il mondo esterno o di elaborazione dati

- **Receive**: il processo attende un messaggio
- **Reply**: il processo risponde ad un precedente messaggio (insieme a receive forma un processo request-response)

- **Invoke**: invoca una operazione su un partner
- **Assign**: modifica il valore di una variabile
- **Throw**: solleva una eccezione
- **Wait**: il processo è sospeso per una certo intervallo di tempo
- **Empty**: operazione nulla, utile per sincronizzare flussi paralleli di esecuzione

**Strutturate**: sono contenitori per altre attività, gestiscono il flusso di esecuzione

- **Sequence**: costruisce una sequenza di azioni
- **Flow**: consente l'esecuzione di più attività in parallelo
- **Switch**: consente la scelta condizionale tra diverse azioni
- **While**: implementa un ciclo su un insieme di azioni
- **Pick**: blocca un'attività sino all'arrivo di un messaggio

Vediamo ora un esempio completo di documento BPEL. Consideriamo un processo per la concessione di un prestito finanziario. Tale processo sarà attivato dalla ricezione di un messaggio da parte di un potenziale cliente. Nel caso la somma richiesta sia inferiore ad una soglia prefissata (nell'esempio pari a mille euro), il prestito sarà approvato senza particolari controlli sulla solvibilità del cliente. In caso contrario il processo demanderà ad un servizio esterno la stima del livello di rischio. Una volta ricevuta la risposta sarà possibile decidere se approvare o meno la richiesta in base al fattore di rischio stimanto (nell'esempio "high" oppure "low"). Il diagramma di flusso può essere rappresetanto graficamente come in figura 2.7

Ecco di seguito il documento BPEL:

```
<process name="loanApprovalProcess"...>
   <partnerLinks>
      <partnerLink name="customer"
myRole="approver"
partnerLinkType="loanApprovalLinkType"/>
      <partnerLink name="assessor"
partnerRole="assessor"
partnerLinkType="riskAssessmentLinkType"/>
   </partnerLinks>
```

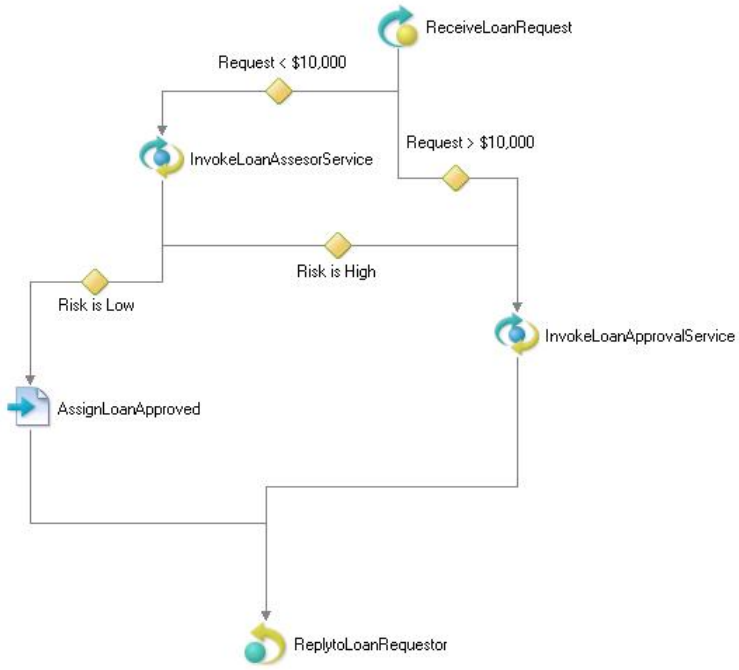



Figura 2.7: Gestione di un prestito

```
   <variables>
      <variable name="request"
messageType="creditInformation"/>
      <variable name="riskAssessment"
messageType="riskAssessmentMessage"/>
      <variable name="approvalInfo"
messageType="approvalMessage"/>
   </variables>

   <flow>
      <links>
         <link name="receive-to-assess"/>
         <link name="assignApproved"/>
 <link name="assignApproved"/>
 <link name="do-reply"/>
      </links>
```

```
      <receive
name="receiveRequest"
partnerLink="customer"
portType="apns:loanApprovalPT"
operation="approve"
variable="request">
         <source linkName="assignApproved"
transitionCondition=
"bpws:getVariableData('request', 'amount')&lt;=1000"/>
         <source linkName="receive-to-assess"
transitionCondition=
"bpws:getVariableData('request', 'amount')&gt;1000"/>
      </receive>

      <invoke
name="invokeAssessor"
partnerLink="assessor"
portType="riskAssessmentPT"
operation="check"
inputVariable="request"
outputVariable="riskAssessment">
         <target linkName="receive-to-assess"/>
         <source linkName="assignApproved"
transitionCondition=
"bpws:getVariableData('riskAssessment','risk')='low'"/>
         <source linkName="assignNotApproved"
transitionCondition=
"bpws:getVariableData('riskAssessment', 'risk')='high'"/>
      </invoke>

  <assign name="assignApproved">
         <target linkName="assignApproved"/>
         <source linkName="do-reply"/>
         <copy>
            <from expression="'approved'"/>
            <to part="accept" variable="approvalInfo"/>
         </copy>
      </assign>
```

```
  <assign name="assignNotApproved">
         <target linkName="assignNotApproved"/>
         <source linkName="do-reply"/>
         <copy>
            <from expression="'sorry, not approved'"/>
            <to part="accept" variable="approvalInfo"/>
         </copy>
      </assign>

      <reply name="reply"
partnerLink="customer"
portType="apns:loanApprovalPT"
operation="approve"
variable="approvalInfo">
        <target linkName="do-reply"/>
      </reply>
   </flow>
</process>
```

# Capitolo 3

# CooWS

*In questo capitolo introduciamo CooWS, un'implementazione del modello Coo-BDI orientato ai Web Service. Vengono analizzate le motivazioni, gli obiettivi e l'architettura generale del sistema.*

## 3.1 Motivazioni

La teoria sugli agenti intelligenti e l'architettura dei web service condividono alcuni concetti fondamentali dai quali trae spunto questo lavoro di tesi. Le affinità vanno ricercate prima di tutto nel significato stesso di "agente intelligente" e nelle proprietà salienti della WSA. Il termine "agente" suggerisce proprietà quali specializzazione e personalizzazione, mentre il termine "intelligente" suggerisce buone capacità decisionali. Come già accennato nel primo capitolo, i software di questo tipo svolgono un ruolo simile a quello di un assistente personale quale un gestore appuntamenti, un agente di viaggio, un gestore prenotazioni oppure, in un contesto aziendale, un gestore delle ordinazioni. In generale un agente intelligente lavora nell'interesse del proprio utente, ovvero conosce le sue esigenze ed effettua buone decisioni nell'ambito di uno specifico settore.
Per quanto concerne i Web service, e più in generale il modello orientato ai servizi, va osservato come nelle specifiche dei differenti standard il concetto di servizio viene modellato come programma passivo in attesa di messaggi esterni. Ma da dove originano questi messaggi? Chi attiva l'interazione? È qui che possono allora subentrare gli agenti intelligenti, nel ruolo di clienti che usufruiscono di servizi web.
A seconda della tipologia di agente le dinamiche di interazione variano e di conseguenza le problematiche e le possibili soluzioni.

### Agenti mobili e Web service

Un agente è detto mobile se durante la propria vita può cambiare ambiente di esecuzione. Un agente mobile può quindi trovarsi in momenti differenti su un computer desktop, su un PDA, su un server aziendale. Un modello ad agenti simile, per offrire una qualche efficacia, deve poter contare su una piattaforma comune a tutti gli ambienti, attraverso la quale poter condividere le risorse. In questo contesto si può allora collocare la WSA, con standard quali WSDL, UDDI e SOAP, fornendo una reale piattaforma distribuita perfettamente adatta a soddisfare tali esigenze. Gli agenti mobili hanno bisogno di un linguaggio comune per descrivere le proprie interfacce: l'impiego di WSDL risulta il più adatto. Ancora, gli agenti mobili hanno bisogno di un meccanismo di registrazione e ricerca di nuovi agenti: UDDI è stato progettato per fornire tali funzionalità. Infine il protocollo di comunicazione deve essere aperto per facilitare l'integrazione con sistemi differenti: SOAP è la soluzione ideale.

## Semantic Web Service

I Semantic Web Service possono essere visti come un modello orientato ai servizi esteso con idee e linguaggi tipici del Web semantico (Semantic Web). Il Semantic Web, per la prima volta concettualizzato da Berners-Lee [30], può essere visto come il complemento del World Wide Web (WWW) sotto forma di informazioni esplicitamente rappresentate per una fruizione automatica. Questo nuova forma di Web è reso possibile grazie alla definizione di un insieme di nuovi standard da parte del World Wide Web Consortium (W3C) [31]. Le nuove tendenze mostrano come il Semantic Web stia guadagnando credito. Le sorgenti di informazione sono sempre più consultabili da agenti software così come da persone umane (la recente esplosione dei feed RSS [32] ne è un esempio).
Così come il Semantic Web è un'estensione del World Wide Web, i Semantic Web service sono quindi un'estensione dei Web service. Ad esempio per quanto riguarda la definizione dell'interfaccia di un servizio, WSDL permette di definire le operazioni, i tipi dei parametri e la codifica dei dati ma non ha alcuna capacità descrittiva per quanto concerne l'aspetto semantico del servizio. Allo stesso modo UDDI fornisce uno standard per la catalogazione e ricerca di servizi ma manca di strumenti di descrizione e ricerca di natura più "semantica".

## Agenti cooperativi e web service

Come osservato più volte, nel campo dei Web Service la ricerca si è focalizzata molto sugli aspetti di interazione, coordinamento e composizione di diversi servizi. Come trattato da M.N. Huhns, 2002 [33] ed in AgentLink III, 2004 [34] molto del lavoro svolto nel campo degli agenti intelligenti può essere sfruttato per risolvere questo tipo di problematiche.
In questo lavoro viene studiata e sviluppata una particolare implementazione basata sul *procedural learning* (appreddimento procedurale). Con questo approccio si vuole realizzare un sistema di componenti (agenti) adattivi in grado di cercare e scambiarsi informazioni utili alla risoluzione di uno specifico insieme di problemi in uno dato ambiente.
A differenza degli approcci basati sulla ricerca di servizi e planning (pianificazione) nel procedural learning il goal non viene soddisfatto né da servizi ad-hoc di terzi né tramite ragionamento (reasoning) esplicito dell'agente. Nel primo caso si richiede infatti l'esistenza di server di elaborazione addizionali, non sempre ammissibili, sopratutto quando sia necessario trasmettere a server intermedi dati "delicati" dell'agente. Nel secondo caso invece si riscontrano penalità sia in termini di computazione, sopratutto in ambienti con risorse limitate quali dispositivi "embedded" (integrati) con necessità di risposta in tempo reale, sia in termini di autonomia (è

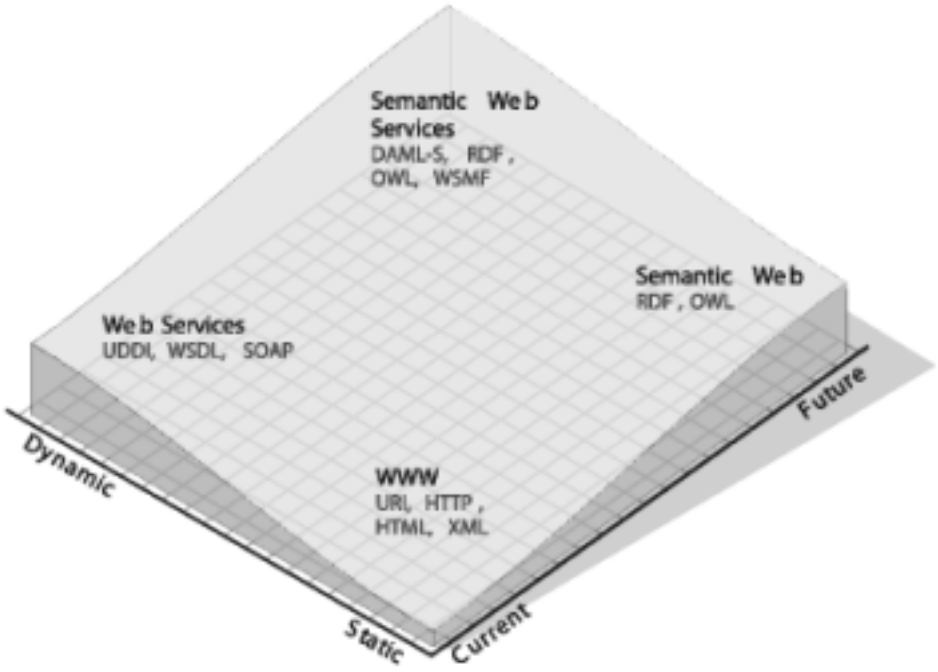



Figura 3.1: Evoluzione del World Wide Web

richiesto l'intervento umano quando in un albero di scelte le alternative proposte risultino equivalenti).
Con il procedural learning non è invece necessario il planning (non si generano nuovi piani) e tantomeno affidarsi a servizi terzi (ogni piano viene eseguito localmente all'agente).

## 3.2 Obiettivi

Alla luce delle considerazioni precedenti, la piattaforma CooWS si pone i seguenti obiettivi:

- **supporto a sistemi MAS**: così come nel modello Coo-BDI, la piattaforma deve supportare una comunità di agenti. È compito della piattaforma implementare le funzionalità necessarie alla comunicazione inter-agente.
- **distribuzione**: gli agenti sono distribuiti sul web. Il sistema CooWS è costituito da una rete di macchine ognuna delle quali ospita una istanza del server CooWS. Ogni istanza ospita a sua volta un insieme di agenti.

Ogni agente identifica una comunicazione con la coppia (agente mittente, agente destinatario) senza curarsi della dislocazione fisica del partner. È compito della piattaforma CooWS trasportare il messaggio dalla macchina sorgente alla macchina destinazione attraverso opportune operazioni di ricerca dell'agente destinatario e codifica del messaggio.

– **aderenza agli standard della WSA**: le operazioni esposte ed i messaggi scambiati sono conformi agli standard della WSA. I messaggi scambiati via rete sono codificati in SOAP, l'interfaccia esposta da ogni nodo appartenente alla rete CooWS, è rappresentata da un Web Service. Il body di un piano è codificato attraverso un processo BPEL. L'esecuzione di un piano corrisponde quindi all'invocazione di un insieme di servizi web.

## 3.3 Storia del sistema CooWS

### L'idea di Paolo Busetta

L'idea dell'implementazione del modello Coo-BDI nel contesto dei Web Service nasce da Paolo Busetta. Inizialmente contattato per una verifica di alcuni dettagli dell'articolo su Coo-BDI [7], ha quindi proposto una collaborazione mirata all'implementazione di un sistema basato sugli standard della WSA e con le stesse idee di Coo-BDI. L'idea fondamentale, dalla quale è poi scatturito questo lavoro, è quella di utilizzare le specifiche di processi BPEL per rappresentare il body dei piani. Vedremo nei paragrafi successivi cosa questo significhi e comporti.

### Il primo prototipo

Dopo un periodo passato sulla documentazione delle differenti specifiche della WSA vede quindi la luce il primo prototipo. Nelle intenzioni di questa prima versione c'era l'utilizzo di BPEL come processo di specifica non solo per i body dei piani ma anche per il motore dell'intero applicativo. Furono pertanto sviluppati i processi BPEL per la gestione della cooperazione (eventi *requested* e *provided*) e l'interazione con le risorse "backend" (database dei piani ed altre informazioni). Ben presto emerse la debolezza di fondo di una simile struttura. La maggiore limitazione con la quale ci scontrammo fu la mancanza del concetto di stato all'interno dei processi BPEL (questi sono infatti specificati come "stateless"). Una tale limitazione era inaccettabile per riuscire a conseguire gli obiettivi proposti. Fu quindi necessario ripensare da capo l'architettura, previa una buona dose di ulteriore documentazione. Nel frattempo videro la luce i primi engine BPEL open

source: ([35], [36], [37]). In particolare fu grazie ad ActiveBPEL, ed al suo codice sorgente, che prese forma il secondo, e attuale, prototipo.

### La versione attuale

La versione attuale dell'applicativo mantiene della prima l'idea originale, ovvero la rappresentazione dei body dei piani come processi BPEL. Per il resto l'architettura è stata completamente riprogettata e sviluppata in Java. L'interfaccia pubblica dell'engine viene esposta attraverso Axis [38] compatibilmente con gli standard della WSA. Per l'esecuzione dei processi BPEL si è scelto di utilizzare l'engine ActiveBPEL [35]. Il catalogo degli agenti poggia su un registro UDDI sul quale viene registrata una scheda per ogni agente della rete CooWS. Nell'attuale implementazione la scheda di ogni agente contiene l'identificativo univoco dell'agente e l'indirizzo della macchina sulla quale è ospitato.

## 3.4 Scelte effettuate

L'architettura definitiva del sistema CooWS è il risultato di una serie di iterazioni nel processo di progettazione e sviluppo. Per raffinamenti successivi i prototipi iniziali hanno preso forma guidati da una serie di scelte e principi che elenchiamo qui di seguito per maggiore chiarezza:

- L'applicazione, sviluppata in Java, espone la propria interfaccia tramite Web Service con un insieme di operazioni.
- Il sistema, una rete di nodi distribuita sul Web, fornisce un ambiente logicamente unificato nel quale vengono installati gli agenti. In termini di visibilità, ogni agente può interagire con qualsiasi altro agente situato su una qualsiasi macchina appartenente alla rete CooWS.
- Ogni nodo della rete CooWS è rappresentato da una istanza del motore CooWS. Ogni nodo può ospitare diversi agenti. Non vi sono dipendenze tra i nodi. Questo significa che in qualsiasi istante un nodo può essere attivo o inattivo senza per questo influenzare il funzionamento degli altri "peer".
- Ogni agente viene registrato su un registro centrale UDDI associando all'identificatore dell'agente l'indirizzo della macchina che lo ospita. Il singolo agente non ha conoscenza sulla dislocazione fisica (locale o remota) degli agenti partner. L'utilizzo minimale del registro UDDI è giustificato dalle potenziali estensioni future dell'applicativo, nella direzione dei Semantic Web Service (vedi **??**).

- L'interazione tra due agenti è identificata dalla coppia degli identificatori dei due agenti coinvolti (mittente, destinatario). Gli agenti astraggono quindi il livello di rete. È compito della piattaforma gestire l'instradamento dei messaggi tra agenti locali e remoti.

- Ogni agente mantiene localmente una coda degli eventi, un registro dei piani ed un insieme di intenzioni gestite secondo la specifica Coo-BDI.

- Il body di un piano è rappresentato da un processo BPEL. Eseguire un piano significa pertanto eseguire il processo BPEL associato al suo body. L'esecuzione dei processi BPEL è demandata ad ActiveBPEL [35], un engine BPEL esterno all'applicazione.

- In base alle precedenti assunzioni non è più possibile conoscere con certezza lo stato di ogni agente sulla rete. È quindi errato supporre che l'invio ad $n$ agenti di un evento *requested* debba sempre generare $n$ eventi *provided* di risposta. Per tale ragione è stato necessario apportare alcune modifiche al protocollo di cooperazione rilassandone opportunamente alcuni vincoli. In particolare, ogni qual volta una nuovo evento *requested* è costuito ed inviato l'intenzione sorgente assegna al record al top dello stack un valore temporale di *timeout* in base al quale sarà possibile stabilire il suo eventuale fallimento. Questa condizione sarà verificata una volta esaurito l'insieme dei piani rilevanti e superato il valore di timeout. Dall'altra parte grazie a questo rilassamento non è più necessario inviare un evento *provided* nel caso l'insieme dei piani forniti risulti vuoto. Questo porta inoltre ad una riduzione del carico di messaggi/eventi scambiati.

Questo elenco, sebbene non dettagliato, fornisce un'utile panoramica sull'intero sistema e la strada seguita durante lo sviluppo. Nei paragrafi successivi ci focalizzeremo via via sui diversi aspetti scendendo nei dettagli di ciascuno, pur mantenendo un analisi orientata alla progettazione.
Sarà invece compito del capitolo successivo fornire uno spaccato più tecnico.
Vediamo ora meglio cosa significhi e comporti implementare i body dei piani tramite specifiche di processi BPEL.

## 3.5 Body dei piani come processi BPEL

L'idea dalla quale è scaturito questo lavoro è il punto centrale tramite il quale si è riusciti ad avvicinare, di più, integrare, il mondo dei Web Service con quello degli agenti intelligenti: rappresentare la conoscenza procedurale del body di un piano attraverso la specifica di un processo BPEL.

