ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica

ARCHITETTURA MULTI-AGENTE PER LA RISOLUZIONE DI PROBLEMI DI MEETING-SCHEDULING

Elaborata nel corso di: Sistemi Multi-Agente LS

Relazione di:
JIMMY CICCOLINI

ANNO ACCADEMICO 2009–2010

Indice

Capitolo 1

Problemi di Meeting-Scheduling

1.1 Definizioni

Una delle problematiche più ricorrenti nella attuale società moderna ed industrializzata è, senza dubbio, l'organizzazione di meeting ed appuntamenti a cui debbono partecipare più individui, i quali, ovviamente, riempiono il loro tempo in base ai loro impegni.
Coloro che organizzano i suddetti incontri si trovano spesso a scontrarsi con gli impegni personali dei singoli partecipanti e si assumono la responsabilità di stabilire la data di un meeting rispettando e soddisfacendo le richieste e le prefernze da parte dei partecipanti stessi.
In prima analisi, sembrerebbe un processo decisionale statico, ma, analizzando attentamente il problema si intuisce che il processo è dinamico e distribuito. Infatti, durante l'organizzazione di un evento un partecipante potrebbe essere invitato ad altri eventi e l'utente si trova, quindi, ad entrare in un albero decisionale in cui deve scegliere a quale evento partecipare in base alla sua importanza e rilevanza, nonchè comunicare l'impossibilità di partecipare ad altri eventi proposti.
Questa classe di problemi prende il nome di **meeting-scheduling problems** e riguarda più in generale l'organizzazione di eventi proposti e seguiti da alcune entità partecipanti. Il problema da risolvere consiste nell'organizzare il maggior numero di eventi massimizzando l'utile in base alle disponibilità degli utenti.

1.2 Approcci per la risoluzione

I problemi MS (Meeting-Sceduling) non possono essere, come abbiamo visto, risolti attraverso un processo centralizzato, in quanto ogni partecipante ha una propria autonomia e durante la sua esistenza viene chiamato a partecipare a molteplici eventi. Questo ci porta a definire i problemi di MS come processi distribuiti che si coordinano al fine di raggiungere obiettivi comuni e personali, a volte in contrasto tra loro. Inoltre, lo scenario in cui si cerca di risolvere questo problema di ottimizzazione non è statico, in quanto gli utenti possono aggiungere, rimuovere oppure decidere di sostituire un evento con un altro. In questo modo, durante l'algoritmo di risoluzione che porta all'ottimizzazione dello scenario organizzativo attuale, potrebbero inserirsi altri eventi che cambiano i vincoli e quindi la soluzione complessiva finale. Inizialmente i problemi di MS vennero considerati come problemi vincolati, ossia CSP (Constrain Sadisfaction Problem), considerando tali problemi centralizzati. Analizzando le considerazioni sopra effettuate si è giunti poi a risolvere i problemi di meeting-scheduling attraverso la metodologia offerta da DistVCSP, ossia Distribuited Valued CSP. Questa metodologia risolve il problema della distribuzione ma presenta ancora quello della dinamicità. Gli algoritmi di risoluzione per questo tipo di problemi sono applicabili per ambienti statici e non dinamici come nel nostro caso. Inoltre, queste metodologie non sono efficienti quando vi sono in gioco un numero elevato di variabili, e quindi sono poco scalabili. Per risolvere queste problematiche, dopo vari studi, si è arrivati a considerare come migliore strategia l'MSRAC (Meeting Scheduling with Renfoircement Arc Consistency). Questa metodologia presenta il vantaggio di non effettuare una ricerca esaustiva delle soluzioni in caso di alterazioni dell'ambiente durante la risoluzione dell'algoritmo, permette un controllo di consistenza degli archi distribuito in modo da rendere efficiente la ricerca.
Questo approccio ha portato a considerare gli agenti come base risolutiva dell'algoritmo capace di risolvere in modo ottimale il problema in quanto le caratteristiche principali di tali strutture sono la dinami-

cità e la capacità di comunicare con altri agenti distribuiti su di un ambiente comune.
Un ultima considerazione riguarda l'utilizzo del DynVCSP (Dynamic Valued CSP) per l'ottimizzazione della soluzione dal momento che vi sono preferenze di orari impartite dagli utenti. Questo formalismo fornisce infatti delle linee guida da applicare agli agenti durante le loro scelte effettuate in un ambiente dinamico.
Dopo aver formalizzato il problema attraverso l'ausilio della matematica vedremo come applicare il modello multi-agente per creare un algoritmo risolutore efficace ed efficiente.

1.3 Formalizzazione del problema

Le considerazioni apportate nel paragrafo precedente portano a formalizzare il problema dal punto di vista matematico come un DynVCSP. Vediamo, quindi, quali sono le grandezze in gioco e come valutare la soluzione migliore.
Innanzitutto possiamo distinguere i vincoli in due categorie: i vincoli *hard* e i vincoli *soft*. I primi sono veri e propri vincoli che non possono essere rilassati al fine di avere una soluzione non consistente. I secondi, invece, sono vincoli che possono essere rilassati per ottenere una soluzione migliore e ancora consistente. Esemplificando, un vincolo hard è rappresentato dall'impossibilità di un utente a partecipare ad un evento in una particolare data, mentre un vincolo soft riguarda il valore delle preferenze di orario espresse dagli utenti in merito ad un meeting.
Gli elementi in gioco in un Dynamic Valued CSP sono raccolte in una quintupla del tipo {X, D, C, S, φ}.

- X indica quali sono i meeting da schedulare all'istante t;
- D è l'insieme delle possibili date per ogni meeting;
- C è l'insieme di vincoli, divisi in hard e soft;
- S rappresenta la struttura di valutazione delle soluzioni, ed è quindi utilizzata per trovare la soluzione migliore;

- φ è la funzione che associa i vincoli ed i loro pesi (il peso è 1 per i vincoli hard e compreso tra 0 e 1 per quelli soft) al livello di soddisfacimento di ogni utente.

Un ulteriore accorgimento riguarda l'asseganzione di un peso per ogni meeting in modo da gestire le priorità degli incontri.
Risolvere, quindi, un problema di questo tipo significa trovare un assegnamento delle ore proposte ai diversi meeting ottimizzando le preferenze degli utenti che vi partecipano.

Capitolo 2

Risoluzione con il modello multi-agente

Dopo aver introdotto i problemi di MS vediamo come implementare un algoritmo risolutivo utilizzando un sistema Multi-Agente, analizzando la struttura, il protocollo comunicativo e l'algoritmo capaci di risolvere questa tipoplogia di problemi.

2.1 Il modello multi-agente

Il modello proposto utilizza due tipologie di agenti: un agente Interfaccia che serve a controllare il raggiungimento dell'obiettivo globale e ad informare gli utenti riguardo il risultato, ed una serie di agenti Utenti che rappresentano gli individui che partecipano ed organizzano le riunioni. Da quanto appena detto si evince che gli agenti Utenti possono essere divisi in due gruppi proponenti e partecipanti in base alla funzione che hanno in un determinato istante nell'ambito di un meeting.
L'obiettivo di ogni agente Utente è quello di schedulare (e quindi partecipare) tutti i meeting assegnando loro le date che portano ad un massimo valore di soddisfacimento dei vincoli soft, oltre che naturalmente a soddisfare a pieno i vincoli hard imposti. Inoltre, durante la fase decisionale un agente sceglie di accettare gli eventi che hanno maggior peso rinunciando ad altri di peso minore.
L'obiettivo finale dello scheduling è, invece, quello di assegnare gli

orari ai meeting massimizzando la somma dell'utilità dei singoli utenti. Questo può portare a volte a far scegliere ad un agente un assegnamento che non ottimizza al massimo la somma delle preferenze (o utilità locale) per permettere un miglioramento complessivo della funzione obiettivo.