Cosa comporta un simile approccio? Se la specifica di un processo BPEL rappresenta un flusso di invocazioni a diversi Web Service, allora rappresentare il body di un piano Coo-BDI con un processo BPEL significa rappresentare la sequenza di "azioni" che compongono il body come una sequenza di "chiamate" a Web Service. Eseguire il body di un piano si traduce allora nell'invocazione di un processo BPEL dalla quale scaturisce una sequenza di invocazioni a Web Service.
Ecco allora che l'integrazione si fa evidente. L'elemento fondamentale di un piano Coo-BDI, la sequenza di azioni del suo corpo, viene sviluppata sul mattone base della WSA, il concetto di Web Service appunto.
Per poter eseguire i piani è quindi necessario appoggiarsi ad un engine BPEL, attraverso il quale installare e successivamente invocare le specifiche dei piani/processi BPEL.
Vediamo qui di seguito cosa comporti tutto ciò.

## Interfaccia con il motore BPEL

É necessario fare qui alcune considerazioni che, seppur di carattere tecnico, hanno fortemente condizionato il design del sistema CooWS.
Una prima considerazione rigurarda il funzionamento del motore BPEL. Indipendentemente dall'implementazione scelta ogni processo può essere installato una sola volta per poi rispondere a differenti invocazioni. Nel caso quindi nel nostro sistema uno stesso piano sia condiviso da diversi agenti, nel momento in cui alcuni ne richiedano l'esecuzione sarà necessario installare una sola volta il processo BPEL sull'engine e quindi invocarlo più volte, una per ogni agente richiedente. Da questo si deduce pertanto la necessità di un meccanismo di sincronizzazione.
Una seconda considerazione è dovuta a problemi di sovraccarico della memoria e della banda di rete conseguenti all'utilizzazione degli archivi dei processi BPEL (file con estensione bpr, nel caso dell'engine ActiveBPEL). Ogni processo BPEL per poter essere installato su un engine BPEL necessita infatti dei documenti WSDL dei servizi partner e talvolta questi ultimi hanno dimensioni dell'ordine di centinaia di KB. Per citare un esempio, il Web Service del portale di aste on line Ebay raggiunge un occupazione superiore al megabyte. Sono quindi evidenti i problemi di spazio che comporta la memorizzazione ridondante dello stesso processo per ogni piano nonché la trasmissione di un insieme di piani via rete (nel caso di una cooperazione che scaturisce tra due agenti ospitati su macchine differenti).
Le problematiche qui sopra esposte hanno trovato soluzione nella centralizzazione dell'archivio dei processi BPEL e della loro gestione attraverso l'introduzione di un modulo dedicato locale ad ogni nodo della rete. Questo modulo si preoccupa di mantenere un archivio dei processi BPEL (associandovi un identificatore univoco — il nome del servizio associato al processo — che sarà quindi utilizzato

come chiave-puntatore all'interno dei singoli piani delle diverse librerie di piani degli agenti) e di mediare con l'engine BPEL per l'installazione ed invocazione dei processi BPEL.
Così facendo anche l'occupazione della banda di rete viene minimizzata. Ogni piano porta con sé infatti l'informazione essenziale (il nome del servizio associato al processo BPEL, una semplice stringa di caratteri) per identificare il processo BPEL associato al proprio corpo. La trasmissione via rete di un insieme di piani (come accade con l'evento *provided*) risulta pertanto sensibilmente alleggerita. È compito del gestore dei processi tenere traccia della dislocazione dei processi associata a piani acquisiti da istanze remote dell'engine CooWS. Se infatti, in seguito ad una richiesta cooperativa, un agente riceve un insieme di piani e decide di istanziarne uno tra questi, può accadere che il processo associato al suo corpo non sia disponibile nella libreria locale dei processi BPEL. In tal caso il gestore dei processi deve essere in grado di recuperare dalla macchina sorgente (dalla quale è stato inviato l'evento *provided*) la specifica BPEL del processo in oggetto. Tratteremo meglio più avanti questo scenario in questo e nel prossimo capitolo.

## Achieve e risultato

Un altro problema sorto dall'adozione delle specifiche BPEL è stata la necessità di mantenere il meccanismo di annidamento delle istanze di piano. Come già osservato infatti, una volta che un processo viene installato ed avviato, non è più possibile mantenere un controllo sul suo flusso esecutivo. Questo perché si è scelto di utilizzare un motore BPEL "a scatola chiusa" per ovvie ragioni di praticità e compatibilità con gli standard della WSA. Di conseguenza le uniche "azioni" richiamabili all'interno dei processi BPEL, come definito nella specifica del linguaggio, sono le operazioni appartenenti ad altri Web Service. Per emulare quindi un'azione come la *achieve* del modello Coo-BDI è stato necessario introdurre un'operazione *dispatchAchieve* nel WS del nostro motore. Argomenti di tale operazione sono l'identificatore dell'agente che ha richiamato il processo corrente, l'identificatore dell'intenzione ed il desiderio oggetto della *achieve*. L'azione *achieve* si è così tramutata in una invocazione di un semplice Web Service. Quando il Web Service riceve una invocazione su tale operazione costruisce un messaggio da inoltrare all'agente per notificarlo della richiesta. Per non forzare l'architettura del motore si è scelto di definire un nuovo tipo di evento ad-hoc in modo tale da poter sfruttare la catena di consegna ed elaborazione già utilizzata per gli eventi cooperativi *requested* e *provided*. L'evento di tipo *achieve* viene creato in seguito alla invocazione dell'operazione suddetta e consegnato al gestore degli agenti locale il quale la inserirà nella coda eventi globale insieme agli altri eventi in attesa di essere consegnati.

Nel frattempo il processo BPEL dal quale è partita la richiesta resta in attesa bloccato da un'azione *receive*. Una soluzione simile alla precedente è stata necessaria per poter fornire agli agenti una notifica sul completamento con successo o fallimento dell'esecuzione di un processo da loro richiesta. Come per la *achieve* è stata introdotta una nuova operazione, *dispatchPlanOutcome*, nel Web Service del motore. Questa volta però, per non introdurre vincoli eccessivi nella struttura dei processi utilizzati, l'invocazione di tale operazione non compare direttamente all'interno del corpo del processo BPEL associato al piano. È stato infatti introdotto un processo BPEL *invocatore* il cui compito è eseguire i processi che via via vengono richiesti introducendo la gestione della notifica di completamento. Il funzionamento di tale processo può essere riassunto con il seguente pseudocodice:

```
invoke_process(agentID, intentionID, serviceName)
  try {
   // qui invochiamo effettivamente il processo
   invoke(serviceName, agentID, intentionID)

   // se il processo e' completato senza fallimenti
   invoke(CooWS.dispathPlanOutcome,agentID, intentionID, TRUE)
  } catch (Exception e) {
    // se si e' verificata una eccezione
    invoke(CooWS.dispathPlanOutcome,agentID, intentionID, FALSE)
  }
```

Questa soluzione permette di mantenere una sintassi più pulita dei processi.

## 3.6 I piani

Nella sezione precedente si è discussa la rappresentazione dei body dei piani ma sino ad ora non è stata affrontata la rappresentazione di un piano nel suo complesso.
Ricordiamo che un piano in Coo-BDI è rappresentato da una tupla costituita da:

- *access specifier*: può assumere valore *PUBLIC*, *PRIVATE*, *only(AgentIdSet)*
- *trigger*: rappresenta il desiderio che il piano è in grado di conseguire
- *precondition*: formula
- *invariante*: formula

– *body*: albero di stati con archi etichettati da azioni, desideri, query.

– *success and failure actions*: insiemi di azioni da eseguire una volta completato il piano.

Come si può osservare questa struttura dati è strettamente legata all'ambiente di esecuzione per il quale è stato progettato Coo-BDI, Prolog appunto. In particolare, concetti quali formula ed invariante sono impiegati al meglio nella programmazione logica dove troviamo un ambiente di variabili nel quale è possibile effettuare operazioni di *valutazione* ed *unificazione*. Questi concetti diventano di difficile implementazione quando ci si sposta in un contesto come il Web, con un linguaggio come Java e sopratutto con un rilassamento dei vincoli introdotto. Inoltre, scegliendo come body dei piani specifiche di processi BPEL, si rende di fatto irrealizzabile l'applicazione dell'invariante durante la loro esecuzione: una volta richiesta l'esecuzione di un processo BPEL ad un engine esterno se ne perde di fatto il controllo.
I problemi sopra esposti ci hanno portato quindi ad optare per l'abolizione della precondizione e dell'invariante. Tale semplificazione corrisponde ad avere tutti i piani con precondizione ed invariante sempre verificate. Un discorso analogo riguarda il trigger, anch'esso rappresentato in Coo-BDI attraverso una formula. Qui l'abolizione non è così indolore come per invariante e precondizione. Il trigger di un piano è infatti essenziale durante il recupero dell'insieme dei piani rilevanti, nel quale si effettua l'operazione di *match* tra desire e trigger del piano. Per sopperire alla mancanza di formule è stato quindi necessario cambiare la rappresentazione dei desire in modo da mantenere l'operazione di match. In questo primo prototipo di CooWS, data la considerevole mole di lavoro, si è scelta una soluzione temporanea dove i trigger sono stringhe di caratteri e l'operazioni di match si risolve nell'operazione di confronto, nell'ottica di una prossima versione dove l'impiego di ontologie e strumenti/linguaggi di descrizione semantica introdurrano una migliore capacità espressiva. Per ulteriori considerazioni rimandiamo qui alla sezione sviluppi futuri del capitolo conclusivo 5.2.

## 3.7 L'agente in CooWS

Alla luce di precedenti considerazioni, cosa rappresenta un agente in CooWS? Quali funzionalità supporta? Come interagisce con gli altri agenti? A queste domande daremo risposta in questo paragrafo.
In linea di massima la specifica di un agente di CooWS ricalca quella data nel modello Coo-BDI introducendo, laddove necessario, opportune semplificazioni o

migliorie.
Così come in Coo-BDI, ogni agente in CooWS ha una propria coda degli eventi, una libreria di piani, un insieme di intenzioni, un insieme di agenti partner, una strategia di acquisizione ed una strategia di cooperazione.
Analizziamo tutte queste componenti qui di seguito, soffermandoci su nuovi aspetti introdotti dalla nostra implementazione.

- **La coda degli eventi**: così come in Coo-BDI, rappresenta la struttura dati che si occupa di accogliere gli eventi esterni in attesa di essere elaborati. La gestione della coda segue una politica FIFO.
  Una delle peculiarità dell'implementazione Java, sfruttata in parti differenti di questo lavoro, rispetto al modello in Prolog, è la gestione concorrente di differenti *task* di elaborazione. In questo caso, per quanto concerne la gestione della coda degli eventi, abbiamo un insieme limitato di thread per ogni agente con i quali vengono gestiti in parallelo i nuovi eventi in ingresso. Ogni volta che un nuovo evento entra nella coda si controlla se è disponibile un thread. In caso affermativo il thread si occuperà di gestire opportunamente l'evento richiamando una delle procedure locali all'agente. In caso negativo l'evento resterà in coda in attesa del primo thread libero. Per una trattazione più dettagliata sui vari tipi di eventi rimandiamo al paragrafo successivo. Per ulteriori dettagli tecnici rimandiamo invece al capitolo successivo.

- **La libreria dei piani**: ogni agente dispone localmente di un insieme di piani che costituiscono la propria libreria. Su questo insieme di piani è possibile effettuare operazioni di inserimento (aggiornamento della libreria in base alla strategia di acquisizione) e ricerca (recupero di piani rilevanti per un dato desiderio). La libreria è privata: solo l'agente proprietario ha diritto di accedere ad essa.

- **Le intenzioni**: in CooWS riprendono la struttura ed il comportamento da Coo-BDI estendendolo con alcuni accorgimenti orientati ad una implementazione reale in un contesto Web Service. Ogni intenzione ha associato uno stack di esecuzione più altre informazioni. In questo stack troviamo appilate le istanze di piano. Una istanza di piano è rappresentata da un desiderio, un insieme di piani rilevanti, il piano attualmente in esecuzione, un insieme di piani rilevanti falliti ed il valore di timeout (un valore pari a venti secondi è più che sufficiente per la maggior parte delle attuali reti). Tutte queste caratteristiche sono state ereditate dal modello Coo-BDI. La novità principale introdotta in CooWS è la gestione concorrente multithread delle

intenzioni. Mentre in Coo-BDI le intenzioni vengono schedulate singolarmente in CooWS è possibile avere contemporaneamente diverse intenzioni in esecuzione allo stesso tempo.

- **Agenti partner**: rappresentano i collaboratori dell'agente. Ogni qual volta la strategia di cooperazione lo richiede l'agente invia ad ogni partner una richiesta di piani rilevanti per un determinato desiderio. Tale lista nel modello Coo-BDI, così come in questa implementazione, è nota staticamente per ogni agente e non è presente alcun meccanismo per l'aggiornamento dinamico. La gestione dinamica di questa lista potrà essere una futura estensione del motore CooWS seguendo i meccanismi di fiducia e reputazione illustrati in [39]

- **Strategia di cooperazione**: il processo di cooperazione è strettamente legato alla strategia di cooperazione di ogni singolo agente. In CooWS la strategia, così come in Coo-BDI (vedi sezione 1.2), è determinata dall'insieme degli agenti partner, dalla politica di recupero dei piani e dalla politica di acquisizione dei piani.
  La politica di recupero dei piani può assumere uno dei due valori: *always*, ed in tal caso l'agente richiede la cooperazione degli agenti partner ogni qual volta sia necessario recuperare un insieme di piani rilevanti, oppure *no local*, ed in tal caso l'agente ricorre agli agenti partner solo quando l'insieme dei piani locali rilevanti è vuoto.
  La politica di acquisizione dei piani determina l'aggiornamento della libreria dei piani con l'insieme dei piani recuperati attraverso il processo di cooperazione. Essa può assumere uno tra i tre valori: *add* (aggiungi alla libreria), *discard* (non aggiungere alla libreria), *replace* (sostituisci i piani della libreria con quelli recuperati aventi lo stesso trigger).

## 3.8 Gli eventi

La famiglia degli eventi rappresenta un'estensione di quella utilizzata nel modello Coo-BDI:

**requested**: come in Coo-BDI l'evento requested è utilizzato per attivare il processo di cooperazione. Quando un agente deve recuperare un insieme di piani rilevanti per un dato desiderio può richiedere aiuto agli agenti partner inviando loro un evento di tipo requested contenente il proprio identificativo, il desire e l'identificativo della richiesta formato dalla coppia (identificativo dell'intenzione, identificativo del record dello stack).

**provided**: costruito in risposta ad un evento *requested* contenente un desiderio per il quale è stato possibile recuperare un insieme non vuoto di piani rilevanti. In tal caso viene costruito un evento *provided* contenente l'insieme trovato e l'identificativo della richiesta.

**ordinary**: in questa categoria facciamo ricadere tutti gli eventi provenienti dall'esterno. Qui troviamo alcuni cambiamenti rispetto al modello originario, legati alla semplificazione introdotta nella rappresentazione dei desideri. Mentre nel modello Coo-BDI alla ricezione di un evento di tipo *ordinary* segue una fase nella quale si genera l'insieme di desideri principali, nel modello CooWS questa non compare più in favore dell'introduzione di una lista di desideri all'interno dell'evento stesso.

**achieve**: come illustrato a *pag. 57* questo evento è stato introdotto per mantenere il meccanismo di annidamento delle intenzioni. Scaturisce dall'invocazione dell'operazione *dispatchAchieve* del WS dell'engine da parte di un processo BPEL in esecuzione. L'evento contiene l'identificativo dell'agente e dell'intenzione alla quale appartiene il piano che lo ha originato più il desire argomento della *achieve*.

**plan outcome**: discorso analogo per l'evento *plan outcome* (risultato del piano), introdotto per fornire un responso (positivo o negativo) sul completamento di un processo BPEL. Esso viene generato al termine dell'esecuzione di ogni processo e contiene l'identificativo dell'agente e dell'intenzione alla quale appartiene più un valore booleano che segnala il successo (valore `true`) o il fallimento (valore `false`).

**remote**: introdotto per le comunicazioni via rete, questo evento rappresenta un contenitore per uno qualsiasi degli eventi precedenti, ai quali aggiunge l'informazione sulla macchina di provenienza dell'evento. Viene costruito dal gestore del catalogo degli agenti quando è necessario inviare un evento ad un agente ospitato su un server remoto.

## 3.9 Architettura del sistema CooWS

Illustriamo ora in questa sezione la struttura della singola istanza di un nodo appartenente alla rete CooWS.
Nel processo di analisi descritto nella sezione 3.4 si sono definiti i moduli principali dell'architettura di CooWS: una interfaccia verso l'esterno fornita tramite un Web Service, un modulo gestore degli agenti locali, un modulo di interfaccia con il catalogo degli agenti, un modulo gestore dei processi BPEL. Tutti insieme, questi

moduli concorrono a formare un framework compatibile con le specifiche richieste. Il loro scopo principale è il supporto di una comunità distribuita astraendo la dislocazione fisica dei suoi componenti e l'implementazione scelta per il body dei piani.

- **Il Web Service**: espone una interfaccia con le operazioni necessarie allo scambio di messaggi tra i nodi della rete. Tra queste troviamo un'operazione per lo scambio di messaggi tra gli agenti , nonché un insieme di operazioni di servizio dettate dalle particolari scelte di struttura che vedremo più nel dettaglio nel prossimo capitolo.
- **Il gestore degli agenti**: svolge la funzione di contenitore per l'insieme degli agenti locali ad un nodo, gestendo la consegna degli eventi in ingresso tramite una coda di sistema. Questa coda viene gestita attraverso un pool di thread in modo da garantire buoni risultati in termini di prestazioni ed allo stesso tempo limitare il consumo delle risorse.
  Durante l'avvio (arresto) si preoccupa di ripristinare (salvare) lo stato di ogni agente attraverso il database.
- **Il gestore del catalogo degli agenti**: gestisce l'interfaccia con il registro centrale UDDI fornendo le primitive necessarie per l'invio dei messaggi in uscita. Durante la fase di avvio (arresto) si preoccupa di registrare (cancellare) l'insieme degli agenti locali. In tal modo sul registro centrale è sempre presente la dislocazione fisica di tutti gli agenti attivi.
- **Il gestore dei processi BPEL**: già introdotto nelle pagine precedenti, si preoccupa di centralizzare la gestione dei processi BPEL, fornendo le operazioni per l'installazione, l'esecuzione e scambio (tra differenti nodi della rete) dei processi.

## 3.10 CooWS all'opera

Analizziamo ora il comportamento di CooWS passando in rassegna il ciclo di vita di una singola istanza del motore per poi scendere nei dettagli dei differenti scenari che si possono riproporre a tempo di esecuzione.

### 3.10.1 Uno sguardo dall'esterno

Considerando la singola istanza del motore CooWS, a scatola chiusa si possono osservare due tipologie di interazioni con l'ambiente esterno, come illustrato nel diagramma dei casi d'uso di Figura 3.3.

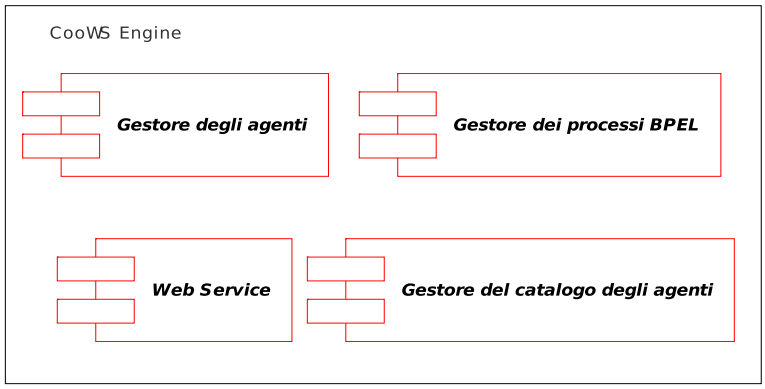



Figura 3.2: I principali moduli di CooWS

Il primo caso d'uso identifica il processo cooperativo costituito dagli eventi *requested* e *provided* scambiati tra due agenti appartenenti alla rete CooWS e ospitati su due macchine distinte della rete CooWS.
Nel secondo caso d'uso l'interazione è attivata da un cliente esterno alla rete CooWS, tipicamente un altro agente specializzato.
Vi è infine una terza classe di interazioni, quella che comprende le operazioni *achieve* e *plan outcome* introdotte in precedenza, ma per la loro stessa natura di espedienti tecnici abbiamo preferito escludere dal diagramma.

### 3.10.2 Ciclo di vita di CooWS

Il ciclo di vita di un'istanza del motore CooWS può essere suddiviso in tre fasi principali: avvio, esecuzione ed arresto.

- **Avvio**: questa fase è caratterizzata dall'inizializzazione degli agenti, dalla registrazione presso il server UDDI, e dall'avvio della fase di esecuzione. L'inizializzazione degli agenti prevede il ripristino, tramite database, di uno stato precedentemente congelato, la creazione delle code eventi, la registrazione degli agenti presso un server centrale UDDI ed infine l'attivazione degli agenti.

- **Esecuzione**: la fase di esecuzione è caratterizzata da un comportamento guidato dagli eventi (event-driven). La ricezione di nuovi messaggi attiva una procedura di consegna per l'aggiornamento della coda eventi dell'agente destinatario. A questo punto è compito dell'agente gestire in maniera opportuna l'evento secondo la specifica data dal modello Coo-BDI. L'invio dei messaggi in uscita è demandata al gestore del catalogo degli agenti, il quale

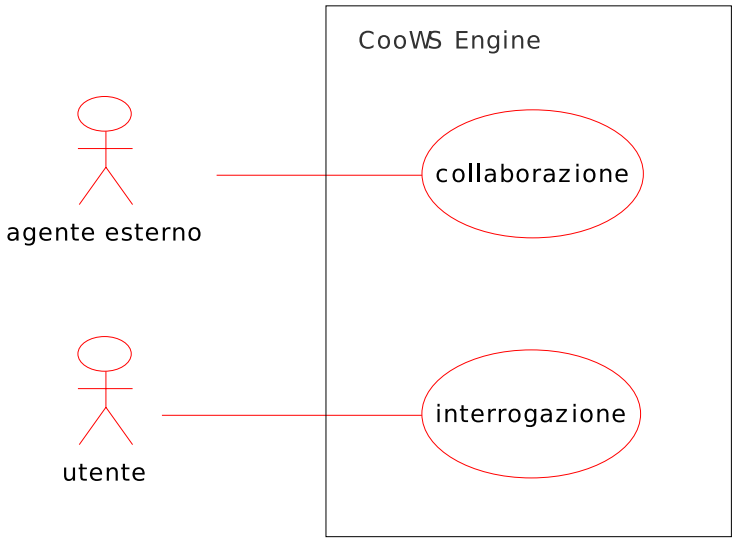



Figura 3.3: Casi d'uso di CooWS

si preoccupa di smistare i messaggi tra gli agenti locali e remoti. Se l'agente destinatario non è reperibile localmente si effettua una ricerca sul registro centrale e, in caso di successo, si costruisce ed invia un messaggio SOAP via rete.

- **Arresto**: l'arresto del sistema deve preoccuparsi di salvare in maniera persistente, mediante un database, lo stato corrente del motore. Quindi è necessario cancellare dal registro UDDI la lista degli agenti locali (in quanto non più accessibili una volta arrestata l'istanza locale del motore CooWS).

### 3.10.3 I differenti scenari

Passiamo ora in rassegna gli scenari che si presentano alla ricezione dei vari eventi. Per ogni evento consideriamo le parti coinvolte, il comportamento previsto in caso di normale esecuzione e i casi di eccezione limitando l'attenzione prima a quanto accade a livello di framework focalizzandoci solo successivamente sul singolo agente.
Nuovi eventi possono provenire dall'esterno o da componenti locali alla singola istanza del motore CooWS. Nel primo caso rientrano tutti gli eventi ricevuti tramite Web Service e provenienti da agenti remoti (evento *remote*) oppure da istanze di processi BPEL attualmente in esecuzione (*achieve* e *planOutcome*). Nel secondo caso troviamo invece tutti gli eventi generati da agenti locali e destinati ad agenti locali, catturati questi dal catalogo degli agenti.

La principale differenza tra gli eventi/messaggi ricevuti tramite Web Service e gli altri è la fase di "filtraggio" che i primi attraversano. A seconda del messaggio/evento ricevuto è necessario costruire o modificare un evento e quindi inoltrarlo al gestore degli agenti locale. Ad esempio nel caso di una invocazione dell'operazione *achieve* è necessario costruire un evento di tipo *EventAchieve* contenente tutte le informazioni ricevute (identificatore dell'agente, identificatore dell'intenzione e rappresentazione del desire) e quindi inoltrarlo al gestore degli agenti locale. Discorso analogo per quanto concerne la gestione dell'operazione *planOutcome*. Nel caso di ricezione di un evento di tipo *remote* o di tipo *ordinary* non si crea alcun nuovo evento inoltrando invece quello appena ricevuto direttamente al gestore degli agenti.
Quando un evento viene invece scambiato tra due agenti ospitati dal medesimo motore non è più necessario passare per il livello Web Service e quindi il gestore del catalogo degli agenti (responsabile per i messaggi in uscita) inoltra direttamente l'evento al gestore degli agenti locale.
Illustriamo ora i differenti scenari aiutandoci con dei diagrammi di sequenza UML.

### Eventi ricevuti tramite Web Service

- **Eventi *remote* e *ordinary***

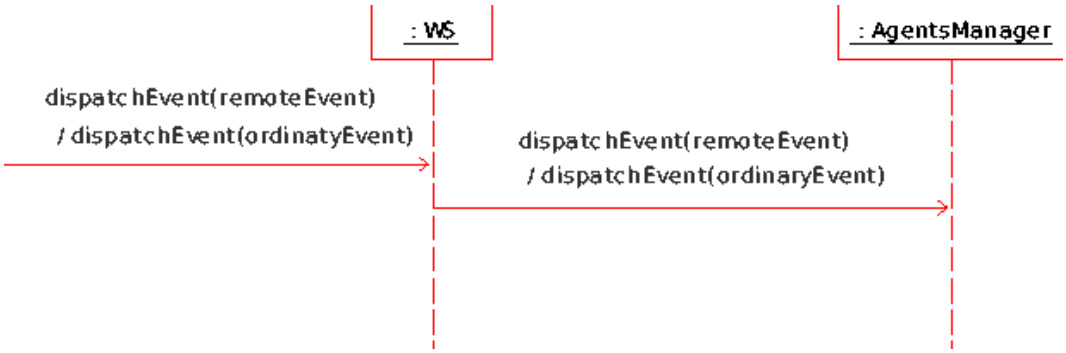



Figura 3.4: Ricezione eventi di tipo *remote* ed *ordinary*

Gli eventi *remote* e *ordinary* non necessitano di alcuna elaborazione da parte del Web Service e vengono quindi inoltrati al gestore degli agenti che si preoccuperà di inserirli nella coda degli eventi di sistema. In particolare nel caso di un evento di tipo *remote* sarà compito del gestore degli agenti estrarre l'evento incapsulato (a seconda dei casi potrà trovare un evento di tipo *requested* oppure di tipo *provided*) per poi inserirlo in coda.

- **Messaggi *achieve* e *planOutcome***

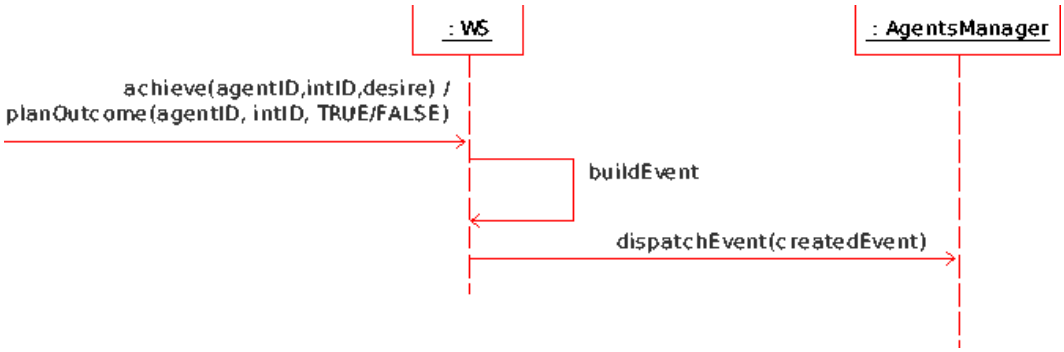



Figura 3.5: Invocazione operazioni *achieve* e *planOutcome*

La gestione delle operazioni *achieve* e *planOutcome* prevede la creazione dei corrispettivi eventi a partire dai parametri ricevuti e quindi l'inoltro al gestore degli agenti locale il quale si preoccuperà di inserirli nella coda di sistema.

### Eventi locali

Gli unici eventi scambiabili direttamente tra due agenti sono quelli di natura cooperativa, ovvero l'evento *requested* e il corrispettivo evento di risposta *provided*. In particolare quando l'agente mittente e l'agente destinatario sono ospitati sulla stessa macchina, l'evento effettua un percorso più breve dei casi precedenti passando dal gestore del catalogo degli agenti direttamente al gestore degli agenti.

### Scenari alternativi

In tutti i casi qui esposti è poi necessario considerare alcuni potenziali casi di errore. Tra questi sicuramente:

- agente destinatario sconosciuto: l'identificativo dell'agente destinatario non corrisponde a nessun agente ospitato localmente. Si richiede la consegna dell'evento al gestore del catalogo degli agenti il quale provedderà a ricercare l'agente destinatario sul registro centrale e, in caso di successo, inoltrerà il messaggio all'host corretto.
- evento sconosciuto: l'agente riceve un messaggio di tipo sconosciuto. In tal caso si segnala nel log l'anomalia e si prosegue l'esecuzione scartando l'evento.

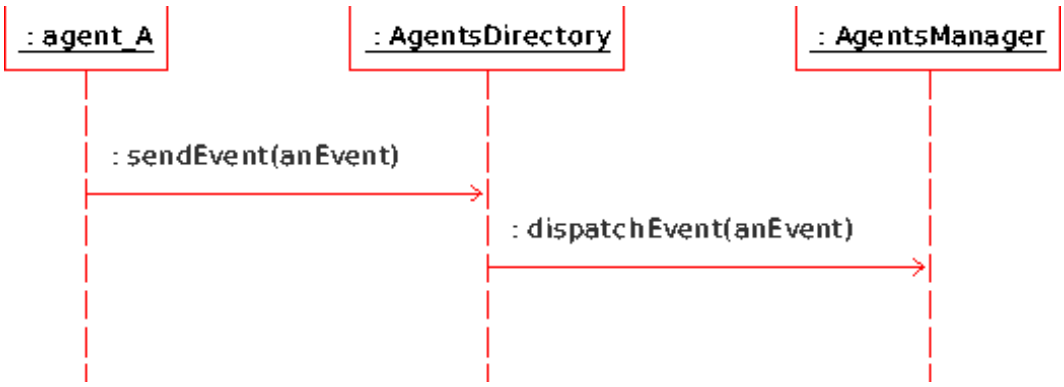



Figura 3.6: Comunicazione tra agenti locali

In tutti gli scenari sinora osservati la consegna di un evento si è fermata al gestore degli agenti. Questo per evidenziare il fatto che la consegna all'agente destinatario non è immediata ma subordinata alla disponibilità/occupazione della coda di sistema. A seconda del momento può accadere di trovare la coda di sistema libera, ed in tal caso l'evento sarà immediatamente inoltrato alla coda locale dell'agente destinatario, oppure occupata e con un numero di thread attivi pari al massimo. In quest'ultimo caso la consegna dell'evento è ritardata nel tempo in attesa che un thread attivo venga liberato.

### La gestione da parte dell'agente

Se consideriamo adesso quanto accade a livello di singolo agente, ai fini della nostra analisi è ininfluente il comportamento qui sopra illustrato. Negli scenari che osserveremo ora partiamo quindi sempre dalla situazione "nuovo evento in ingresso" e vediamo come esso viene gestito da parte dell'agente.
Prima di scendere nei dettagli dei diversi tipi di evento va puntualizzato come il funzionamento della coda eventi dell'agente è segue quello della coda di sistema. Abbiamo infatti una coda gestita con politica FIFO ed un insieme limitato di thread per l'esecuzione concorrente delle primitive di gestione degli eventi. Può quindi accadere che un evento appena consegnato ad un agente venga subito elaborato oppure venga messo in coda di attesa a seconda della disponibilità di risorse del momento.

- **evento *ordinary***: gli eventi ordinari corrispondono ad una particolare interrogazione da parte di un agente esterno alla rete CooWS. Quando un

agente riceve un evento ordinario deve creare una nuova intenzione a partire dal desiderio specificato nell'evento. La nuova intenzione ha associato uno stack contenente come primo elemento un record con il desire ricevuto, l'insieme dei piani locali rilevati per esso, e il tempo di timeout (vedi pag. 55). A seconda della politica di cooperazione viene inviato un evento requested agli agenti partner contenente il desiderio, l'identificatore dell'agente e l'identificatore della richiesta. La nuova intenzione viene aggiunta all'insieme delle intenzioni dell'agente e trattata secondo la specifica del modello Coo-BDI.

- **evento *requested***: la gestione dell'evento requested viene gestito anch'esso secondo le specifiche del modello Coo-BDI. A partire dal desiderio e dall'identificatore dell' agente mittente in esso contenuti si seleziona un insieme di piani rilevanti. Nel caso l'insieme non sia vuoto viene creato un evento *provided* contenente l'insieme e l'identificatore della richiesta (contenuto nell'evento requested) e se ne demanda l'invio al gestore del catalogo degli agenti.

- **evento *provided***: alla ricezione di un insieme di piani rilevanti deve corrispondere una prima ricerca nell'insieme delle intenzioni in base all'identificatore della richiesta. Se la ricerca ha successo si aggiorna lo stack dell'intenzione trovata con l'insieme fornito.

- **evento *achieve***: se un agente riceve un evento di tipo *achieve* allora deve avere sicuramente una intenzione attiva con un piano attualmente in esecuzione. È allora necessario recuperare l'intenzione in oggetto e attivare su di essa il meccanismo di annidamento creando un nuovo record sullo stack a partire dal desiderio specificato dall'achieve.

- **evento *planOutcome***: la ricezione di un evento di tipo *planOutcome* segnala la terminazione (con successo o fallimento in base alle informazioni contenute) di un processo BPEL associato ad un piano in esecuzione su una particolare intenzione. È quindi necessario aggiornare lo stack dell'intenzione in oggetto. Se il piano è fallito sarà necessario selezionare un nuovo piano rilevantte (se disponibile) e richiederne l'esecuzione. Nel caso l'insieme dei piani rilevanti sia vuoto e sia stato raggiunto il tempo di timeout è necessario fare fallire il record sullo stack e attivare il meccanismo di backtracking (se il record è il primo dello stack l'intenzione fallirà, altrimenti il record precedente riceverà la notifica di fallimento). Se il piano è terminato con successo è necessario eliminare il record dallo stack e, a seconda dello stato risultante dello stack, notificare l'ultimo record dello stack (se non

vuoto) oppure rimuovere l'intenzione dopo aver eseguito le *success actions* associate (se lo stack è vuoto).

# Capitolo 4

# Implementazione

*In questo capitolo viene presentata l'implementazione del sistema CooWS così come è stato presentato nel precedente capitolo. Vengono inoltre trattate le problematiche tecniche incontrate e le soluzioni adottate.*

## 4.1 Introduzione

## 4.2 Linguaggi e strumenti utilizzati

### Java

La scelta del linguaggio di programmazione da utilizzare è stata a senso unico. Sebbene sia J2EE [13] che .NET [40] rappresentino le piattaforme di riferimento per lo sviluppo di Web Service, buona parte della documentazione e degli strumenti basati su J2EE risultano di più facile accesso grazie ai più disparati progetti open source (vedi gli strumenti qui di seguito utilizzati).

### Apache Tomcat

Apache Tomcat [41] è il servlet container sviluppato dalla Apache Software Foundation [42] divenuto in questi ultimi anni il server "de facto" per quanto concerne lo sviluppo Web su piattaforma Java. In questo lavoro è utilizzato come container per CooWS ed altri strumenti/librerie.

### ActiveBPEL

ActiveBPEL $^{TM}$[35] è il motore BPEL (compatibile con lo standard BPEL 1.1) sviluppato dalla Active Endpoints Inc. rilasciato sotto licenza GPL. Per poter installare processi BPEL su questo motore è necessario impacchettare i documenti WSDL e BPEL in opportuni file `.bpr` e, quando necessario, copiarli in una apposita cartella `bpr/`.

### jUDDI

jUDDI [43] è il server UDDI (compatibile con lo standard UDDI 2.0) sviluppato dalla Apache Software Foundation. Installato su un servlet container, necessita di un database sul quale archiviare i *publisher* autorizzati ed servizi registrati. Con jUDDI è stato possibile mantenere il catalogo centrale degli agenti dove si tiene traccia della collocazione fisica di ogni agente della rete CooWS. I publisher autorizzati sono tutte le istanze del motore CooWS. Ognuna di esse dispone di un identificatore ed una parola chiave di accesso (specificati nel file di configurazione).

### MySQL

MySQL [44] è uno dei database relazionali open source di maggior successo. Viene utilizzato in questa applicazione come supporto al livello di archiviazione dati delle singole istanze dei nodi della rete CooWS nonché, come accennato sopra, come supporto per il catalogo degli agenti.

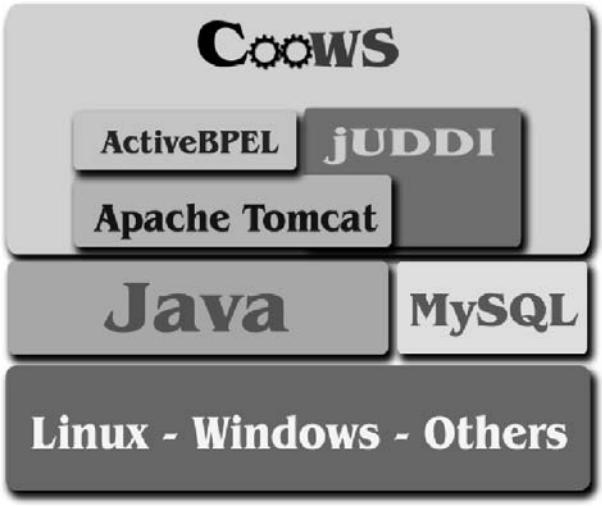



Figura 4.1: Lo stack delle applicazioni di CooWS

## 4.3 Architettura di CooWS

In questa sezione analizziamo la struttura statica del sistema CooWS. Partendo dall'architettura generale vengono quindi descritti i dettagli e le scelte implementative incontrate durante lo sviluppo.
Come introdotto nel capitolo precedente il motore CooWS è composto da quattro componenti:

- **Il Web Service**: l'interfaccia Web Service viene creata ed installata tramite Axis esponendo i metodi della classe dedicata *CooWS* del package *org.coows.ws*. In particolare è stato definito il documento di intallazione Web Service Desployment Descriptor (WSDD) nel quale si forniscono gli estremi necessari per la creazione automatica del Web Service. Una volta avviato il server tomcat è possibile recuperare il documento WSDL all'indirizzo `http://localhost:8080/active-bpel/services`.

- **Il gestore degli agenti**: il gestore degli agenti è implementato dalla classe `CooAgentsManager` del package `org.coows.engine` la quale fornisce le primitive necessarie alla consegna dei messaggi in ingresso (`dispatchEvent`) e all'avvio/arresto del processo di esecuzione degli agenti locali. La consegna dei messaggi viene svolta attraverso una coda di sistema alla quale accedono in modalità concorrente un insieme prestabilito di thread dedicati. In questo modo lo scheduling della coda viene accelerato e garantisce una buona robustezza nell'eventualità di un sovraccarico di risorse (il numero di thread è limitato pertanto non occupa oltre una certa quantità di memoria di sistema).

- **Il gestore del catalogo degli agenti**: il gestore del catalogo degli agenti è implementato dalla classe `CooAgentsDirectory` del package `org.coows.engine` la quale fornisce le primitive necessarie per la registrazione dell'insieme degli agenti locali sul registro centrale UDDI e l'invio dei messaggi in uscita. La comunicazione con il registro UDDI è realizzata attraverso la libreria UDDI4J [45].

- **Il gestore dei processi BPEL**: il gestore dei processi BPEL è realizzato dalla classe `CooBPELProcessesManager` del package `org.coows.engine`. La funzione principale di questo modulo è l'interfaccia del motore CooWS con il motore ActiveBPEL per l'installazione e l'esecuzione dei processi BPEL associati ai body dei piani. La gestione dell'archivio dei processi BPEL è demandata a questo modulo il quale si occupa di mantenere una copia aggiornata dell'insieme dei processi associati all'insieme dei piani locali in una cartella di sistema.

### 4.3.1 Struttura three-tier

Riprendiamo qui la struttura introdotta nel capitolo precendente e introduciamo ulteriori dettagli. La soluzione che meglio si presta alla realizzazione di un'architettura come quella di CooWS è la *three-tier* (a tre livelli) che suddivide il sistema nei tre livelli seguenti:

- **Livello presentazione**
  - **I Web service**: Uno dei requisiti essenziali di CooWS, il più ovvio, è fornire una piattaforma compatibile con gli standard della WSA. Per raggiungere tale obiettivo l'interfaccia esposta dal motore CooWS è stata pubblicata tramite Axis [38], la piattaforma di sviluppo per Web Service della Apache Software Foundation. Con Axis è stato

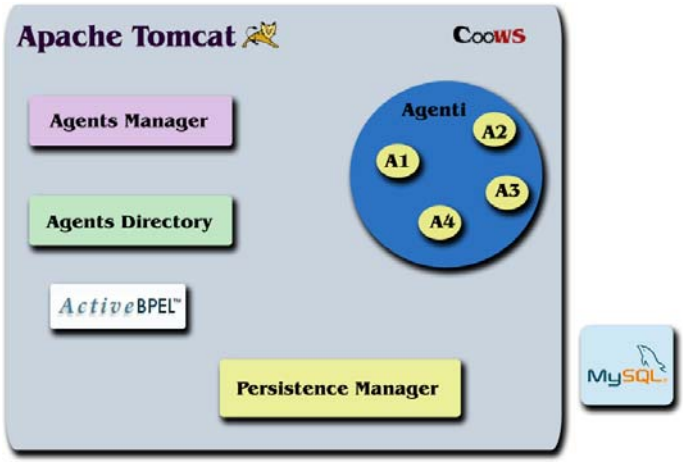



Figura 4.2: L'architettura di CooWS

facile esporre le interfacce esterne nei termini previsti dagli standard della WSA. Le operazioni esposte sono state raggruppate nella classe `CooWS` del package `org.coows.ws`. Queste operazioni rappresentano delle callback ai moduli implementati nel livello business.