2.2 Il processo di risoluzione

Analizziamo più in dettaglio le dinamiche di questo modello.
Un primo problema consiste nella modalità di assegnamento delle preferenze delle date, in quanto, diverse scale porterebbero a errati valori di utilità per i diversi eventi. Ovviamente non è comodo avere una scala comune di preferenze, altrimenti ogni nuovo agente Utente che si collega al sistema si deve sincronizzare con la scala adottata dal sistema stesso. Si è, quindi, deciso di ordinare le date in base alla propria scala di preferenze e di mostrare all'agente proponente dell'evento la lista di date secondo questo ordinamento.
Il processo di risoluzione si divide in due passi fondamentali: rafforzare la consistenza degli archi per eliminare le date non possibili per quell'evento, e interagire con gli altri utenti per trovare una data comune che massimizzi l'utilità. L'idea principale parte, quindi, dalla volontà di un agente di organizzare un evento. L'agente proponente filtra già le date in cui non è possibile organizzare l'evento a causa, ad esempio, di altri eventi più importanti già schedulati. Se vi sono date disponibili, il proponente invia ai partecipanti le possibili date da assegnare all'evento. Gli agenti partecipanti filtrano ulteriormente le date in base ai loro impegni ed eventi già organizzati e inviano al proponente le possibili date rimanenti. Questo, infine, sceglie una data e la comunica ai partecipanti che la confermeranno.
Ovviamente si possono presentare alcuni problemi; ad esempio, non vi sono date disponibili, oppure un altro evento è stato organizzato durante la conferma di un altro. Queste problematiche saranno affrontate analizzando l'algoritmo risolutore nei prossimi capitoli.

2.2.1 Protocollo di comunicazione

Prima di passare all'analisi dell'algoritmo apportiamo alcune considerazioni sui messaggi scambiati tra gli utenti.
Innanzitutto si vuole utilizzare il minor numero di messaggi possibili e contenenti solamente le informazioni indispensabili per motivi di prestazioni a livello di occupazione della banda e della sicurezza.
Un'ulteriore ipotesi è quella che i messaggi vengano letti nello stesso ordine in cui vengono ricevuti.
I messaggi scambiati sono i seguenti:

- Start: messagio inviato dall'agente Interfaccia all'agente proponente per dare inizio al processo di assegnamento;
- RedMeetCalendar: messaggio inviato dal proponente ai partecipanti in cui vi sono indicate le date disponibili da assegnare all'evento e il peso dell'evento stesso;
- Reply: messaggio inviato dai partecipanti al proponente in cui vengano inviate le date filtrate dai diversi partecipanti in base ai propri impegni;
- ReceiveProp: messaggio inviato dal proponente ai partecipanti per l'invio della data scelta per l'evento;
- MeetNotPossible: messaggio inviato dagli agenti per indicare l'impossibilità di partecipare ad un evento;
- MeetingOk: messaggio inviato dai partecipanti per confermare la data dell'evento;
- UpdateProp: messaggio inviato da un partecipante ad un proponente che a causa di sovrapposizioni è costretto a cancellare un evento e attraverso il quale propone altre date per rischedularlo.

2.2.2 Algoritmo di risoluzione

Di seguito riportiamo uno schema che illustra le dinamiche dell'algoritmo e scriveremo in pseudo-codice le istruzioni compiute dai vari agenti per risolvere un MS problem.

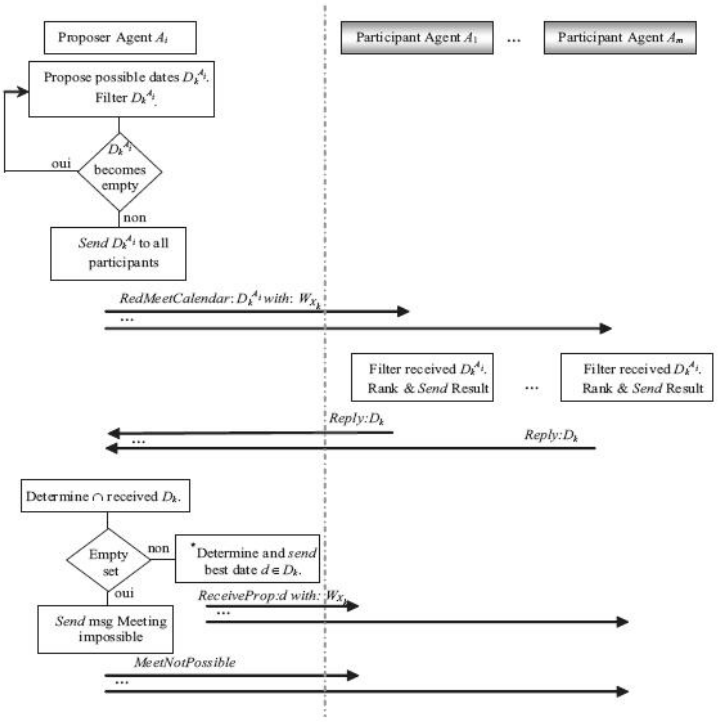

Figura 2.1: Dinamica di comunicazione tra proponente e partecipanti

Come detto precedentemente nel momento in cui un evento viene inserito nel sistema di pianificazione, l'agente Interfaccia invia un messaggio di start all'agente utente incaricato come proponente dell'evento.
A questo punto, come si vede dall'algoritmo 1 (figura 2.3), il proponente dell'evento applica un primo filtro eliminando le date che vìolano i suoi vincoli di tipo hard, applicando quindi un rafforzamento alla consistenza degli archi. L'agente rimuove ancora altre date se in queste sono già previsti eventi con un peso maggiore. Se non vi sono più date disponibili in questo processo il proponente cercherà di cambiare il range di date offerte dall'utente Interfaccia. Se, invece, vi sono date disponibili, queste vengono inviate a tutti gli agenti Utenti attraverso un messaggio di tipo RedMeetCalendar.

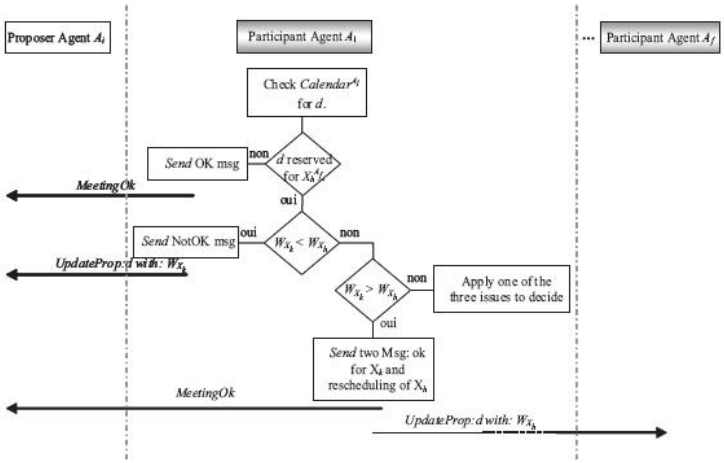

Figura 2.2: Proposta di un partecipante verso un proponente in caso di sovrapposizione di eventi

Algorithm 1. *Start* message executed by a User agent A_i for each meeting $X_k^{A_i}$.

Start

1: Delete from $D_k^{A_i}$ all non-viable values;
2: **if** $D_k^{A_i} = \emptyset$ **then**
3: Change meeting possible dates for $X_k^{A_i}$;
4: **else**
5: **for all** ($X_l^{A_j}$ such that $(X_l^{A_j}, dt_p) \in$ Calendar^{A_i}) AND ($W_{X_l} > W_{X_k}$) **do**
6: Delete($D_k^{A_i}$, dt_p);
7: Add($DReserve^{A_i}$[k], dt_p);
8: **if** $D_k^{A_i} = \emptyset$ **then**
9: Change meeting possible dates for $X_k^{A_i}$;
10: **else**
11: **forall** $A_j \in$ Part($X_k^{A_i}$) **do**
12: *Send*(*self*, A_j, *RedMeetCalendar*: $D_k^{A_i}$*with*:W_{X_k});
13: **end for**
14: **end if**
15: **end for**
16: **end if**

Figura 2.3: Algoritmo 1: Ricezione di un messaggio di start

Quando un agente partecipante viene attivato tramite la ricezione del messaggio appena descritto viene eseguito l'algoritmo 2 (figura

Algorithm 2. *RedMeetCalendar*: *with*: message corpus
executed by each Participant agent A_i.
RedMeetCalendar: D **with**: W

```
1:  Delete from D all non-viable values;
2:  if D = ∅ then
3:     Send(self, sender, MeetNotPossible);
4:  else
5:     for all X_l^{A_j} such that ((X_l^{A_j}, dt_p) ∈ Calendar^{A_i})
    AND (W_{X_l} > W) do
6:        Delete(D, dt_p);
7:        if D = ∅ then
8:           Send(self, sender, MeetNotPossible);
9:        else
10:          Rank(D); /* According to A_i's preferences
    w_p^{A_i} */
11:          Send(self, sender, Reply:D);
12:       end if
13:    end for
14: end if
```

Figura 2.4: Algoritmo 2: Ricezione di un messaggio di RedMeetCalendar

2.4). L'agente elimina in primo luogo le date non consistenti con i suoi vincoli hard. Se non vi sono date disponibili invia un messaggio al proponente di tipo MeetNotPossible. In caso contrario l'utente continuerà ad eliminare le date nelle quali vi sono già altri eventi di maggiore importanza, ordinerà le rimanenti in base alla preferenza in maniera crescente e invierà le date, secondo questo ordine, all'agente proponente attraverso un messaggio di tipo Reply.