- **La servlet**: Per poter installare l'engine CooWS su Tomcat, è stata definita una servlet minimale che, in fase di inizializzazione, si occupa di richiamare il metodo `init` della classe `CooEngine`. Analogamente in fase di arresto. Questa soluzione consentirà una facile estensione futura per l'introduzione di un pannello di controllo (ad esempio attraverso la tecnologia JSP [46]) aspetto in questa attuale propotipo completamente assente.

– **Livello business**: Il livello business costituisce il cuore dell'applicazione CooWS. Qui troviamo i quattro componenti presentati nella sezione precedente più ulteriori classi di supporto.

– **Livello backend**: Il backend della piattaforma è rappresentato dal database MySQL e dall'archivio dei file bpr. Il database viene utilizzato nelle fasi di avvio e arresto del server (vedi pag. 83).

### 4.3.2 I moduli principali

Passiamo ora in rassegna le principali classi del sistema CooWS, evidenziando le strutture dati e le funzionalità supportate.

### CooEngine

La classe CooEngine è il modulo centrale di tutto il motore CooWS. Fornisce le primitive per l'avvio e l'arresto del server CooWS e mantiene i riferimenti a tutte le istanze dei moduli attivi localmente.
Strutture dati fondamentali:

```
/* Rappresentazione in memoria del file XML di configurazione */
private static CooEngineConfiguration mConfig;

/* Gestore degli agenti */
private static CooAgentsManager mAgentsManager;

/* Gestore del catalogo degli agenti */
private static CooAgentsDirectory mAgentsDirectory;

/* Gestore dei processi BPEL */
private static CooBpelProcessesManager mProcessesManager;
```

Metodi publicati:

```
/* avvia il server */
public static void init()

/* arresta il server */
public static void shutdown()

/* getters */
public static CooBpelProcessesManager getMProcessesManager()
public static CooAgentsDirectory getAgentsDirectory()
public static CooAgentsManager getMAgentsManager()
public static CooEngineConfiguration getMConfig()
```

Notiamo in questo stralcio di codice la presenza di un modulo sinora sconosciuto, la rappresentazione in memoria del motore CooWS implementata da un'istanza della classe `CooEngineConfiguration`. Tale classe altro non è che una interfaccia di consultazione per un documento DOM [47] creato a partire dal contenuto di un file XML contenente i parametri di configurazione del motore. Con questo meccanismo è possibile configurare ed instanziare agevolmente differenti nodi della rete CooWS. Più avanti affronteremo nel dettaglio il contenuto di questo documento.

### Gestore degli agenti (CooAgentsManager)

Il gestore degli agenti si occupa di ripristinare in fase di boot l'insieme degli agenti, di gestire la coda degli eventi globale attraverso la quale consegnare i messaggi agli agenti locali. Il ripristino degli agenti si appoggia al gestore della persistenza (`CooPersistenceManager`) il quale a sua volta si interfaccia con il database MySQL. La coda degli eventi globale viene implementata dalla classe `CooGlobalEventsQueue` la quale attraverso una lista di eventi (`LinkedList`) e l'ausilio del framework Commons Pool [48] della Apache Software Foundation gestisce in parallelo la ricezione e consegna di più messaggi.
Strutture dati fondamentali:

```
/* insieme degli agenti locali */
private Hashtable<String,CooAgent> mAgents;

/* coda eventi globale */
private CooGlobalEventsQueue mGlobalEventsQueue;
```

Metodi publicati:

```
/* attiva il ciclo di esecuzione degli agenti locali */
public void startAgents()

/* consegna di un evento */
public void dispatchEvent(ICooEvent anEvent)

/* consegna di un evento remoto */
public void dispatchRemoteEvent(CooEventRemote anEvent)

/* arresta il ciclo di esecuzione degli agenti locali */
public void shutdown(CooEngineConfiguration aConf)
```

### Gestore del registro degli agenti (CooAgentsDirectory)

Il gestore del registro degli agenti si occupa di smistare i messaggi in uscita tra i diversi agenti locali e remoti. Appoggiandosi al registro centrale UDDI effettua la registrazione degli agenti locali e la ricerca di agenti remoti. Ogni agente viene registrato sul registro UDDI con un corrispondente indirizzo IP. In questo modo tutti i nodi appartenenti alla rete sono in grado di risalire alla posizione attuale di un particolare agente.

Strutture dati fondamentali:

```
/* Cache parziale del contenuto del registro UDDI */
private static Hashtable<String,String> mUDDIRepositoryCache;

/* Proxy (stub) per la connessione al registro UDDI */
private UDDIProxy mUDDIProxy;

/* Token di autenticazione necessario per validare le comunicazioni*/
private AuthToken mAuthToken;
```

Metodi pubblicati:

```
/* Publica sul server UDDI la lista degli agenti locali */
public void publishLocalAgents(Set<String> agents)

/* Spedisce un evento */
public void sendEvent(ICooEvent anEvent)

/* Spedisce un evento ad un insieme di agenti */
public void sendEvent(CooEventRequested anEvent, Vector dests)
```

### Gestore dei processi BPEL (CooBpelProcessesManager)

Il gestore dei processi BPEL gestisce l'archivio locale dei processi BPEL preoccupandosi di installare ed invocare, quando richiesto, i processi BPEL sul motore BPEL esterno (nel nostro caso ActiveBPEL).
Diamo prima uno sguardo alla struttura della classe.
Strutture dati fondamentali:

```
/* Insieme dei processi BPEL disponibili localmente */
private Hashtable<String,ICooBpelProcess> mProcesses;

/* Tabella degli identificatori di processi BPEL associati
   a piani ricevuti da nodi esterni e non ancora utilizzati/acquisiti */
private Hashtable<String,Vector<String>> mNoLocalProcesses;

/* Lista dei processi BPEL attualmente installati sul motore BPEL */
private Vector<String> mProcessesDeployed;
```

Metodi pubblicati:

```
/* L'intenzione di un agente richiede l'esecuzione di un piano-processo */
public void runProcess(
   String anAgentID, String anIntentionID, String aServiceName)

/* Tiene traccia della provenienza di un insieme dei piani ricevuti
   da un nodo remoto */
public void updateNoLocal(CooSetOfPlans aSop, String anHostname)

/* Disinstalla tutti i processi installati dal motore BPEL e salva l'intero
   archivio dei processi in maniera persistente (dove necessario,
   secondo l'implementazione utilizzata) */
public void shutdown(CooEngineConfiguration aConf)
```

spiegare web service/eventi per achieve e planOutcome Altre modifiche consistenti sull'architettura sono state necessarie per mantenere un'operazione importante come la *achieve* (senza la quale diventa irrealizzabile il meccanismo di annidamento delle intenzioni)

### L'agente (CooAgent)

Strutture dati fondamentali:

```
/* L'identificatore univoco dell'agente */
private String mAgentID;

/* La lista degli agenti partner */
private Vector mPartners;

/* La libreria dei piani dell'agente */
private CooSetOfPlans mPlans;

/* La strategia di acquisizione.
   Puo' assumere uno tra i valori:
   - CooAgent.ACQUSITION_STRATEGY_ADD
   - CooAgent.ACQUISITION_STRATEGY_REPLACE
   - CooAgent.ACQUISITION_STRATEGY_DISCARD    */
private int mAcquisitionStrategy;
```

```
/* La strategia di recupero piani.
   Puo' assumere uno tra i valori:
   - CooAgent.RETRIEVAL_STRATEGY_ALWAYS
   - CooAgent.RETRIEVAL_STRATEGY_NOLOCAL  */
private int mRetrievalStrategy;

/* L'insieme delle intenzioni attuali dell'agente: */
private CooSetOfIntention mIntentions;

/* La coda degli eventi dell'agente */
private CooAgentEventsQueue mEventQueue;
```

Metodi pubblicati:

```
/* Costruttore. Ripristina un agente a partire dallo
   stato precedentemente salvato (vedi metodo getAgentState()) */
public CooAgent(CooAgentState anAgentState)

/* Ricezione eventi in ingresso */
public void onEvent(ICooEvent anEvent)

/* Gestione evento di tipo "achieve"*/
public void onEventAchieve(CooEventAchieve anEvent)

/* Gestione evento di tipo "ordinary" */
public void onEventOrdinary(CooEventOrdinary anEvent)

/* Gestione evento di tipo "provided" */
public void onEventProvided(CooEventProvided anEvent)

/* Gestione evento di tipo "requested" */
public void onEventRequested(CooEventRequested anEvent)

/* Fornisce una istantanea sullo stato dell'agente.
   Utilizzato al momento dell'arresto per salvare in maniera
   persistente lo stato dell'agente */
public CooAgentState getAgentState()
```

Come si può osservare l'implementazione dell'agente in CooWS cerca di man-

tenere il più possibile l'indipendenza dal framework che lo ospita. In tal modo sarà
Uno dei pricipi fondamentali adottati durante lo sviluppo di CooWS è stato

## 4.4 La configurazione del motore

Una delle problematiche affrontate durante lo sviluppo della piattaforma CooWS è stata la semplificazione del processo di configurazione dello stesso server istanziato su una moltitudine di macchine distribuite su una rete.
La soluzione adottata è basata sull'utilizzo di un documento testuale XML, chiamato `cooEngineConf.xml`, all'interno del quale è stato creato un albero di nodi ognuno specializzato per la configurazione di un particolare modulo del motore.
In questo documento troviamo quindi una sezione dedicata alla configurazione del gestore dei processi BPEL, una per il gestore degli agenti, una per il catalogo degli agenti, una per la singola istanza di agente più altre sezioni moduli secondari quali il manager delle connessioni al database, il gestore della persistenza, ecc ...
Le sezioni di maggior rilievo sono:

- `agentsDirConf`: dedicata alla configurazione del gestore del catalogo degli agenti. Qui vengono specificati gli URL [49] per l'interrogazione e la pubblicazione del server UDDI, il nome utente e la password assegnati alla particolare istanza del motore CooWS. Osserviamo qui che ogni istanza del motore CooWS deve essere registrato presso il server UDDI con una coppia `(nome utente, password)` attraverso la quale poter accedere alle operazioni di registrazione e cancellazione degli agenti. Questo meccanismo garantisce quindi una sorta di controllo sull'accesso alla rete CooWS: solo gli agenti ospitati su nodi regolarmente registrati presso il server UDDI potranno accedere alla rete.

- `processesManagerConf`: dedicata alla configurazione del gestore dei processi BPEL. Contiene il percorso della cartella del file system contenente l'archivio dei file `.bpr`.

- `pools`: dedicata alla configurazione dei diversi pool di thread utilizzati all'interno del motore. Tra questi troviamo il pool per la gestione della coda di sistema, il pool per la gestione della coda locale del singolo agente ed infine il pool per l'invio dei messaggi in uscita.

Il contenuto di questo documento XML viene caricato in fase di avvio dal motore di CooWS e memorizzato all'interno di un oggetto DOM [47]. Questo oggetto, o meglio la classe wrapper che lo contiene (`CooEngineConfiguration`), verrà poi uti-

lizzato durante l'inizializzazione dei differenti moduli per estrarre i vari parametri di configurazione. Una esempio di possibile configurazione puó essere la seguente:

```
<config>
    <agentsDirConf>
        <uddi-registry>
            <inquiry-url>
               http://localhost:5555/juddi/inquiry
            </inquiry-url>
            <publish-url>
               http://localhost:5555/juddi/publish
            </publish-url>
            <userid>coows1</userid>
            <password>coows1</password>
        </uddi-registry>
    <agentsDirConf>
    <pools>
        <globalDispatchers>
            <maxActive>10</maxActive>
        </globalDispatchers>
        <agentWorkers>
            <maxActive>3</maxActive>
        </agentWorkers>
        <senders>
            <maxActive>15</maxActive>
        </senders>
    </pools>

    <DbConnectionManagerConf>
         <jdbcdriver>
            com.mysql.jdbc.Driver
         </jdbcdriver>
         <dburl>
            jdbc:mysql://localhost/CooWS
         </dburl>
         <dbusername>
             coows_userID
         </dbusername>
         <dbpassword>
             coows_password
```

```
        </dbpassword>
    </DbConnectionManagerConf>

    <processesManagerConf>
        <bprFolder>~/bpr</bprFolder>
    </processesManagerConf>
</config>
```

## 4.5 CooWS all'opera

In questa sezione analizziamo il comportamento del motore CooWS a runtime. Sia per quanto concerne le interazioni a livello di singola istanza sia nel contesto di una rete di nodi interoperanti.

### 4.5.1 Ciclo di vita di CooWS

Il ciclo di vita di una istanza del motore CooWS può essere spezzato in tre fasi principali:

- **Avvio**: All'avvio del servlet container Tomcat viene caricata la servlet CooEngineServlet. Al suo interno in fase di inzializzazione viene invocato il metodo statico `init()` della classe `CooEngine` il quale si occupa di avviare il server CooWS ripristinandone l'ultimo stato salvato su database.
  Nell'ordine si effettuano le seguenti operazioni:
  - Si carica il file di configurazione XML `cooEngineConf.xml` in un documento DOM (Document Object Model) **??**. Questo documento viene gestito da un'istanza della classe `CooEngineConfiguration` la quale fornisce alcuni metodi per una facile consultazione del documento. Tale oggetto viene utilizzato nell'inizializzazione dei differenti componenti.
  - Si ripristina il gestore dei processi BPEL (`CooProcessesManager`). Vengono caricati i processi BPEL. dal database MySQL (tramite il modulo Persistence Manager).
  - Si riattiva l'interfaccia con il registro UDDI (`CooAgentsDirectory`).
  - Si ripristina il gestore degli agenti (`CooAgentsManager`). Questo si occupa di reinizializzare gli agenti allo stato precedente l'ultimo arresto (attraverso il Persistence Manager). Le code eventi vengono ricreate. Gli agenti non vengono ancora attivati (letteralmente i thread non sono ancora in esecuzione).

– Attraverso l'interfaccia con il registro UDDI si effettua la pubblicazione degli agenti locali

– Vengono infine attivati gli agenti.

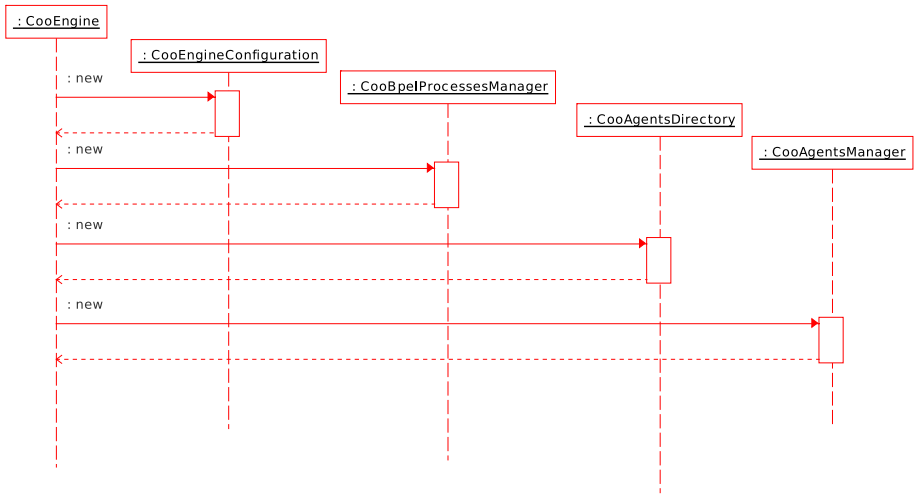



Figura 4.3: Avvio del server CooWS

Se la fase di avvio termina con successo un messaggio di notifica (`CooWS engine started!`) viene inserito all'interno del log del server tomcat.

- **Esecuzione**: Il server CooWS, una volta attivo, osserva un ciclo di esecuzione guidato dagli eventi. Si distinguono gli eventi provenienti da agenti appartenenti alla rete CooWS oppure da agenti/utenti esterni. Un'altra distinzione, importante a livello di implementazione, è tra gli eventi scambiati localmente e gli eventi scambiati via rete. Nel primo caso l'evento viene passato dal gestore del catalogo direttamente al gestore degli agenti il quale lo accoderà nella coda eventi globale di sistema. Nel secondo caso si attiva una procedura più complessa che, dal lato mittente, prevede:

  – localizzazione dell'agente tramite registro UDDI.

  – serializzazione dell'evento all'interno di un messaggio SOAP ed invio tramite rete all'host destinazione.

  Dalla parte del server ospitante l'agente destinatario troviamo invece la sequenza:

  – ricezione nuovo evento tramite Web Service.

- tracciamento dell'host sorgente per eventuali future comunicazioni.
- controllo presenza agente destinatario
- se il controllo precedente ha esito positivo si inserisce l'evento nella coda di sistema

Gli eventi in ingresso ricevuti tramite il Web Service vengono inoltrati al gestore degli agenti locale che si occupa di inserirli nella coda eventi di sistema la quale a sua volta si occuperà di smistarli nella coda eventi locale degli agenti destinatari.
Ogni agente, in base all'evento ricevuto potrà inviare un evento di risposta all'agente/utente mittente, creare una nuova intenzione ed eventualmente inviare un evento di tipo *requested* agli agenti partner ed infine richiedere al gestore dei processi l'esecuzione di un determinato piano-processo.
Nella fase di ricezione eventi sono stati introdotti alcuni controlli e miglioramenti per ottimizzare le prestazioni del sistema. Una prima miglioria è stata il tracciamento dell'host sorgente per quanto concerne gli eventi di tipo *requested*. Ogni volta che si riceve un messaggio avente come mittente un agente esterno si forza l'aggiornamento della cache del gestore del catalogo degli agenti con la coppia `(agent_ID, source_host)` in modo da minimizzare le operazioni di accesso al registro centrale UDDI. Infatti, nel caso l'evento ricevuto generi un evento di risposta (ad esempio nel caso di uno scenario requested-provided), l'operazione di invio non avrà bisogno di interrogare il registro UDDI disponendo già di tutte le informazioni necessarie.
Un'altra ottimizzazione, già introdotta nel capitolo precedente, riguarda il tracciamento dell'host sorgente per quanto riguarda gli eveni di tipo *provided* al fine di minimizzare la banda occupata dal trasferimento dei processi BPEL. Quando un agente riceve un insieme di piani rilevanti tramite l'operazione provided in verità esso riceve solo una lista di identificatori di processi BPEL. Tali identificatori devono poi essere associati a processi realmente presenti nell'archivio locale del gestore dei processi BPEL affinchè l'agente ne possa richiedere una eventuale esecuzione. Nel caso uno dei piani forniti sia associato ad un processo assente dall'archivio locale sarà carico del gestore dei processi effettuare il recupero del file (nel caso dell'implementazione da noi scelta del file bpr) dall'host sorgente precentemente tracciato. Questa operazione di recupero andrà inoltre effettuata anche nel caso l'agente aggiorni la propria libreria di piani con il nuovo piano pur non utilizzandolo: questo per avere sempre una sincronizzazione tra l'insieme dei piani in possesso degli agenti locali e l'insieme dei processi BPEL.

- **Arresto**: Quando il server Tomcat viene fermato la servlet ne riceve notifica

ed a sua volta richiede l'arresto del motore CooWS invocando il metodo `shutdown()` della classe singleton `CooEngine`.
Nell'ordine vengono eseguite le seguenti operazioni:

- Il gestore del catalogo degli agenti cancella dal registro centrale UDDI le informazioni degli agenti ospitati localmente.
- Il gestore degli agenti svuota la coda eventi globale, blocca il ciclo di esecuzione degli agenti e ne salva lo stato in maniera persistente tramite il gestore della persistenza (`CooPersistenceManager`).
- Il gestore dei processi richiede la rimozione di tutti i processi installati sull'engine BPEL.

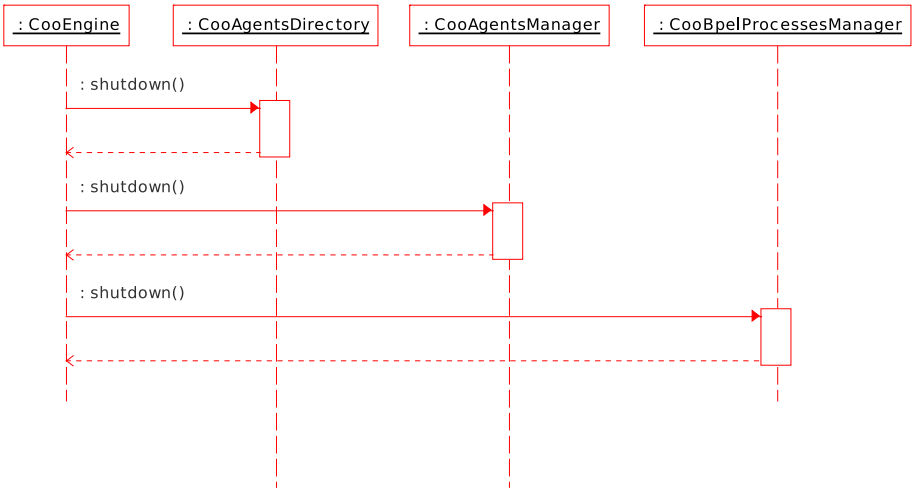



Figura 4.4: Arresto del server

# Capitolo 5

# Conclusioni

## 5.1 La piattaforma sviluppata

CooWS, il prototipo progettato e sviluppato in questo lavoro di tesi, rappresenta un primo passo per lo sviluppo di un framework BDI basato sui Web Service. I vantaggi di questa integrazione sono principalmente due:

- il modello ad agenti in stile BDI viene applicato in un ambiente software reale quale il Web.
- l'integrazione ed il confronto del mondo degli agenti intelligenti con quello dei Web Service ha qui evidenziato alcune carenze di quest'ultimo nel caso del supporto ad una conoscenza procedurale adattiva.

Documenti degni di nota nella prospettiva dei Semantic Web Service ed in qualche modo correlati al lavoro svolto sono stati proposti da P.Buhler e J.M. Vidal in [50] dove gli autori propongono l'utilizzo di BPEL per la coordinazione delle interazioni all'interno di un sistema multiagente in base ai cambiamenti che si possono verificare all'interno dell'ambiente di esecuzione.
Un'altra proposta guidata da motivazioni simili alle nostre è quella di K. Sycara, M. Paolucci, J. Soudry e N. Srinivasan i quali suggeriscono l'estensione del linguaggio OWL-S [51] con un'operazione chiamata `exec` per l'esecuzione del modello di processi al fine di supportare un agente broker nel processo di ricerca e interazione con gli agenti partner.
Sebbene molto lavoro sia già stato svolto nel campo degli agenti applicato alle specifiche di Web Service non è stato possibile trovare ulteriori proposte per l'adozione di linguaggi di specifica per processi basati sui Web Service nella definizione del-

la conoscenza procedurale di un singolo agente così come di una comunità di agenti.

## 5.2 Sviluppi Futuri

Nella prospettiva di sviluppi futuri, la piattaforma CooWS potrà focalizzarsi meglio nella direzione dei semantic web service e più in generale nell'ottica di una Semantic Web Service Architecture (SWSA), sfruttando i linguaggi di descrizione semantica quali OWL-S [51] e DAML-S [52]. Saranno inoltre indicati miglioramenti e integrazioni per quanto concerne la gestione dei processi BPEL (meccanismo di deploy ad esempio), la rappresentazione di desire e trigger con linguaggi di descrizione semantica

### 5.2.1 Verso i Semantic Web Service

#### Etensioni semantiche di UDDI

**??** L'utilizzo che è stato fatto di UDDI in questo lavoro di tesi, sebbene essenziale, non ne ha sfruttato appieno le vere potenzialità. La specifica di UDDI supporta differenti tipi di registrazione nonché articolati meccanismi di ricerca. In un contesto di utilizzo reale la flessibilità del sistema CooWS potrebbe essere notevolmente incrementata attraverso l'introduzione di capacità di ricerca e interrogazione per gli agenti. La specifica di Coo-BDI non fa cenno ad una tale funzionalità ma in un sistema aperto e flessibile quale il Web è sensato, se non imperativo, considerare una simile eventualità. Sono di particolare interesse alcuni lavori recenti orientati all'integrazione di UDDI con OWSL-S [53], un linguaggio derivato da OWL [54] con il quale è possibile arricchire la descrizione dei Web Service con informazioni di natura semantica.

#### Ontologie per matching trigger-desideri

La specifica Coo-BDI, sfruttando le potenzialità del linguaggio Prolog, definisce il matching tra il trigger di un piano ed un desire attraverso il meccanismo di unificazione. In un contesto Web ovviamente non è più possibile parlare di ambiente di variabili o sostituzione o ancora di unificazione. Le ontologie risultano essere la soluzione ideale per dotare i piani con una descrizione semantica adeguata.

### 5.2.2 Ulteriori miglioramenti

#### Agenti mobili

Una proprietà interessante degli agenti mobili è la capacità di poter migrare in maniera indolore tra diversi ambienti di esecuzione. Nella attuale piattaforma non sarà particolarmente gravosa una simile estensione, consentendo quindi agli agenti la migrazione tra i nodi della rete CooWS e aprendo così di fatto la strada verso i dispositivi integrati (PDA, cellulari, ecc.)

#### Scoperta di nuovi partner

In CooWS ogni agente possiede una lista "statica" degli agenti a lui noti. Non è previsto nessun meccanismo di ricerca e scoperta di nuovi agenti a runtime.
In tempi recenti la crescente diffusione delle reti p2p ha favorito la ricerca nel campo del "reputation and trust" [39] (letteralmente stima e fiducia). In breve si sono studiate tecniche per la valutazione della credibilità di nodi appartenenti ad una stessa rete.
L'applicazione di simili concetti a CooWS porterebbe all'architettura vantaggi in termini di flessibilità e adattabilità, caratteristiche che, in un contesto Web, non è possibile trascurare.
Allo stato corrente CooWS sfrutta UDDI per la registrazione degli agenti in un registro centrale. Lo scenario ben si presta ad una futura estensione di questo genere sebbene dovranno essere affrontate problematiche inerenti ai criteri di ricerca.

### 5.2.3 Interfaccia utente

Sul versante utilizzabilità ed interfaccia utente molto deve essere ancora svolto. Sebbene CooWS sia stato progettato come sistema autonomo e autosufficiente, sarebbe utile disporre di una interfaccia grafica in stile pannello di controllo attraverso la quale poter monitorare il comportamento del sistema.
Tra le modalità di visualizzazione saranno certamente indicate le seguenti:

- **piani**: lista dei piani installati attualmente sull'engine BPEL locale, con lista delle invocazioni passate ed in corso.
- **code eventi**: possibilità di visionare il contenuto della coda eventi generale e locale ad ogni singolo agente, con possibilità di modifica del contenuto tramite operazioni di di cancellazione, modifica, riordino, inserimento.
- **agenti**: possibilità di visualizzare nel dettaglio lo stato del singolo agente con accesso alle strutture dati (piani, intenzioni, coda eventi, lista agenti, strategia di cooperazione).

# Appendice A

# Manuale

## A.1 Configurazione tipo

Illustriamo in questa sezione i prerequisiti per la corretta installazione di una rete CooWS. Una configurazione tipica per un sistema CooWS prevede un insieme di nodi distribuiti sul Web costituiti da due tipologie di server::

– Server CooWS: l'engine CooWS, ospitato da un servlet container, si interfaccia con un database locale per la gestione della persistenza.

– Server jUDDI: implementazione del server UDDI, poggia anch'esso su un servlet container ed un database.

Pertanto, elementi essenziali per un corretta installazione di CooWS sono:

– Installazione di un server **jUDDI**. Il server UDDI deve essere unico e pertanto jUDDI dovrà essere installato su una sola macchina. jUDDI presuppone la presenza di un runtime Java, di un servlet container (qui Tomcat) e di un database (qui MySQL).

– Installazione di uno o più server CooWS. Per ognuno di questi sarà necessario predisporre un runtime Java un server Tomcat [41], un'istanza del motore BPEL ActiveBPEL [35] ed infine una installazione del database MySQL [44], tutti liberamente scaricabili dai rispettivi siti Web.

Lo sviluppo ed i test si sono svolti in ambiente linux sebbene tutti gli strumenti utilizzati possano essere installati su piattaforma Windows. Qui di seguito illustriamo le linee generali per una corretta installazione del sistema CooWS su tale piattaforma.

## A.2 Installazione

### Linux

Non vi sono indicazioni particolari per quanto concerne la distribuzione linux. I pacchetti utilizzati in questo lavoro (JDK Java, Apache Tomcat, MySQL) sono disponibili per la maggior parte delle distribuzioni attualmente sul mercato.
*Nota*: Sviluppo e test di CooWS sono stati svolti utilizzando la distribuzione Ubuntu Linux 5.04.

### Java

#### Installazione

Scaricare da sito `http://java.sun.com` l'ultima versione del Java Development Kit (JDK) ed installarlo in una directory di sistema (ad esempio `/usr/java/jdk1.5`).

#### Configurazione

Impostare la variabile di ambiente `JAVA_HOME` con il path della cartella di installazione. Aggiungere la cartella `JAVA_HOME/bin` al PATH.

#### Test dell'installazione

Eseguire da shell il comando java. Se l'installazione è stata eseguita correttamente il comando stamperà a video un breve sunto delle opzioni disponibili (`Usage: java` ... ecc).

### MySQL

#### Installazione

Se la distribuzione utilizzata ha una gestione automatizzata per l'installazione dei pacchetti (apt, urpmi, yum, ecc ...) cercare ed installare tramite l'opportuna interfaccia il pacchetto del server MySQL. In caso contrario scaricare dal sito `http://www.mysql.com` il pacchetto compresso, scompattarlo e seguire le istruzioni contenute nella documentazione.

#### Configurazione

Sará necessario abilitare l'accesso via rete nel caso l'istallazione di default non preveda tale opzione attivata. Per fare ciò, nel file `/etc/mysql/my.cnf`, commentare la linea con l'opzione "`skip-networking`"

### Creazione utenti e database

Nella cartella `conf` del pacchetto CooWS troverete il DDL per la creazione automatica dei database e degli utenti necessari all'engine. Con l'account di amministratore lanciate il comando `mysql < create_coodb.sql`.

### Test dell'installazione

Tramite la console testuale (oppure una console grafica quale MySQL Control Center) controllate la presenza del database CooWS e dell'utente `coowsuserID`

## Apache Tomcat

### Installazione

Scaricare dal sito web `http://tomcat.apache.org` l'ultima versione del server tomcat 5.0 (sviluppo e test di CooWS hanno utilizzato la versione 5.0.28).
Una volta scompattato l'archivio in un cartella a propria scelta (ad esempio `/home/-user/apps/tomcat`) impostare la variabile di ambiente `CATALINA_HOME` con il percorso di tale cartella (ad esempio, nel caso di shell bash, con il comando `export CATALINA_HOME = /home/user/apps/tomcat`)

### Test dell'installazione

Avviare il server Tomcat (con lo script `$CATALINA_HOME/bin/startup.sh` e controllare l'esito visitando con un web browser la pagina Web presente all'indirizzo `http://localhost:8080/`.

## ActiveBPEL

### Installazione

Scaricare dal sito web `http://www.activebpel.org` la versione 1.0.2 dell'engine ActiveBPEL. Scompattare l'archivio in una directory temporanea e lanciare lo script `install.sh`

### Test dell'installazione

Riavviare il server Tomcat e puntare il browser alla pagina `http://localhost:8080/-BpelAdmin`. Se non si verificano errori provare il processo di esempio `loan_approval` che si trova nella cartella `samples` di ActiveBPEL.

# jUDDI

## Installazione

Scaricare dal sito `http://juddi.apache.org` l'ultima versione del server jUDDI (sviluppo e test di CooWS hanno utilizzato la versione 0.9rc3).
Una volta scompattato l'archivio in una cartella temporanea, copiare la sottocartella `juddi` nella cartella `$CATALINA_HOME/webapps`.

## Installazione

Modificare il file di configurazione di tomcat `$CATALINA_HOME/conf/server.xml` aggiungendo il contenuto del file `tomcatContext4juddi.txt` immediatamente prima del tag `</Host>`

## Test dell'installazione

Riavviare il server tomcat e puntare il browser alla pagina `http://localhost:8080/-juddi` e quindi cliccare sul link `validate`. La pagina che comparirà potrà contenere dei messaggi di avviso (*warning*) ma nessun errore.

# CooWS

## Installazione

Scaricare dal sito `http://coows.altervista.org` il pacchetto `coows.tar.gz`, scompattarlo in una cartella temporanea e lanciare lo script `install.sh`.

## Configurazione

Aprire in un editor testuale il file `$CATALINA_HOME/conf/cooEngineConf.xml` e modificare opportunamente i parametri relativi al database MySQL (nel caso il server si trovi su un'altra macchina) ed al server jUDDI. Ulteriori informazioni per una corretta configurazione si trovano all'interno dei file stesso,

## Test dell'installazione

Riavviare il server tomcat e controllarne il log di avvio. Una linea simile a
`[CooEngine] Init successful!`
segnalerà il corretto avvio dell'engine CooWS.

## A.3 Installazione tramite codice sorgente

La compilazione del codice sorgente di CooWS richiede l'installazione dei seguenti strumenti/server nell'ordine qui elencato: Sun JDK 1.5, Apache Ant, Apache Tomcat, ActiveBPEL.
Una volta installati tali strumenti/server è necessario scaricare dal sito `http://coows.altervista.org` l'ultima versione del server CooWS, scompattare l'archivio compresso in una cartella temporanea e quindi lanciare da essa il comando `ant dist && ./install.sh`. Il messaggio a video `Congratulations, you have just installed CooWS!` segnalerà la conclusione con successo dell'installazione. A questo punto sarà possibile effettuare il test dell'installazione avviando il server tomcat e controllarne nel log la presenza del messaggio `CooWS engine started!`.

# Appendice B

# Codice sorgente

## B.1 Package org.coows

*org.coows.CooAgent*

```
package org.coows;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.coows.engine.CooEngine;
import org.coows.engine.CooException;

import org.coows.util.CooAgentState;
import org.coows.util.CooRequestID;
import org.coows.util.CooSetOfIntention;
import org.coows.util.CooSetOfPlans;


import java.util.Calendar;
import java.util.Vector;

/**
 * The CooWS agent
 */
public class CooAgent
    extends Thread {
    /**
     * Logger for this class
     */
    private static final Log logger = LogFactory.getLog(CooAgent.class);

    /* Retrieval strategy constants */
    public static final int RETRIEVAL_STRATEGY_ALWAYS = 0;

    public static final int RETRIEVAL_STRATEGY_NOLOCAL = 1;

    /* Acquisition strategy constants */
    public static final int ACQUISITION_STRATEGY_ADD = 0;

    public static final int ACQUISITION_STRATEGY_REPLACE = 1;

    public static final int ACQUISITION_STRATEGY_DISCARD = 2;

    /**
     * The agent ID
     */
    private String mAgentID;

    /**
     * The list of known agents
     */
    private Vector mPartners;

    /**
     * The agent plan library
     *
     * @uml.property name="mPlans"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private CooSetOfPlans mPlans;
```

```
/**
 * The acquisition strategy.<br>
 * Values range within the following values: <br>
 * CooAgent.ACQUSITION_STRATEGY_ADD <br>
 * CooAgent.ACQUISITION_STRATEGY_REPLACE <br>
 * CooAgent.ACQUISITION_STRATEGY_DISCARD
 */
private int mAcquisitionStrategy;

/**
 * The retrieval strategy. <br>
 * Values range within the following values: <br>
 * CooAgent.RETRIEVAL_STRATEGY_ALWAYS <br>
 * CooAgent.RETRIEVAL_STRATEGY_NOLOCAL
 */
private int mRetrievalStrategy;

/**
 * The intentions table <br>
 * For each intention ID a CooIntention is
 *
 * @uml.property name="mIntentions"
 * @uml.associationEnd multiplicity="(0 1)"
 */
private CooSetOfIntention mIntentions;

/**
 * The agent's event queue
 *
 * @uml.property name="mEventQueue"
 * @uml.associationEnd multiplicity="(0 1)"
 */
private CooAgentEventsQueue mEventQueue;


private boolean mPrepare4shutdown;

private boolean mShutdown;

/**
 * Constructor
 *
 * @param anAgentState
 *            A previously serialized agent state
 * @throws CooException
 */
public CooAgent(CooAgentState anAgentState)
    throws CooException {
    init();
    mAgentID = anAgentState.getMAgentID();
    mPlans = anAgentState.getMSop();
    mPartners = anAgentState.getMPartners();
    mAcquisitionStrategy = anAgentState.getMAcquisitionStrategy();
    mRetrievalStrategy = anAgentState.getMRetrievalStrategy();

    if (logger.isDebugEnabled()) {
        logger.debug("CooAgent() - agent " + mAgentID + " created");
    }
}

/**
 * Auxiliary constructor (for debug and test)
```

```
 *
 * @param anAgentID
 * @param anAcquisitionSTRATEGY
 * @param aSop
 * @throws CooException
 */
public CooAgent(
        String anAgentID, int anAcquisitionStrategy, CooSetOfPlans aSop)
    throws CooException {
    init();

    mAgentID = anAgentID;
    mAcquisitionStrategy = anAcquisitionStrategy;
    mPlans = aSop;

    if (logger.isDebugEnabled()) {
        logger.debug("CooAgent() - agent " + mAgentID + " created");
    }
}



/**
 * Init the local data structures (event queue, workers pool and boolean
 * variables)
 *
 * @throws CooException
 */
public void init()
        throws CooException {
    //mEventQueueOLD = new LinkedList();
    mEventQueue = new CooAgentEventsQueue(this);
    //mWorkers = new GenericObjectPool(new CooWorkerProcessEventFactory());
    //mMaxActiveWorkers = Integer.parseInt(CooEngine.getMConfig().getText(
      //      "/config/pools/agentWorkers/maxActive"));
    //mWorkers.setMaxActive(mMaxActiveWorkers);
    mPrepare4shutdown = false;
    mShutdown = false;
}

/**
 * Process new incoming events (assigns the work to a new thread or enqueue
 * locally)
 *
 * @param anEvent
 */
public void onEvent(ICooEvent anEvent) {
    if (logger.isDebugEnabled()) {
        logger.debug("agent: " + mAgentID + ", " + "onEvent(anEvent = "
                + anEvent + ")");
    }

    // if we are going to shutdown
    if (true == mPrepare4shutdown)
    // discharge all events
        return;

    mEventQueue.enqueue(anEvent);
}

/**
```

```
     * Callback for "achieve" requests
     *
     * @param anEvent
     */
    public void onEventAchieve(CooEventAchieve anEvent) {
        if (logger.isDebugEnabled()) {
            logger.debug("Agent: " + mAgentID
                    + "onEventAchieve(" + anEvent + ")");
        }

        CooSetOfPlans relPlans = mPlans.getRelevant(anEvent.getMDes());

        CooInstanceRecord instRec =
            new CooInstanceRecord(anEvent.getMDes(), relPlans);


        try {
            CooIntention theInt;
            theInt = mIntentions.getIntention(anEvent.getMIntentionID());
            String instID = theInt.push(instRec);
            // TODO lanciare in esecuzione un piano se relplans NON vuoto

            if ((RETRIEVAL_STRATEGY_ALWAYS == mRetrievalStrategy)
                    || ((RETRIEVAL_STRATEGY_NOLOCAL == mRetrievalStrategy)
                            && relPlans.isEmpty()))
                sendRequested(
                        anEvent.getMDes(), anEvent.getMIntentionID(), instID);
        }
        catch (CooException e) {
            if (logger.isErrorEnabled()) {
                logger.error("Agent "+ mAgentID +" encountered an exception " +
                        "processing an <achieve> message for intention " +
                        anEvent.getMIntentionID(),e);
            }
        }

    }

    public void onEventOrdinary(CooEventOrdinary anEvent) {
        String intID =
            String.valueOf(Calendar.getInstance().getTimeInMillis());

        // Create a new intention
        CooIntention newInt = new CooIntention(mAgentID, intID,
                anEvent.getMSuccessAction(), anEvent.getMFailureAction());

        // Retrieve relevant plans
        CooSetOfPlans relPlans = mPlans.getRelevant(anEvent.getMDesire());

        // Create the first instance record for the intention stack
        CooInstanceRecord instRec =
            new CooInstanceRecord(anEvent.getMDesire(), relPlans);

        // Push the record on the stack
        newInt.push(instRec);

        // Update the intentions set
        mIntentions.addIntention(newInt);

        // If available, run the first relevant plan
        if (!relPlans.isEmpty())
            instRec.runNextPlan();
```

```
    // Send requested
    if ((mRetrievalStrategy == RETRIEVAL_STRATEGY_ALWAYS)
         || ((mRetrievalStrategy == RETRIEVAL_STRATEGY_NOLOCAL)
                 && (relPlans.isEmpty())))
        sendRequested(anEvent.getMDesire(), intID, "0");
}

public void onEventPlanOutcome(CooEventPlanOutcome anEvent) {
    try {
        CooIntention anInt =
            mIntentions.getIntention(anEvent.getMIntentionID());
        anInt.dispatchPlanOutcome(anEvent);
    }
    catch (CooException e) {
        if (logger.isErrorEnabled()) {
            logger.error("Agent " + mAgentID + " encountered an exception" +
                    " processing a <planOutcome> message for intention" +
                    anEvent.getMIntentionID(), e);
        }
    }
}

/**
 * Callback for incoming "provided" events
 *
 * @param anEvent
 */
public void onEventProvided(CooEventProvided anEvent) {
    if (logger.isDebugEnabled()) {
        logger.debug("onEventProvided() - Agent " + mAgentID +
                ": Processing " + anEvent);
    }


    if (mAcquisitionStrategy == ACQUISITION_STRATEGY_ADD) {
        mPlans.addPlans(anEvent.getMProvidedPlans());
    } else if (mAcquisitionStrategy == ACQUISITION_STRATEGY_REPLACE)
        mPlans.replace(anEvent.getMProvidedPlans());
    else if (mAcquisitionStrategy == ACQUISITION_STRATEGY_DISCARD)
               ; // nothing to do
    else
        if (logger.isErrorEnabled()) {
            logger.error("onEventProvided() - Agent " + mAgentID +
                    "has an unknown acquisition policy!", null);
        }

    try {
        CooIntention anInt =
            mIntentions.getIntention(anEvent.getMReqID().getMIntentionID());
        anInt.dispatchProvided(anEvent);
    }
    catch (CooException e) {
        if (logger.isErrorEnabled()) {
            logger.error("Agent " + mAgentID + " encountered an exception" +
                    " processing a <planOutcome> message for intention" +
                    anEvent.getMReqID().getMIntentionID(), e);
        }
    }


    // TODO update the intention stack
```

```
}

/**
 * Callback for incoming "requested" events
 *
 * @param anEvent
 */
public void onEventRequested(CooEventRequested anEvent) {
    if (logger.isDebugEnabled()) {
        logger.debug("onEventRequested() - Agent " + mAgentID +
                ": Processing " + anEvent);
    }

    CooSetOfPlans res = mPlans.getRelevantForAgent(anEvent.getMAgentFROM(),
            anEvent.getMDes());

    if (res.isEmpty()) {
        if (logger.isDebugEnabled()) {
            logger.debug("onEventRequested() - no relevant plans " +
                    "found for " + anEvent.getMDes());
        }

        return;
    }
    else {
        // let's build the reply event...
        CooEventProvided evt = new CooEventProvided(mAgentID, anEvent
                .getMAgentFROM(), anEvent.getMReqID(), res);
        // ... and send it!
        CooEngine.getAgentsDirectory().sendEvent(evt);
    }
}

public void prepare4shutdown() {
    this.mPrepare4shutdown = true;
}

public void run() {
    while (!mPrepare4shutdown) {
        try {
            sleep(5000);
        }
        catch (InterruptedException e) {}
    }

    //TODO Gestire intenzioni in esecuzione

    while (!mShutdown) {

    }

    // TODO


    if (logger.isDebugEnabled()) {
        logger.debug("Agent " + mAgentID + " stopped!");
    }
}

public void shutdown() {
    this.mShutdown = true;
```

```
    }

    /**
     * Creates and returns a snapshot of the agent state
     *
     * @return
     */
    public CooAgentState getAgentState() {
        return new CooAgentState(mAgentID, mPlans, mPartners,
                mRetrievalStrategy, mAcquisitionStrategy);
    }

    /**
     *
     * @uml.property name="mEventQueue"
     */
    public CooAgentEventsQueue getMEventQueue() {
        return mEventQueue;
    }

    /**
     *
     * @uml.property name="mAgentID"
     */
    public String getMAgentID() {
        return mAgentID;
    }

    /**
     * Construct and send a "requested" event to the set of partner agents
     *
     * @param aDes
     *            The desire
     * @param intD
     *            The intention source of the request
     * @param stackPos
     *            The position inside the intention stack of the
     * @param instanceID
     *            The id of the instance record
     */
    private void sendRequested(
            ICooDesire aDes,
            String intID,
            String instanceID) {

        CooEventRequested aReq = new CooEventRequested(mAgentID, "",
                new CooRequestID(intID, instanceID), aDes);
        CooEngine.getAgentsDirectory().sendEvent(aReq, mPartners);
    }
}
```