Il proponente, una volta ricevuti tutti i messaggi Reply, controlla che vi sia almeno una data in comune per organizzare l'evento. Se non vi sono date comuni invia a tutti i partecipanti un messaggio di tipo MeetNotPossible, altrimenti sceglie la data che massimizza il valore di utilità dell'evento. Per far questo assegna un valore crescente di utilità in base all'ordine delle date. In caso di due o più valori di pari utilità l'agente sceglierà quella con differenza di utilità minore rispetto a tutti gli agenti in quanto evita la scelta di date a cui qualche utente ha assegnato l'utilità minima. Una volta scelta la data la invia a tutti i partecipanti attraverso un messaggio di tipo ReceiveProp. Questa procedura viene descritta nell'algoritmo 3 (figura 2.5)

Come si evince invece dall'algoritmo 4 (figura 2.6), quando gli agen-

Algorithm 3. *Reply*: message corpus executed by each Proposer agent A_i.
Reply: D

```
1:  if All ranked D are received from all
    A_j ∈ Part(X_k^{A_i}) then
2:    Update(D_k^{A_i}); /* According to received Ds */
3:    dt_p ← the date with higher utility;
4:    if dt_p = nil then
5:      for all A_j ∈ Part(X_k^{A_i}) do
6:        Send(self, A_j, MeetNotPossible);
7:      end for
8:    else
9:      for all A_j ∈ Part(X_k^{A_i})
10:       Send(self, A_j, ReceiveProp:dt_p with:W_{X_k});
11:     end for
12:   end if
13: end if
```

Figura 2.5: Algoritmo 3: Ricezione di un messaggio Reply

ti partecipanti ricevono un messaggio di tipo ReceiveProp eseguono ulteriori controlli su quella data in quanto nel frattempo potrebbero aver ricevuto altre proposte di partecipazioni ad eventi. Se infatti, a quella data è stato assegnato un evento con peso maggiore dell'evento che si sta tentando di organizzare, il partecipante in questione invierà un messaggio di tipo UpdateProp al proponente dell'evento. Se invece, è presente un evento con priorità minore dell'evento che si sta organizzando l'agente invierà un messaggio di MeetingOk al proponente per quello che riguarda l'evento in questione, ed invierà un messaggio di tipo UpdateProp al proponente dell'evento che è stato sovrascritto in modo da richiedere la riorganizzazione dello stesso. Nel caso in cui non vi siano conflitti gli agenti partecipanti invieranno semplicemente il messaggio di tipo MeetingOk.

L'iter di organizzazione di un evento finisce qui, salvo che non sia stata richiesta la riorganizzazione di un evento attraverso l'invio di un messaggio del tipo UpdateProp. Quest'ultimo attiverebbe il proponente dell'evento cancellato che, dopo aver analizzato nuovamente le date riservate per quell'evento invierà un messaggio a tutti i partecipanti. Il messaggio sarà di tipo MeetNotPossible se non vi sono date comuni ai partecipanti, oppure di ReceiveProp per proporre una data ai partecipanti e ripetere l'iter sopra descritto. L'algoritmo che rappresenta questa procedura è il numero 5 (figura 2.7).