*org.coows.CooDesire*

```
package org.coows;

import org.coows.ICooDesire;

/**
 * A simple desire implementation
 */
public class CooDesire
```

```
implements ICooDesire {

    /**
     * A desire is a string
     */
private String mDes;

/**
 * Constructor
 *
 */
public CooDesire() {
}


/**
 * Constructor
 */
public CooDesire(String aDes) {
this.mDes = aDes;
}


public String toString() {
return "Desire("+mDes+")";
}


/**
 *
 * @see org.coows.ICooDesire#match(org.coows.ICooDesire)
 */
public boolean match(ICooDesire aDes) {
        if (aDes  instanceof CooDesire)
            return mDes.equals(((CooDesire)aDes).getMDes());
        else
            return false;
}

    /**
     *
     * @uml.property name="mDes"
     */
    public String getMDes() {
        return mDes;
    }

    /**
     *
     * @uml.property name="mDes"
     */
    public void setMDes(String des) {
        mDes = des;
    }

}
```

*org.coows.CooEventAchieve*

```
package org.coows;
```

```
/**
 * This event carries the info for the "achieve" op
 */
public class CooEventAchieve
extends CooAbstractEvent {

    /**
     * The desire to achieve
     *
     * @uml.property name="mDes"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private ICooDesire mDes;


    /**
     * The intention ID wich belongs the plan
     * that generate this event
     */
private String mIntentionID;


    /**
     * Constructor
     * @param anAgentTO
     * @param anIntentionID
     * @param aDes
     */
public CooEventAchieve(
            String anAgentTO,
            String anIntentionID,
            String aDes) {
mAgentTO = anAgentTO;
mIntentionID = anIntentionID;
mDes = new CooDesire(aDes);
}

    /**
     *
     * @uml.property name="mDes"
     */
    public ICooDesire getMDes() {
        return mDes;
    }

    /**
     *
     * @uml.property name="mDes"
     */
    public void setMDes(ICooDesire des) {
        mDes = des;
    }



/**
 * @see org.coows.ICooEvent#dd(org.coows.CooAgent)
 */
public void dd(CooAgent anAgent) {
anAgent.onEventAchieve(this);
}
```

```
    /**
     *
     * @uml.property name="mIntentionID"
     */
    public String getMIntentionID() {
        return mIntentionID;
    }

    /**
     *
     * @uml.property name="mIntentionID"
     */
    public void setMIntentionID(String intentionID) {
        mIntentionID = intentionID;
    }


    public String toString() {
        return "EventAchieve("+mAgentTO+","+mIntentionID+","+mDes+")";
    }


}
```

*org.coows.CooEventOrdinary*

```
package org.coows;

/**
 * The ordinary event
 */
public class CooEventOrdinary
extends CooAbstractEvent {

    /**
     * The desires
     *
     * @uml.property name="mDesire"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private Vector<>ICooDesire> mDesires;

    /**
     * List of actions to be performed when the desire is achieved
     *
     * @uml.property name="mSuccessAction"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private ICooAction mSuccessAction;

    /**
     * List of actions to be performed when the desire isn't achieved
     *
     * @uml.property name="mFailureAction"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private ICooAction mFailureAction;


    /**
     * Constructor
```

```
     * @param anAgentTO
     */
public CooEventOrdinary(
            String anAgentTO,
            ICooDesire aDesire,
            ICooAction aSuccessAction,
            ICooAction aFailureAction) {
mAgentTO = anAgentTO;
        mDesire = aDesire;
        mSuccessAction = aSuccessAction;
        mFailureAction = aFailureAction;
}


/**
 * @see org.coows.ICooEvent#dd(org.coows.CooAgent)
 */
public void dd(CooAgent anAgent) {
anAgent.onEventOrdinary(this);
}

    public String toString() {
        return "EventOrdinary("+mAgentTO+")";
    }

    /**
     *
     * @uml.property name="mDesire"
     */
    public ICooDesire getMDesire() {
        return mDesire;
    }

    /**
     *
     * @uml.property name="mDesire"
     */
    public void setMDesire(ICooDesire desire) {
        mDesire = desire;
    }

    /**
     *
     * @uml.property name="mFailureAction"
     */
    public ICooAction getMFailureAction() {
        return mFailureAction;
    }

    /**
     *
     * @uml.property name="mFailureAction"
     */
    public void setMFailureAction(ICooAction failureAction) {
        mFailureAction = failureAction;
    }

    /**
     *
     * @uml.property name="mSuccessAction"
     */
    public ICooAction getMSuccessAction() {
```

```
        return mSuccessAction;
    }

    /**
     *
     * @uml.property name="mSuccessAction"
     */
    public void setMSuccessAction(ICooAction successAction) {
        mSuccessAction = successAction;
    }

}
```

org.coows.CooEventPlanOutcome

```
package org.coows;

/**
 *  The event for the dispatch of plan outcome
 */
public class CooEventPlanOutcome
extends CooAbstractEvent {

    /** The intention ID wich belongs the completed plan */
private String mIntentionID;

    /** The outcome: TRUE (successfully completed) or FALSE (failed) */
private boolean mOutcome;


    /**
     * Constructor
     * @param anAgentTO
     * @param anIntentionID
     * @param anOutcome
     */
public CooEventPlanOutcome(String anAgentTO, String anIntentionID,
boolean anOutcome) {
mAgentTO = anAgentTO;
mIntentionID = anIntentionID;
mOutcome = anOutcome;
}

    /**
     *
     * @uml.property name="mIntentionID"
     */
    public String getMIntentionID() {
        return mIntentionID;
    }

public boolean isSuccess() {
return mOutcome;
}


/**
 * @see org.coows.ICooEvent#dd(org.coows.CooAgent)
 */
public void dd(CooAgent anAgent) {
anAgent.onEventPlanOutcome(this);
```

```
}

    public String toString() {
        return "EventPlanOutcome("+mAgentTO+","+mIntentionID+","+mOutcome+")";
    }
}
```

*org.coows.CooEventProvided*

```
package org.coows;

import org.coows.util.CooRequestID;
import org.coows.util.CooSetOfPlans;

/**
 * The event for "provided" op
 */
public class CooEventProvided
extends CooAbstractEvent {

    /**
     * The request ID
     *
     * @uml.property name="mReqID"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private CooRequestID mReqID;


    /** The agent sender */
private String mAgentFROM;

    /**
     * The set of provided plans
     *
     * @uml.property name="mProvidedPlans"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private CooSetOfPlans mProvidedPlans;


/**
 * Constructor
 *
 */
public CooEventProvided() {
mAgentTO = "";
mReqID = null;
        mAgentFROM = "";
mProvidedPlans = new CooSetOfPlans();
}

/**
 * Constructor
 * @param anAgentFROM
 * @param anAgentTO
 * @param aReqID
 * @param aSop
 */
public CooEventProvided(String anAgentFROM, String anAgentTO,
CooRequestID aReqID, CooSetOfPlans aSop) {
```

```
mReqID = aReqID;
mAgentFROM = anAgentFROM;
mAgentTO = anAgentTO;
mProvidedPlans = aSop;
}

    /**
     *
     * @uml.property name="mAgentFROM"
     */
    public void setMAgentFROM(String agentProvider) {
        mAgentFROM = agentProvider;
    }

    /**
     *
     * @uml.property name="mAgentFROM"
     */
    public String getMAgentFROM() {
        return mAgentFROM;
    }

    /**
     *
     * @uml.property name="mProvidedPlans"
     */
    public void setMProvidedPlans(CooSetOfPlans providedPlans) {
        mProvidedPlans = providedPlans;
    }

    /**
     *
     * @uml.property name="mProvidedPlans"
     */
    public CooSetOfPlans getMProvidedPlans() {
        return mProvidedPlans;
    }

    /**
     *
     * @uml.property name="mReqID"
     */
    public void setMReqID(CooRequestID reqID) {
        mReqID = reqID;
    }

    /**
     *
     * @uml.property name="mReqID"
     */
    public CooRequestID getMReqID() {
        return mReqID;
    }

public String toString() {
return "EventProvided("+mAgentFROM+","
            +mAgentTO+","+mReqID+","+mProvidedPlans+")";
}


    /**
     *
```

```
     * @see org.coows.ICooEvent#dd(org.coows.CooAgent)
     */
public void dd(CooAgent anAgent) {
anAgent.onEventProvided(this);
}
}
```

*org.coows.CooEventRemote*

```
package org.coows;

/**
 * Define the interface for the remote events
 * (a sort of wrapper)
 */
public class CooEventRemote {

    /**
     *
     * @uml.property name="mEvent"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private ICooEvent mEvent;

    private String mSourceHost;

/** Constructor */
    public CooEventRemote()    {
    }

/** Constructor */
public CooEventRemote(ICooEvent anEvent, String anHost) {
mEvent = anEvent;
        mSourceHost = anHost;
}

    /**
     * Modifier
     *
     * @uml.property name="mEvent"
     */
    public ICooEvent getMEvent() {
        return mEvent;
    }

    /**
     * Modifier
     *
     * @uml.property name="mEvent"
     */
    public void setMEvent(ICooEvent event) {
        mEvent = event;
    }


    /** Modifier */
public String getMAgentTO() {
return mEvent.getMAgentTO();
}

    /**
```

```
     *
     * @uml.property name="mSourceHost"
     */
    public String getMSourceHost() {
        return mSourceHost;
    }

    /**
     *
     * @uml.property name="mSourceHost"
     */
    public void setMSourceHost(String sourceHost) {
        mSourceHost = sourceHost;
    }

}
```

*org.coows.CooEventRequested*

```
package org.coows;

import org.coows.util.CooRequestID;

/**
 * Wrapper for data used by requested op
 */
public class CooEventRequested
extends CooAbstractEvent {

/**
 * The requestor ID
 */
private String mAgentFROM;

    /**
     * The request ID
     *
     * @uml.property name="mReqID"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private CooRequestID mReqID;

    /**
     * The desire
     *
     * @uml.property name="mDes"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private ICooDesire mDes;


/**
 * Constructor
 *
 */
public CooEventRequested() {
mAgentTO = "";
mAgentFROM = "";
mReqID = null;
mDes = new CooDesire();
}
```

```
/**
 * Constructor
 *
 * @param anAgentID
 * @param aDes
 */
public CooEventRequested(String anAgentFROM, String anAgentTO,
CooRequestID aReqID, ICooDesire aDes) {
mAgentFROM = anAgentFROM;
mAgentTO = anAgentTO;
mReqID = aReqID;
mDes = aDes;
}

    /**
     *
     * @uml.property name="mAgentFROM"
     */
    public String getMAgentFROM() {
        return mAgentFROM;
    }

    /**
     *
     * @uml.property name="mAgentFROM"
     */
    public void setMAgentFROM(String agentID) {
        mAgentFROM = agentID;
    }

    /**
     *
     * @uml.property name="mDes"
     */
    public ICooDesire getMDes() {
        return mDes;
    }



public void setMDes(CooDesire des) {
mDes = des;
}

    /**
     *
     * @uml.property name="mReqID"
     */
    public CooRequestID getMReqID() {
        return mReqID;
    }

    /**
     *
     * @uml.property name="mReqID"
     */
    public void setMReqID(CooRequestID reqID) {
        mReqID = reqID;
    }
```

```
public String toString() {
return "EventRequested("+mAgentFROM+","
            +mAgentTO+","+mReqID+","+mDes+")";
}


/**
 * @see org.coows.ICooEvent#dd(org.coows.CooAgent)
 */
public void dd(CooAgent anAgent) {
anAgent.onEventRequested(this);
}
}
```

*org.coows.CooPlan*

```
package org.coows;

/**
 * A CooWS plan is defined by the tuple
 * (plan ID, trigger, body, access specifier)
 *
 */
public class CooPlan {
/**
 * The unique plan identifier
 */
private String mPlanID;

/**
 * The trigger of the plan
 *
 * @uml.property name="mTrigger"
 * @uml.associationEnd multiplicity="(0 1)"
 */
private ICooDesire mTrigger;

/**
 *
 * The name of the service provided by the bpel process that implements the
 * body of the plan
 */
private String mBodyID;

/**
 * The access specifier for the plan
 */
private ICooAccessSpecifier mAccessSpecifier;


/**
 * Constructor
 *
 */
public CooPlan() {
}


/**
 * Constructor
 *
```

```
 * @param plandID
 * @param trigger
 * @param bpelPlan
 * @param accessSpecifier
 */
public CooPlan(String aPlanID, ICooDesire aTrigger, String aServiceName,
ICooAccessSpecifier aAccessSpecifier) {
mPlanID = aPlanID;
mTrigger = aTrigger;
mBodyID = aServiceName;
mAccessSpecifier = aAccessSpecifier;
}


/**
 *
 * @see java.lang.Object#hashCode()
 */
public int hashCode() {
return mPlanID.hashCode();
}


public boolean equals(Object o) {
        if (o instanceof CooPlan)
            return mPlanID.equals(((CooPlan)o).getMPlanID());
        else
            return false;
}


public String toString() {
String res = "(PLANID: ";
res += this.mPlanID;
res += (", TRIGGER=" + mTrigger);
res += (", BODY=" + mBodyID);
res += (", ACCSPEC=" + mAccessSpecifier);
res += ")";

return res;
}


/**
 *
 * @uml.property name="mPlandID"
 */
public String getMPlanID() {
return mPlanID;
}


/**
 *
 * @uml.property name="mPlandID"
 */
public void setMPlanID(String planID) {
mPlanID = planID;
}


/**
```

```
 *
 * @uml.property name="mTrigger"
 */
public ICooDesire getMTrigger() {
return mTrigger;
}


/**
 *
 * @uml.property name="mTrigger"
 */
public void setMTrigger(ICooDesire trigger) {
mTrigger = trigger;
}


public ICooAccessSpecifier getMAccessSpecifier() {
return mAccessSpecifier;
}


public void setMAccessSpecifier(
            ICooAccessSpecifier accessSpecifier) {
mAccessSpecifier = accessSpecifier;
}


public String getMServiceName() {
return mBodyID;
}


public void setMServiceName(String serviceName) {
mBodyID = serviceName;
}
}
```

***org.coows.ICooDesire***

```
package org.coows;

/**
 * Interface for Desire implementations
 */
public interface ICooDesire {

/** Modifier for the mDes field */
    //public String getMDes();

    /** Modifier for the mDes field */
//public void setMDes(String aDes);


/**
 * Test the match between two desire objects
 * (usually a plan trigger and a desire)
 */
public boolean match(ICooDesire aDes);
}
```

***org.coows.ICooEvent***

```
package org.coows;

/**
 * Interface for Desire implementations
 */
public interface ICooDesire {

/** Modifier for the mDes field */
    //public String getMDes();

    /** Modifier for the mDes field */
//public void setMDes(String aDes);


/**
 * Test the match between two desire objects
 * (usually a plan trigger and a desire)
 */
public boolean match(ICooDesire aDes);
}
```

## B.2 Package org.coows.engine

*org.coows.engine.CooAgentsDirectory*

```
package org.coows.engine;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.ser.BeanDeserializerFactory;
import org.apache.axis.encoding.ser.BeanSerializerFactory;

import org.coows.CooAccessSpecifierOnlyTrusted;
import org.coows.CooAccessSpecifierPrivate;
import org.coows.CooAccessSpecifierPublic;
import org.coows.CooBpelProcessBpr;
import org.coows.CooDesire;
import org.coows.CooEventProvided;
import org.coows.CooEventRequested;
import org.coows.CooPlan;
import org.coows.CooEventRemote;
import org.coows.ICooEvent;

import org.coows.ser.CooBprFileDeserializerFactory;
import org.coows.ser.CooBrpFileSerializerFactory;

import org.coows.util.CooBprFile;
import org.coows.util.CooSetOfPlans;

import org.uddi4j.UDDIException;

import org.uddi4j.client.UDDIProxy;

import org.uddi4j.datatype.Name;
import org.uddi4j.datatype.business.BusinessEntity;

import org.uddi4j.response.AuthToken;
```

```
import org.uddi4j.response.BusinessDetail;
import org.uddi4j.response.BusinessInfo;
import org.uddi4j.response.BusinessList;

import org.uddi4j.transport.TransportException;
import org.uddi4j.transport.TransportFactory;

import org.uddi4j.util.DiscoveryURL;
import org.uddi4j.util.DiscoveryURLs;
import org.uddi4j.util.FindQualifier;
import org.uddi4j.util.FindQualifiers;

import java.net.InetAddress;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.UnknownHostException;

import java.util.Hashtable;
import java.util.Iterator;
import java.util.Set;
import java.util.Vector;

import javax.xml.namespace.QName;
import javax.xml.rpc.ParameterMode;
import javax.xml.rpc.ServiceException;

/**
 * Interface the CooWS engine with the UDDI agents repository
 */
public class CooAgentsDirectory {
    /**
     * Logger for this class
     */
    private static final Log logger =
        LogFactory.getLog(CooAgentsDirectory.class);

    private final static QName REMOTEEVENT_QNAME =
        new QName("urn:coows", "RemoteEvent");

    /**
     * A cache for the UDDI repository
     *
     */
    private static Hashtable<String,String> mUDDIRepositoryCache;

    /**
     * The local hostname
     */
    private final String mHostname;

    /**
     * The UDDI Proxy (see the juddi doc)
     */
    private UDDIProxy mUDDIProxy;

    /**
     * The Auth token (see the juddi doc)
     */
    private AuthToken mAuthToken;

    /**
     * Default constructor
```

```
 */
public CooAgentsDirectory(CooEngineConfiguration aConfig)
        throws CooException {
    try {
        mHostname = InetAddress.getLocalHost().getHostName();
    }
    catch (UnknownHostException e) {
        throw new CooException("[CooAgentsDirectory] init failed! "
                + "Unable to resolv hostname!");
    }

    initUDDI(aConfig);
    mUDDIRepositoryCache = new Hashtable<String,String>();

    if (logger.isDebugEnabled()) {
        logger.debug("CooAgentsDirectory() - Agents Directory Loaded!");
    }
}

/**
 * Find an agent location through the UDDI registry
 *
 * @param anAgentID
 *            the agentID
 *
 * @return the host where the agent is hosted
 *
 * @throws TransportException
 * @throws UDDIException
 */
private String getAgentLocation(String anAgentID)
        throws UDDIException, TransportException {
    String agentHostedBy;

    if (mUDDIRepositoryCache.containsKey(anAgentID)) {
        // retrieve the agent info from the local cache
        agentHostedBy = mUDDIRepositoryCache.get(anAgentID);
    }
    else {
        // Retrieve the agent info from the remote UDDI registry
        // creating vector of Name Object
        Vector<Name> names = new Vector<Name>();
        names.add(new Name(anAgentID));

        // Setting FindQualifiers to 'exactNameMatch'
        FindQualifiers findQualifiers = new FindQualifiers();
        Vector qualifier = new Vector();
        qualifier.add(new FindQualifier("exactNameMatch"));
        findQualifiers.setFindQualifierVector(qualifier);

        // Find the agent location
        BusinessList businessList = mUDDIProxy.find_business(names, null,
                null, null, null, findQualifiers, 2);

        Vector businessInfoVector =
            businessList.getBusinessInfos().getBusinessInfoVector();

        if (businessInfoVector.size() > 1) {
            logger.warn("agent " + anAgentID +
                    " is published twice!", null);
        }
```

```
            // Retrieve the agent's info
            BusinessInfo bi = (BusinessInfo) businessInfoVector.elementAt(0);
            BusinessDetail bd =
                mUDDIProxy.get_businessDetail(bi.getBusinessKey());
            BusinessEntity be = (BusinessEntity)
                bd.getBusinessEntityVector().elementAt(0);
            agentHostedBy = be.getDiscoveryURLs().get(0).getText();

            // Update the local cache for future lookups
            mUDDIRepositoryCache.put(anAgentID, agentHostedBy);
        }

        if (logger.isDebugEnabled()) {
            logger.debug("getAgentLocation() - retrieved agent's location : "
                    + "agentHostedBy = " + agentHostedBy);
        }

        return agentHostedBy;

    }

    /**
     * Publish the local agents to the UDDI registry
     *
     * @param config
     * @param enumeration
     *
     * @throws CooException
     */
    public void publishLocalAgents(Set<String> agents)
            throws CooException {
        publishAgents2UDDI(agents);
    }

    /**
     * Send an event to an agent
     *
     * @param mAgents
     * @param anEvent
     */
    public void sendEvent(ICooEvent anEvent) {
        if (logger.isDebugEnabled()) {
            logger.debug("sendEvent() - Dispatching event: " + anEvent);
        }

        Dispatcher d = new Dispatcher(anEvent);
    }

    /**
     * Send the same event to different agents
     *
     * @param anEvent
     * @param dests
     */
    public void sendEvent(CooEventRequested anEvent, Vector dests) {
        if (logger.isDebugEnabled()) {
            logger.debug("sendEvent() - Dispatching event " + anEvent
                    + " to agents " + dests);
        }

        Dispatcher d = new Dispatcher(anEvent, dests);
    }
```

```
/**
 * Unregister agents from the UDDI server
 *
 * @param agents
 */
public void unregisterAgentsFromUDDI(Set agents) {
// TODO
}

/**
 * Shutdown the Agents Directory
 *
 */
public void shutdown() {
    unregisterAgentsFromUDDI(CooEngine.getMAgentsManager().getAgentsIDs());
}

/**
 * Connect to the UDDI server and get an auth token
 *
 * @throws CooException
 */
private void initUDDI(CooEngineConfiguration aConfig)
        throws CooException {
    try {
        String aUDDInquiryURL = aConfig
                .getText("/config/agentsDirConf/uddi-registry/inquiry-url");
        String aUDDIpublishURL = aConfig
                .getText("/config/agentsDirConf/uddi-registry/publish-url");

        System.setProperty(TransportFactory.PROPERTY_NAME,
                "org.uddi4j.transport.ApacheAxisTransport");
        System.setProperty("org.uddi4j.logEnabled", "false");
        System.setProperty("java.protocol.handler.pkgs",
                "com.sun.net.ssl.internal.www.protocol");

        java.security.Security.addProvider((java.security.Provider) Class
                .forName("com.sun.net.ssl.internal.ssl.Provider")
                .newInstance());

        mUDDIProxy = new UDDIProxy();
        mUDDIProxy.setInquiryURL(aUDDInquiryURL);
        mUDDIProxy.setPublishURL(aUDDIpublishURL);

        mAuthToken = mUDDIProxy.get_authToken(
          aConfig.getText("/config/agentsDirConf/uddi-registry/userid"),
          aConfig.getText("/config/agentsDirConf/uddi-registry/password"));
    }
    catch (MalformedURLException e) {
        throw new CooException("[CooAgentsDirectory] init failed! "
                + "UDDI server URL malformed! check the config file!");
    }
    catch (UDDIException e) {
        throw new CooException("[CooAgentsDirectory] init failed! "
                + "Unable to get an authorization token from "
                + "the remote UDDI registry " + e);
    }
    catch (TransportException e) {
        throw new CooException("[CooAgentsDirectory] init failed! "
                + "Unable to get an authorization token from "
                + "the remote UDDI registry " + e);
```

```
        }
        catch (ClassNotFoundException e) {
            throw new CooException("[CooAgentsDirectory] init failed! "
                    + "Error configuring JSSE provider. "
                    + "Make sure JSSE is in classpath! " + e);
        }
        catch (InstantiationException e) {
            throw new CooException("[CooAgentsDirectory] init failed! "
                    + "Error configuring JSSE provider! " + e);
        }
        catch (IllegalAccessException e) {
            throw new CooException("[CooAgentsDirectory] init failed! "
                    + "Error configuring JSSE provider! " + e);
        }
    }

    /**
     * Publish the local agents to the UDDI server
     *
     * @param agents
     * @throws CooException
     */
    private void publishAgents2UDDI(Set<String> agents)
            throws CooException {
        Vector<BusinessEntity> localAgents = new Vector<BusinessEntity>();
        BusinessEntity be;
        String agentID;
        DiscoveryURLs dus = new DiscoveryURLs();
        dus.add(new DiscoveryURL(mHostname, "hostedBy"));

        for (Iterator<String> i = agents.iterator(); i.hasNext();) {
            agentID = (String) i.next();
            be = new BusinessEntity("", agentID);
            be.setDiscoveryURLs(dus);
            localAgents.add(be);

            // Add the local agents to the UDDI repository cache
            mUDDIRepositoryCache.put(agentID, "localhost");
        }

        try {
            BusinessDetail bd = mUDDIProxy.save_business(mAuthToken
                    .getAuthInfoString(), localAgents);
            System.out.println("[CooAgentsDirectory] "
                    + "Local agents registered to UDDI registry!");
        }
        catch (UDDIException e) {
            throw new CooException(
                    "[CooAgentsDirectory] Error (UDDIException) "
                            + "publishing local agents to UDDI " + e);
        }
        catch (TransportException e) {
            throw new CooException(
                    "[CooAgentsDirectory] Error (TransportException) "
                            + "publishing local agents to UDDI " + e);
        }
    }

    /**
     * Keep the cache updated
     *
     * @param anAgentID
```

```
     * @param anHostname
     */
    public void updateCache(String anAgentID, String anHostname) {
        mUDDIRepositoryCache.put(anAgentID, anHostname);
    }

/**
 * A dispatcher (beta version, use the common-pool!)
 */
    class Dispatcher extends Thread {
        /**
         * Logger for this class
         */
        private final Log logger = LogFactory.getLog(Dispatcher.class);

        Vector<String> mAgents;

        /**
         *
         * @uml.property name="mEvent"
         * @uml.associationEnd multiplicity="(0 1)"
         */
        ICooEvent mEvent;

        public Dispatcher(ICooEvent e) {
            super(e.toString());
            mAgents = new Vector<String>();
            mAgents.add(e.getMAgentTO());
            mEvent = e;
            start();
        }

        public Dispatcher(ICooEvent e, Vector<String> agents) {
            super(e.toString());
            mAgents = agents;
            mEvent = e;
            start();
        }

        public void run() {

            String curr;

            while (!mAgents.isEmpty()) {
                curr = mAgents.remove(0);
                mEvent.setMAgentTO(curr);
                try {
                    String hostDest = getAgentLocation(curr);


                    // if (hostDest.equals("localhost"))
                    if (!hostDest.equals("localhost"))
                        // TODO RIATTIVARE GUARDIA CORRETTA!!!!
                        // CON QUESTA GUARDIA SPEDISCO
                        // TRAMITE WS AGLI
                        // AGENTI LOCALI
                        CooEngine.getMAgentsManager().dispatchEvent(mEvent);
                    else {
                        Call call = createCall(hostDest);

                        call.invoke(new Object[] {
                                new CooEventRemote(mEvent, mHostname) });
```

```
            }
        }
        catch (Exception e) {
            System.out.println("[CooAgentsDirectory]"
                    + "Error sending event to " + curr + ": "
                    + e.getMessage());
        }
    }
}

/**
 * Build the Call
 *
 * @param hostDest
 *
 * @return
 *
 * @throws ServiceException
 * @throws MalformedURLException
 */
private Call createCall(String hostDest)
        throws ServiceException, MalformedURLException {
    Call call = (Call) new Service().createCall();
    call.setTargetEndpointAddress(new URL("http://" + hostDest
            + ":4444/active-bpel/services/CooWS"));

    // Registering type mapping
    call.registerTypeMapping(CooDesire.class, new QName("urn:coows",
            "Desire"), BeanSerializerFactory.class,
            BeanDeserializerFactory.class);
    call.registerTypeMapping(CooPlan.class, new QName("urn:coows",
            "Plan"), BeanSerializerFactory.class,
            BeanDeserializerFactory.class);
    call.registerTypeMapping(CooAccessSpecifierPublic.class, new QName(
            "urn:coows", "AccessSpecifierPublic"),
            BeanSerializerFactory.class, BeanDeserializerFactory.class);
    call.registerTypeMapping(CooAccessSpecifierPrivate.class,
            new QName("urn:coows", "AccessSpecifierPrivate"),
            BeanSerializerFactory.class, BeanDeserializerFactory.class);
    call.registerTypeMapping(CooAccessSpecifierOnlyTrusted.class,
            new QName("urn:coows", "AccessSpecifierOnlyTrusted"),
            BeanSerializerFactory.class, BeanDeserializerFactory.class);
    call.registerTypeMapping(CooSetOfPlans.class, new QName(
            "urn:coows", "SetOfPlans"), BeanSerializerFactory.class,
            BeanDeserializerFactory.class);
    call.registerTypeMapping(CooBpelProcessBpr.class, new QName(
            "urn:coows", "BpelProcessBpr"),
            BeanSerializerFactory.class, BeanDeserializerFactory.class);
    call.registerTypeMapping(CooBprFile.class, new QName("urn:coows",
            "BprFile"), CooBrpFileSerializerFactory.class,
            CooBprFileDeserializerFactory.class);
    call.registerTypeMapping(ICooEvent.class, new QName("urn:coows",
            "Event"), BeanSerializerFactory.class,
            BeanDeserializerFactory.class);
    call.registerTypeMapping(CooEventRequested.class, new QName(
            "urn:coows", "EventRequested"),
            BeanSerializerFactory.class, BeanDeserializerFactory.class);
    call.registerTypeMapping(CooEventProvided.class, new QName(
            "urn:coows", "EventProvided"), BeanSerializerFactory.class,
            BeanDeserializerFactory.class);
    call.registerTypeMapping(CooEventRemote.class, new QName(
            "urn:coows", "RemoteEvent"), BeanSerializerFactory.class,
```

```
                    BeanDeserializerFactory.class);

            call.setOperationStyle("rpc");
            call.setOperationName(new QName("CooWS", "dispatchEvent"));
            call.addParameter(REMOTEEVENT_QNAME, REMOTEEVENT_QNAME,
                    CooEventRemote.class, ParameterMode.IN);

            return call;
        }
    }

}
```

*org.coows.engine.CooAgentsManager*

```
package org.coows.engine;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.coows.CooAgent;
import org.coows.CooEventProvided;
import org.coows.CooEventRemote;
import org.coows.CooEventRequested;
import org.coows.CooGlobalEventsQueue;
import org.coows.ICooEvent;

import java.util.Hashtable;
import java.util.Iterator;
import java.util.Set;

/**
 * The manager of local agents
 */
public class CooAgentsManager {
    /**
     * Logger for this class
     */
    private static final Log logger = LogFactory.getLog(CooAgentsManager.class);

    /**
     * A table holding the ref to local agents
     */
    private Hashtable<String,CooAgent> mAgents;

    /**
     * The queue where are stored the incoming events waiting to be dispatched
     *
     * @uml.property name="mGlobalEventsQueue"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private CooGlobalEventsQueue mGlobalEventsQueue;


    /**
     * Restore the agents with the last saved state and initialize all local
     * variables (thread pool and global event queue)
     *
     * @param aConf
     * @throws CooException
     */
```

```
public CooAgentsManager(CooEngineConfiguration aConf)
    throws CooException {
    if (logger.isDebugEnabled()) {
        logger.debug("CooAgentsManager() - "
                + "initializing Agents Manager structures");
    }

    // Restore the agents
    mAgents = CooPersistenceManager.restoreAgents(aConf);

    // Create the global events queue
    mGlobalEventsQueue = new CooGlobalEventsQueue();

    if (logger.isDebugEnabled()) {
        logger.debug("CooAgentsManager() - Agents Manager loaded!");
    }
}


/**
 * Start the agents execution
 *
 */
public void startAgents() {
    for (Iterator<CooAgent> i = mAgents.values().iterator();i.hasNext();) {
        i.next().start();
    }
}




/**
 * Handle incoming events <br>
 * A pool of thread dispatchers enqueue the new incoming envents to the
 * local event queue of each agent
 *
 * @param anEvent
 * @throws CooException
 */
public void dispatchEvent(ICooEvent anEvent) {
    if (logger.isDebugEnabled()) {
        logger.debug("dispatchEvent(anEvent = " + anEvent
                + ") - received new incoming event");
    }

    mGlobalEventsQueue.enqueue(anEvent);
}

/**
 * Dispatch an incoming remote event <br>
 * On event requested the agents directory cache is updated <br>
 * On event provided the processes manager is notified on the source host
 *
 * @param anEvent
 * @throws CooException
 */
public void dispatchRemoteEvent(CooEventRemote anEvent) {
    if (anEvent.getMEvent() instanceof CooEventRequested)
        CooEngine.getAgentsDirectory().updateCache(
                ((CooEventRequested) anEvent.getMEvent()).getMAgentFROM(),
                anEvent.getMSourceHost());
```

```
    else
        if (anEvent.getMEvent() instanceof CooEventProvided)
            CooEngine.getMProcessesManager().updateNoLocal(
                    ((CooEventProvided) anEvent.getMEvent())
                            .getMProvidedPlans(), anEvent.getMSourceHost());

    this.dispatchEvent(anEvent.getMEvent());
}

/**
 * Shutdown the agents manager<br>
 * Serialize the current agents states to the db and halt the agents
 * execution cicle
 *
 * @param aConf
 */
public void shutdown(CooEngineConfiguration aConf) {

    Iterator<CooAgent> curr;

    try {
        // Clear the global queue
        mGlobalEventsQueue.clear();

        // Notify the agents
        for (curr = mAgents.values().iterator();curr.hasNext();)
            curr.next().prepare4shutdown();


        // Serialize agents state to the db
        CooPersistenceManager.serializeAgents(aConf, mAgents.values());

        // Halt the agents execution
        for (curr = mAgents.values().iterator(); curr.hasNext();) {
            curr.next().shutdown();
        }

        if (logger.isDebugEnabled()) {
            logger.debug("shutdown() done!");
        }
    } catch (CooException e) {
        if (logger.isErrorEnabled()) {
            logger.error("shutdown() - Encountered an error while" +
                    "shuting down the Agents Manager!",e);
        }
    }

}


/**
 * getter
 * @param anAgentID
 * @return
 */
public CooAgent getAgent(String anAgentID) {
    return (CooAgent) mAgents.get(anAgentID);
}

public Set<String> getAgentsIDs() {
    return mAgents.keySet();
}
```

```
    /**
     * getter
     * @uml.property name="mGlobalEventsQueue"
     */
    public CooGlobalEventsQueue getMGlobalEventsQueue() {
        return mGlobalEventsQueue;
    }

}
```

*org.coows.engine.CooBpelProcessesManager*

```
package org.coows.engine;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.coows.CooBpelProcessParameters;
import org.coows.CooEventPlanOutcome;
import org.coows.CooPlan;
import org.coows.ICooBpelProcess;
import org.coows.util.CooSetOfPlans;

import java.net.InetAddress;
import java.net.UnknownHostException;

import java.util.Hashtable;
import java.util.Iterator;
import java.util.Vector;

/**
 * The Manager of the local BPEL Processes
 *
 */
public class CooBpelProcessesManager {
    /**
     * Logger for this class
     */
    private static final Log logger =
        LogFactory.getLog(CooBpelProcessesManager.class);

    /**
     * The archive of local BPEL processes The table maps for each serviceName
     * the corresponding BPEL process
     */
    private Hashtable<String,ICooBpelProcess> mProcesses;

    /**
     * This table holds the info necessary for the retrieval of remote processes
     * For each serviceName we have a list of hostnames wich coud supply the
     * BPEL process associated
     */
    private Hashtable<String,Vector<String>> mNoLocalProcesses;

    /**
     * The local hostname
     */
    private String mHostname;

    /**
```

```
 * List of serviceName currently deployed
 */
private Vector<String> mProcessesDeployed;

/**
 * Constructor
 *
 * @param aConf
 *            The engine configuration
 * @throws CooException
 */
public CooBpelProcessesManager(CooEngineConfiguration aConf)
        throws CooException {
    mProcesses = CooPersistenceManager.restoreBpelProcesses(aConf);
    mProcessesDeployed = new Vector<String>();
    mNoLocalProcesses = new Hashtable<String,Vector<String>>();

    try {
        // retrieve the hostname
        mHostname = InetAddress.getLocalHost().getHostName();
    }
    catch (UnknownHostException e) {
        mHostname = "localhost";
        System.out.println("[CooBpelProcessManager.init] WARNING: "
                + "unable to resolv hostname! Using localhost");
    }

    if (logger.isInfoEnabled()) {
        logger.info("CooBpelProcessesManager() - "
                + "BPEL Processes Manager Loaded!");
    }
}

/**
 * Invoke a process on demand
 *
 * @param anAgentID
 *            The agent requestor
 * @param aServiceName
 *            The service name
 */
public void invokeProcess(String anAgentID, String anIntentionID,
        String aServiceName) {

    if (logger.isDebugEnabled()) {
        logger.debug("runProcess(agentID = " + anAgentID + ", "
                + "serviceName = " + aServiceName + ")");
    }

    // If the process is not stored locally and doesn't appear
    // among the remote provided processes
    if (!mProcesses.containsKey(aServiceName)
            && !mNoLocalProcesses.containsKey(aServiceName)) {

        if (logger.isErrorEnabled()) {
            logger.error("runProcess() Error! " +
              "Trying to run unknown process!!! " + aServiceName, null);
        }

        return;
    }
```

```
    // If the process is stored on a remote node
    if (!mProcesses.containsKey(aServiceName)
            && mNoLocalProcesses.contains(aServiceName)) {
        // Retrieve the remote BPEL process
        String remoteHost =
            mNoLocalProcesses.get(aServiceName).lastElement();
        // TODO send to the remote host the request
    }

    ICooBpelProcess proc = (ICooBpelProcess) mProcesses.get(aServiceName);

    try {
        if (!mProcessesDeployed.contains(aServiceName)) {
            // Deploy the process
            proc.deploy();
            mProcessesDeployed.add(aServiceName);
        }

        // Invoke the process
        proc.invoke(
         new CooBpelProcessParameters(mHostname, anAgentID,anIntentionID));
    } catch (CooException e) {
        logger.error("runProcess(serviceName = " + aServiceName
                + ") - Failed to invoke process ", e);

        // Notify the agent requester
        CooEngine.getMAgentsManager().dispatchEvent(
                new CooEventPlanOutcome(anAgentID,anIntentionID,false));
    }
}

/**
 *
 * @param aServiceName
 * @return
 * @throws CooException
 */
public ICooBpelProcess getProcess(String aServiceName)
        throws CooException {
    if (mProcesses.containsKey(aServiceName))
        return (ICooBpelProcess) mProcesses.get(aServiceName);
    else
        throw new CooException("Process " + aServiceName + "not found!");
}

/**
 * Keep trace of the host where could be retrieved the BPEL processes<br>
 * of the plans retrieved remotely
 *
 * @param aSop
 * @param anHostname
 */
public void updateNoLocal(CooSetOfPlans aSop, String anHostname) {
    CooPlan curr;
    Vector<String> aux;

    for (Iterator<CooPlan> i = aSop.getMPlans().iterator(); i.hasNext();) {
        curr = i.next();

        if (mProcesses.containsKey(curr.getMPlanID()))
            continue;
        else {
```

```
                aux = (Vector<String>) mNoLocalProcesses.get(curr.getMPlanID());
                if (null == aux) aux = new Vector<String>();
                aux.add(anHostname);

                mNoLocalProcesses.put(curr.getMPlanID(), aux);
            }
        }
    }

    /**
     * Shutdown the Manager <br>
     * (undeploys all the processes and stores them back to the db)
     *
     * @param aConf
     * @throws CooException
     */
    public void shutdown(CooEngineConfiguration aConf)
            throws CooException {
        // TODO undeployAll();
        CooPersistenceManager.serializeProcesses(aConf, mProcesses.values());

        if (logger.isDebugEnabled()) {
            logger.debug("shutdown() done!");
        }
    }

    /**
     * Dispatch an <b>achieved</b> message to a suspended process
     * @param planID
     */
    public void dispatchAchieved(String anAgentID, String anIntentionID,
            String aServiceName) {

    }
}
```