Algorithm 4. *ReceiveProp*: ***with***: message corpus executed by each Participant agent A_i.
ReceiveProp:*Prop* **with**:W

1: **if** $(\exists(X_l^{A_j}, dt_p) \in Calendar^{A_i}$ such that $Prop = dt_p$ and $W_{X_l} > W)$ **then**
2: *Send*(*self*, *sender*, *UpdateProp*:*Prop with*:W);
3: **else**
4: **if** $(\exists\ (X_l^{A_j}, dt_p) \in Calendar^{A_i}$ such that $Prop = dt_p$ and $W_{X_l} < W)$ **then**
5: *Send*(*self*, A_j, *UpdateProp*:dt_p *with*: W_{X_l});
6: *Send*(*self*, *sender*, *MeetingOK*);
7: **else**
8: *Send*(*self*, *sender*, *MeetingOK*);
9: **end if**
10: **end if**

Figura 2.6: Algoritmo 4: Ricezione di un messaggio ReceiveProp

Algorithm 5. *UpdateProp*: ***with***: message corpus executed by each Proposer agent A_i in case of conflict.
UpdateProp:*Prop* ***with***:W

1: Delete $(D_k^{A_i}, Prop)$;
2: Add$(DReserve^{A_i}[k], Prop)$;
3: **if** $D_k^{A_i} \neq \emptyset$ **then**
4: $dt_p \leftarrow$ the date with *higher utility* in $D_k^{A_i}$;
5: **for all** $A_j \in$ Part$(X_k^{A_i})$
6: *Send*(*self*, A_j, *ReceiveProp*: dt_p *with*: W_{X_k});
7: **end for**
8: **else**
9: **for all** $A_j \in Part(X_k^{A_i})$ **do**
10: *Send*(*self*, A_j, *MeetNotPossible*);
11: **end for**
12: **end if**

Figura 2.7: Algoritmo 5: Ricezione di un messaggio UpdateProp

2.3 Commenti

L'algoritmo appena descritto porta ad un equilibrio temporaneo tra due successive richieste di organizzazione di meeting.
Nel mondo reale, però, vi sono alcune direttive che, pur non facendo parte dell'algoritmo sopra descritto, debbono essere rispettate. Una di queste è il tempo di spostamento del quale necessitano i partecipanti per raggiungere il luogo nel quale si terrà il meeting. Se durante questi spostamenti viene richiesta l'organizzazione di un meeting più importante ed imminente e la sua data di collocazione viene scelta proprio

in concomitanza del meeting al quale si stanno recando alcuni partecipanti, è impensabile che questi facciano 'marcia indietro' per raggiungere un meeting alternativo in quanto quello verso il quale ci si stava recando verrebbe automaticamente cancellato. Questo porterebbe ad una perdita di tempo e di denaro che potrebbero ripetersi se nello stesso tempo verrebbe richiesta l'organizzazione di un meeting ancora più importante di quest'ultimo. E così via... E', quindi necessario, imporre un limite di tempo minimo tra la possibile data di un meeting e il momento in cui questa organizzazione viene richiesta.
Inoltre, un meeting di scarsa importanza corre il rischio di essere rinviato diverse volte, provocando insoddisfazioni nei propri partecipanti. Si potrebbe infatti pensare di aumentare il peso di un meeting ogni volta che ne viene richiesta la rischedulazione. In questo modo un meeting diventerebbe tanto più importante quante volte è stato rimandato e non si avrebbero meeting che, teoricamente, potrebbero essere rimandati all'infinito, portando ulteriore stabilità nel sistema.

Capitolo 3

Un esempio pratico: MAgentA

L'Università degli Studi di Brescia sta lavorando ad un progetto che ha come scopo, appunto, l'organizzazione automatica di impegni ed eventi utilizzando proprio un'architettura ad agenti, ed in particolare utilizzando la piattaforma JADE e lo standard di comunicazione FIPA. Il software sarà utilizzato all'interno dell'Università per l'organizzazione degli appuntamenti dei docenti universitari.

3.1 Architettura

I dati necessari al funzionamento del programma vengono memorizzati in tre diversi database. Un database viene utilizzato per memorizzare gli utenti registrati nel sistema, uno per gli eventi che si devono organizzare ed un terzo per ogni utente utilizzato al fine di memorizzare i propri impegni e le proprie preferenze.
Relativamente a queste fonti di dati, vi sono inoltre 4 tipologie di agenti, ognuno con le proprie funzioni. L'agente UPA (User Personal Agent) ricopre il ruolo dell'agente Utente descritto nel capitolo precedente. Questo agente, inoltre, è l'unico ad accedere al database personale dell'utente che l'agente stesso rappresenta. Vi è poi l'agente UMA (User Management Agent) che ha l'accesso esclusivo al database degli utenti registrati. Un altro agente è l'MMA (Meeting Manager Agent) che ha il ruolo dell'agente Interfaccia descritto nell'algoritmo del capi-

tolo precedente. Questo agente ha l'accesso al database dei meeting ed ha il compito di starter durante la fase iniziale di organizzazione. Infine, vi sono gli agenti GA (GUI Agent) che si occupano dell'interfaccia dei singoli utenti e di intermediare le informazioni tra l'utente ed il suo UPA.

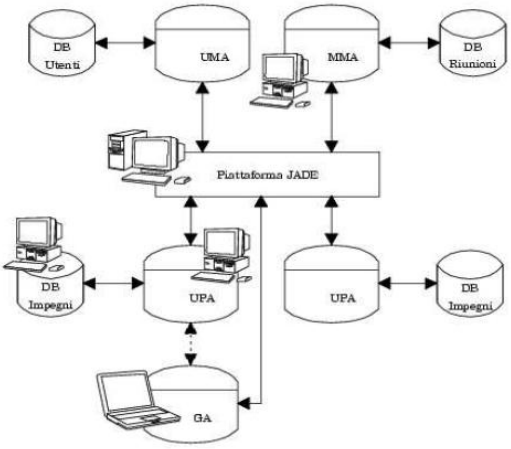

Figura 3.1: Architettura di MAgentA

3.2 Funzionamento

Nel momento in cui viene lanciato il servizio fornito da MAgentA sull'ambiente vengono creati l'UMA e il MMA. Questi agenti all'avvio accedono al database e creano tanti UPA quante sono le agende presenti nel database.
Ogni volta che un utente vuole accedere alla propria agenda viene istanziato il relativo GA che, comunicando con l'UPA, richiede all'utilizzatore l'autenticazione al sistema con il classico binomio 'nome utente - password' per verificare l'autenticità dell'utente e soprattutto per registrare l'indirizzo del GA in modo che l'UPA relativo alla stessa agenda possa mettersi in attesa di messaggi di richieste inviati dall'interfaccia utente. Una volta effettuato il login da parte dell'utente il relativo UPA rimane in ascolto di comandi, ai quali risponde

con l'esito della operazione richiesta.
Quando un utente decide di inserire un nuovo meeting attraverso l'interfaccia utente deve specificare i vincoli da rispettare, quali l'elenco dei partecipanti, alcune date possibili in cui effettuare la riunione, la durata della riunione, ecc. Il GA dell'utente invierà al proprio UPA tale richiesta, che a sua volta inoltrerà al MMA inserendo ulteriori vincoli dedotti dall'agenda di cui l'agente UPA è responsabile.
A questo punto entra in gioco l'algoritmo descritto nel capitolo 2, ossia l'MMA invia dei messaggi di richiesta agli UPA dei partecipanti con una lista di possibili date che rispondono con la stessa lista in cui sono state eliminate le date che creano conflitti con meeting già organizzati. Se non vi sono date comuni a tutti gli utenti l'MMA invia a tutti i partecipanti un messaggio di fallimento, altrimenti mostra all'iniziatore le possibili soluzioni ordinandole in base alla funzione obiettivo in modo che la decisione finale della data sia lasciata all'utente iniziatore. Quando questo sceglierà la data definitiva il proprio UPA invierà la data al MMA che a sua volta la invierà agli UPA partecipanti ed attenderà la risposta di conferma per ritenere organizzato il meeting.

Concludiamo l'articolo mostrando alcuni screenshot del software MAgentA.