*org.coows.engine.CooEngine*

```
package org.coows.engine;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * Singleton class
 */
public class CooEngine {
    /**
     * Logger for this class
     */
    private static final Log logger = LogFactory.getLog(CooEngine.class);


public static final String CATALINA_HOME =
        System.getProperties().getProperty("catalina.home");

public static final String DEFAULT_CONFIG_DIR = CATALINA_HOME + "/conf";

private static final String DEFAULT_CONFIG_FILE = "cooEngineConf.xml";

    /**
```

```
     * The in memory representation of the engine configuration
     *
     * @uml.property name="mConfig"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private static CooEngineConfiguration mConfig;

    /**
     * The agents manager
     *
     * @uml.property name="mAgentsManager"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private static CooAgentsManager mAgentsManager;

    /**
     * The agents directory
     *
     * @uml.property name="mAgentsDirectory"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private static CooAgentsDirectory mAgentsDirectory;

    /**
     * The manager of the bpel processes
     *
     * @uml.property name="mProcessesManager"
     * @uml.associationEnd multiplicity="(0 1)"
     */
    private static CooBpelProcessesManager mProcessesManager;


public static CooAgentsDirectory getAgentsDirectory() {
return mAgentsDirectory;
}

    /**
     *
     * @uml.property name="mAgentsManager"
     */
    public static CooAgentsManager getMAgentsManager() {
        return mAgentsManager;
    }

    /**
     *
     * @uml.property name="mConfig"
     */
    public static CooEngineConfiguration getMConfig() {
        return mConfig;
    }


public static void init() {
init(DEFAULT_CONFIG_FILE, DEFAULT_CONFIG_DIR);
}


    // Convenience method for debug purpose
public static void initTEMP(String configFile, String configDir) {
```

```
        try {
mConfig = new CooEngineConfiguration(configFile, configDir);
} catch (Exception e) {
}
}


    /**
     * Initialize all the CooWS modules and start the execution
     *
     * @param configFile
     * @param configDir
     */
public static void init(String configFile, String configDir) {
        if (logger.isDebugEnabled())
        {
            logger.debug("init() - Initializing CooWS engine components");
        }

try {
            // Load the configuration file
mConfig = new CooEngineConfiguration(configFile, configDir);

// PopulateDB.populateProcessTable(mConfig);

            // Initialize the BPEL Processes Manager
mProcessesManager = new CooBpelProcessesManager(mConfig);

            // Initialize the Agents Manager
mAgentsManager = new CooAgentsManager(mConfig);

// CooPersistenceManager.restoreAgents(mConfig);

            // Initialize the Agents Directory Manager
mAgentsDirectory = new CooAgentsDirectory(mConfig);

            // Update the directory with the local agents
mAgentsDirectory.publishLocalAgents(mAgentsManager.getAgentsIDs());

            // Start the agents execution
mAgentsManager.startAgents();

            if (logger.isDebugEnabled()) {
                logger.debug("init() - CooWS engine started!");
            }
} catch (Exception e) {
            if (logger.isErrorEnabled())
            {
                logger.error("init() - Failed to start the CooWS engine!", e);
            }

}
}


    /**
     * Shutdown the agents
     * @throws CooException
     */
public static void shutdown() throws CooException {
        if (logger.isDebugEnabled())
        {
```

```
            logger.debug("shutdown() - Halting the CooWS engine");
        }


mAgentsDirectory.shutdown();
mAgentsManager.shutdown(mConfig);
mProcessesManager.shutdown(mConfig);


        if (logger.isDebugEnabled())
        {
            logger.debug("shutdown() - " +
                    "CooWS engine has been successfully shutdown!");
        }
}

    /**
     *
     * @uml.property name="mProcessesManager"
     */
    public static CooBpelProcessesManager getMProcessesManager() {
        return mProcessesManager;
    }

}
```

*org.coows.engine.CooEngineConfiguration*

```
package org.coows.engine;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.jaxen.JaxenException;
import org.jaxen.XPath;

import org.jaxen.dom.DOMXPath;

import org.w3c.dom.Document;
import org.w3c.dom.Node;

import java.io.File;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

/**
 *
 */
public class CooEngineConfiguration {
    /**
     * Logger for this class
     */
    private static final Log logger = LogFactory
            .getLog(CooEngineConfiguration.class);

    private Document doc;

private File configFile;
```

```
/**
 * @param configFile
 * @param configDir
 *
 * @throws CooException
 */
public CooEngineConfiguration(String configFileName, String configDir)
throws CooException {
try {
DocumentBuilder builder = DocumentBuilderFactory.newInstance()
.newDocumentBuilder();
configFile = new File(configDir, configFileName);
doc = builder.parse(configFile);

            if (logger.isDebugEnabled()) {
                logger.debug("CooEngineConfiguration() - " +
                        "CooWS engine configuration loaded!");
            }

} catch (Exception e) {
throw new CooException("[CooEngineConfiguration.CONSTRUCTOR]"
+ " Error reading config file: " + e.getMessage());
}
}


/**
 * Given a lookup <var>xpath</var> and an attribute name, returns the value
 * of that attribute.
 *
 * @param xpath
 *            an XPath expression
 * @param attrName
 *            an attribute name
 *
 * @return the value of the text inside the specified element
 */
public String getAttr(String xpath, String attrName) throws CooException {
try {
return get(xpath + "/@" + attrName);
} catch (Exception e) {
throw new CooException(
"[CooEngine ERROR] unable to get the value of the value "
+ xpath + "/@" + attrName + " from config file");
}
}


/**
 * Given a lookup <var>xpath</var>, returns the value of the text inside
 * the specified element.
 *
 * @param xpath
 *            an XPath expression
 *
 * @return the value of the text inside the specified element
 */
public String getText(String xpath) throws CooException {
try {
return get(xpath + "/text()[1]");
} catch (Exception e) {
throw new CooException("[CooEngine ERROR] unable to get the value "
```

```
+ xpath + " from config file");
}
}


/**
 * Given an <var>xpath</var>, returns the value of that element.
 *
 * @param xpath
 *            an XPath expression
 *
 * @return the value
 */
public String get(String xpath) throws CooException {
XPath xp;

try {
xp = new DOMXPath(xpath);

Node node = (Node) xp.selectSingleNode(doc);

return node.getNodeValue().trim();
} catch (JaxenException e) {
throw new CooException("[CooEngine ERROR] unable to get the value "
+ xpath + " from config file");
}
}
}
```

# B.3 Package org.coows.ws

***org.coows.ws.CooWS***

```
package org.coows.ws;

import org.coows.CooEventAchieve;
import org.coows.CooEventPlanOutcome;
import org.coows.CooEventRemote;
import org.coows.ICooBpelProcess;
import org.coows.ICooEvent;

import org.coows.engine.CooEngine;
import org.coows.engine.CooException;

/**
 * The CooWS Web Service
 */
public class CooWS {
    public void dispatchEventRemote(CooEventRemote aREvent) {
        dispatchEvent(aREvent.getMEvent());
    }

    private void dispatchEvent(ICooEvent anEvent) {
            CooEngine.getMAgentsManager().dispatchEvent(anEvent);
    }

    public void achieve(String anAgent, String anIntentionID, String aDes) {
        dispatchEvent(new CooEventAchieve(anAgent, anIntentionID, aDes));
    }
```

```
    public void planOutcome(String anAgent,
            String intentionID, boolean outcome) {
        dispatchEvent(new CooEventPlanOutcome(anAgent, intentionID, outcome));
    }

    public ICooBpelProcess getBPELProcess(String aServiceName)
            throws CooException {
        return CooEngine.getMProcessesManager().getProcess(aServiceName);
    }
}
```

# Appendice C

# Glossario

**B2B** - Business To Business. Interazione tra due soggetti di natura *commerciale*. Tipicamente avviene tramite Internet e senza intervento umano. Si contrappone al B2C.

**B2C** - Business To Consumer. Interazione tra una persona (l'utente) e un'azienda tramite Internet. Tipicamente l'utente fruisce dei servizi dell'azienda tramite un Web browser.

**CORBA** - Common Object Request Broker Architecture. Modello ad oggetti distribuito indipendente dal linguaggio di programmzione definito dall'OMG.

**BPEL4WS / BPEL** - Business Process Execution Language for Web Services. Linguaggio basato su XML per l'orchestrazione di Web Service.

**deploy** - Processo di installazione di un determinato componente software all'interno di un ambiente di esecuzione.

**deserializzazione** - Processo di ricostruzione di una struttura dati a partire da una sua rappresentazione XML.

**DOM** - Document Object Model. Fornisce una rappresentazione Object Oriented di un documento XML. La gerarchia del documento rappresentato si riflette nella gerarchia dell'oggetto.

**EAI** - Enterprise Application Integration. Una forma di computing distribuito dove applicazioni e processi di business, interni ed esterni ad una azienda, vengono integrati in un unico processo di più alto livello.

**EJB** - Architettura a componenti per lo sviluppo di applicazioni secondo il modello object-oriented, distribuite ed a livello aziendale. Le applicazioni sviluppate su architettura EJB possono godere di vantaggia quali transazioni, sicurezza dei dati e scalabilità in maniera trasparente.

**HTTP** - Hypertex Transfer Protocol. Il procollo Internet utilizzato per scambiare documenti ipertestuali tra macchine remote. I messaggi HTTP trasportano richieste dai client ai server e, viceversa, risposte dai server ai client.

**inquiry API** - Insieme di operazioni pubbliche non autenticate per l'interrogazione di un registry UDDI.

**IDL** - Interface Definition Language. Linguaggio per descrivere le interfacce degli oggetti coinvolti in ambiente distribuito. Tramite questa descrizione i compilatori generano proxy e stub per il marshaling automatico dei parametri. Nel contesto dei Web Service tale ruolo é svolto da WSDL.

**LAN** - Local Area Network. Rete di computer locale. Un esempio di lan è una intranet aziendale, un rete domestica o, molto più semplicemente, due macchine connesse con un cavo di rete.

**MOM** - Message Oriented Middleware. É una tipologia di framework orientata allo scambio di messaggi. Una implementazione é ad esempio Java Message Service (JMS)

**orchestrazione** - Composizione di due o più Web service. Il risultato é ancora un Web service. BPEL é un linguaggio concepito per l'orchestrazione di Web service.

**RPC** - Remote Procedure Call. Chiamata a funzione su macchina o processo differente.

**serializzazione** - Processo di creazione di un documento XML a partire da una struttura dati.

**SOA** - Service Oriented Architecture. Modello astratto costituente un architettura basata su tre ruoli ben definiti: Service Provider, Service Registry e Service Requestor. La SOA definisce inoltre il contratto tra questi tre ruoli mediante tre operazioni: publish, find e bind.

**service provider** - É uno dei ruoli specificati all'interno della SOA. É una qualsiasi entità che ospita uno o più Web service che potranno essere invocati da un service requestor.

**service registry** - É uno dei ruoli specificati all'interno della SOA. Implementa un servizio di pubblicazione e ricerca di Web service. Viene utilizzato dai Service Provider e dai Service Requestor.

**service requestor** - É uno dei ruoli specificati all'interno della SOA. É un service requestor chiunque invochi un Web service fornito da un service provider.

**SOAP** - Simple Object Access Protocol. Procollo leggero per lo scambio di messaggi in un ambiente distribuito e decentralizzato. É un protocollo basato su XML.

**UDDI** - Universal Description, Discovery and Integration. Implementazione di un registry per Web services. Fornisce operazioni quali find, publish e bind.

**WSA** - Web Service Architecture. Implementazione della Service Oriented Architecture realizzata con standard basati XML quali SOAP, WSDL, UDDI e BPEL.

**WSDL** - Web Service Description Language. Linguaggio basato su XML per la descrizione delle interfacce dei Web service.

**XML** - eXtensible Markup Language. Linguaggio di demarcazione che consente la definizione di tag (machio) ad-hoc per l'identificazione del contenuto, dei dati e del testo di un documento. Differisce quindi da linguaggi per il markup statici quali ad esempio HTML, il linguaggio utilizzato per la definizione della struttura grafica delle pagine Web.

# Bibliografia


[1] M. Winikoff. AgentTalk Home Page. `http://goanna.cs.rmit.edu.au/~winikoff/agenttalk`, 2001.

[2] M. d'Inverno, K. V. Hindriks, and M. Luck. A formal architecture for the 3APL agent programming language. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *Proc. of the 1st International ZB Conference*, pages 168–187. Springer Verlag, 2000. LNCS 1878.

[3] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Procedings of the Seventh European workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 42–55. Springer-Verlag, 1996. LNAI 1038, URL: `http://www.aaii.oz.au/bios/rao.html`.

[4] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. JACK intelligent agents – components for intelligent agents in Java. *AgentLink News Letter*, 2, 1999.

[5] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.

[6] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. Rao, and M. Wooldridge, editors, *Proc. of the 4th International ATAL Workshop*, pages 155–176. Springer Verlag, 1997. LNAI 1365.

[7] D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI model with cooperativity. In J. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *"Declarative Agent Languages and Technologies", First International Workshop, DALT 2003*, pages 109–134. Springer Verlag, 2003.

[8] D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI Model with Cooperativity. In J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Proceedings of the First Declarative Agent Languages and Technologies*

*Workshop (DALT'03), Revised Selected and Invited Papers*, pages 109–134. Springer-Verlag, 2004. LNAI 2990.

[9] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, 1986.

[10] Carolyn J. Hursch and Jack L. Hursch. *SQL, the structured query language.* Tab Books, 1988.

[11] HTTP – Hypertext Transfer Protocol. `http://www.w3.org/Protocols/`.

[12] TCP/IP – Transmission Control Protocol/Internet Protocol. See a definition from `http://www.webopedia.com/TERM/T/TCP_IP.html`.

[13] Sun Java 2 Enterprise Edition. `http://java.sun.com/j2ee/index.jsp`.

[14] Extensible Markup Language, May 2001. `http://www.w3.org/XML/`.

[15] Charlie Kindel Nat Brown. Distributed component object model — dcom/1.0. 1998.

[16] The Open Group. Dce rpc specification. `http://www.opengroup.org/onlinepubs/9629399`.

[17] Matija Marolt. CORBA - A framework for development of distributed applications, November 08 1996.

[18] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification. Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.

[19] Sun Microsystems. Java Remote Method Invocation - Distributed Computing for Java. `http://www.javasoft.com/marketing/collateral/javarmi.html`, March 1998. White Paper.

[20] Mom: Message oriented middleware. `http://en.wikipedia.org/wiki/Message_Oriented_Middleware`.

[21] P. Prescod. Lista di discussione su xmlp del w3c, 2002. `http://lists.w3.org/Archives/Public/xml-dist-app/2002Jun/0038.html`.

[22] H. Haas and A. Brown. Web Service Glossary, February 2004. `http://www.w3.org/TR/ws-gloss/`.

[23] Webopedia. `http://www.webopedia.com/`.

[24] Web Services Architecture Requirements, April 2002. `http://www.w3.org/TR/wsa-reqs`.

[25] SOA – Service-Oriented Architecture. See a definition from http://www.webopedia.com/TERM/S/Service_Oriented_Architecture.html.

[26] XML – Extensible Markup Language. http://www.w3.org/XML/.

[27] SOAP – Simple Object Access Protocol. http://www.w3.org/TR/soap/.

[28] RPC – Remote Procedure Call. See a definition from http://www.cs.cf.ac.uk/Dave/C/node33.html.

[29] BPEL4WS – Business Process Execution Language for Web Services. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

[30] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[31] W3C – The WWW Consortium. http://www.w3.org/.

[32] Wikipedia. RSS feed. http://en.wikipedia.org/RSS_(file_format).

[33] M. N. Huhns. Agents as web services. *Internet Computing*, 6(4):93–95, 2002.

[34] M. Luck, P. McBurney, O. Shehory, S. Willmott, and the AgentLink Community. *Agent Technology: Computing as Interaction – A Roadmap for Agent-Based Computing*. AgentLink III, 2005.

[35] ActiveBPEL engine. http://www.activebpel.org.

[36] Twister BPEL engine. http://www.smartcomps.org/twister.

[37] The Oracle BPEL Process Manager. http://otn.oracle.com/bpel.

[38] Apache Axis. http://xml.apache.org/axis/index.html, February 2001.

[39] Karl Aberer and Zoran Despotovic. Managing trust in a peer-2-peer information system, 2001.

[40] The Microsoft Corporation. The Microsoft .NET platform, 2002. http://www.microsoft.com/net/.

[41] Apache Tomcat. http://jakarta.apache.org/tomcat/.

[42] Apache Sofware Foundation. http://apache.org/.

[43] Apache jUDDI. http://ws.apache.org/juddi/, July 2004.

[44] David Axmark and Michael Widenius. MySQL introduction. *Linux Journal*, 67, nov 1999.

[45] Uddi4j. http://sourceforge.net/projects/uddi4j.

[46] Jsp: Java server pages. http://java.sun.com/products/jsp/.

[47] Arnaud Le Hors, Joe Kesselman, Laurence Cable, Mike Champion, Netscape Communications Corporation, Software Ag, Staff Contact, and Tom Pixley. Document object model (DOM) level 2 specification. Technical report, September 23 1999.

[48] Apache Commons Pool. http://jakarta.apache.org/commons/pool/.

[49] T. Bernes-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). Internet request for comment RFC, Internet Engineering Task Force, dec 1994.

[50] P. A. Buhler and J. M. Vidal. Towards adaptive workflow enactment using multiagent systems. *Information Technology and Management*, 6:61–87, 2005.

[51] OWL-S – Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/.

[52] Anupriya Ankolekar, David L. Martin, Honglei Zeng, Jerry R. Hobbs, Katia Sycara, Mark Burstein, Massimo Paolucci, Ora Lassila, Sheila A. Mcilraith, Srini Narayanan, and Terry Payne. DAML-S: Semantic markup for web services, July 13 0.

[53] D. Martin, M. Burstein, G. Denker, J. Hobbs, L. Kagal, O. Lassila, D. McDermott, S. McIlraith, M. Paolucci, B. Parsia, T. Payne, M. Sabou, E. Sirin, M. Solanki, N. Srinivasan, and K. Sycara. Owl-based web service ontology, November 2004. http://www.daml.org/services/owl-s/.

[54] S. Bechhofre, F. van Harmelen, J. Hendler, I. Horrocks, D.L. McGuinness, P.F. Patel-Shneider, and L.A. Stein. Owl web ontology language reference, February 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.