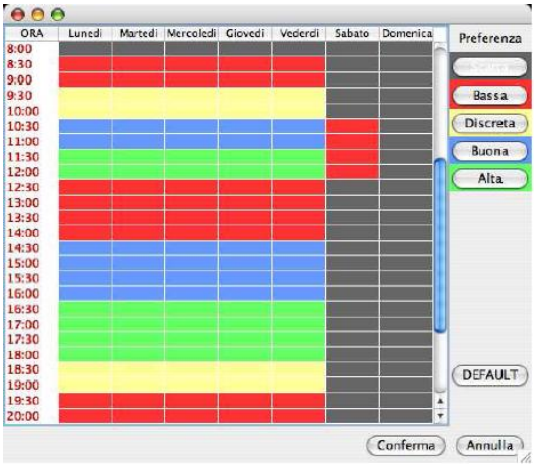

Figura 3.2: Vista delle preferenze di un utente

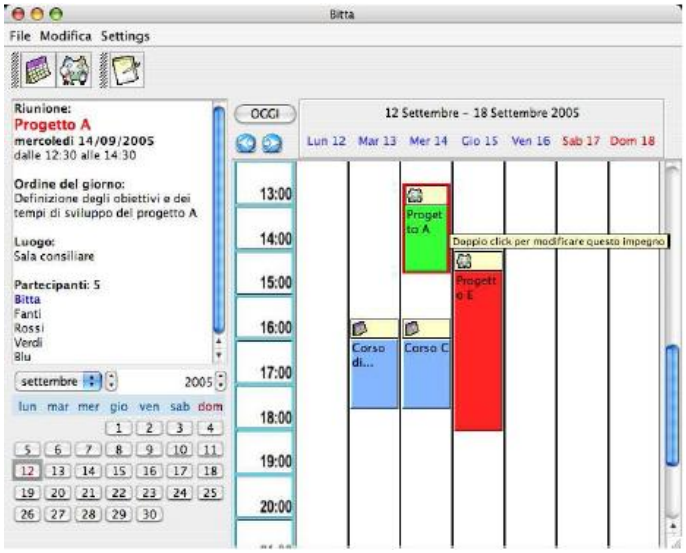

Figura 3.3: Vista dell'agenda di un utente

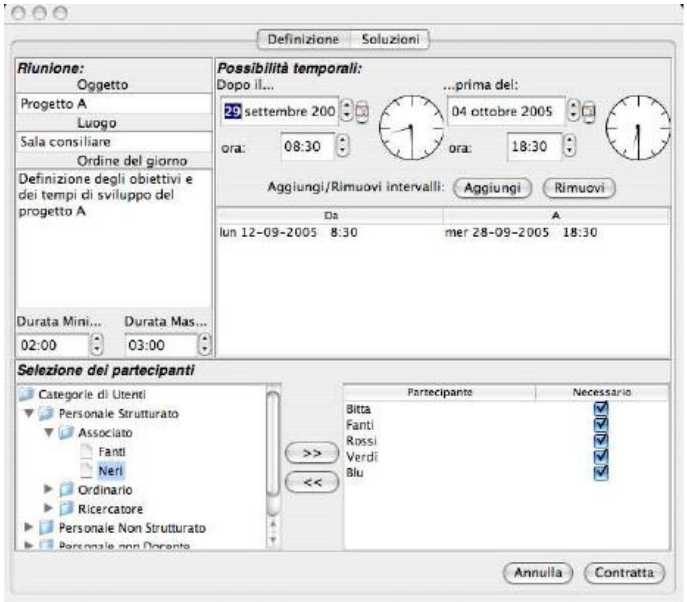

Figura 3.4: Definizione di un meeting con i relativi vincoli

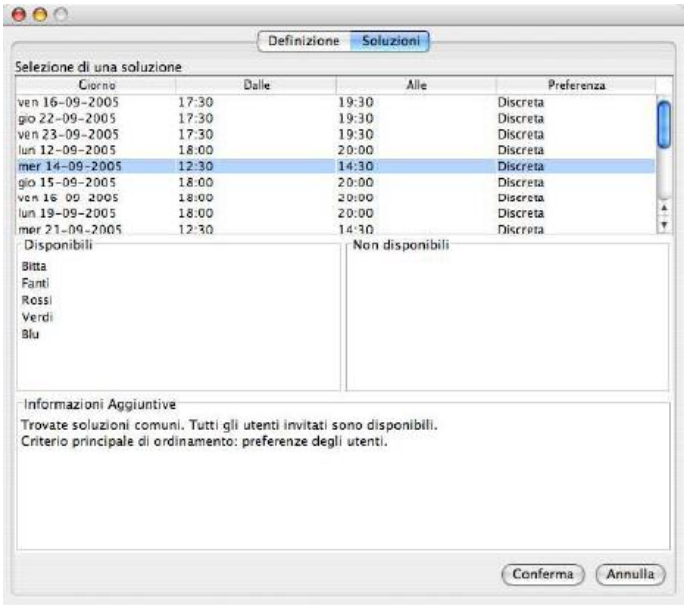

Figura 3.5: Elenco delle possibili soluzioni