Sommario

Introduzione

L'aggregazione di persone attraverso l'uso di mezzi informatici è un tema forte e moderno: esso ha aperto letteralmente un nuovo mercato. Basti pensare alla diffusione delle comunità virtuali e dei *social network*, presso cui ad oggi moltissimi individui e gruppi hanno creato un proprio profilo, aggiornando quotidianamente il proprio stato, condividendo contenuti multimediali e altre informazioni personali, organizzando eventi; la comunità web Facebook, un noto social network, conta attualmente più di trecento milioni di iscritti [2]. In particolare, l'organizzazione di eventi concretizza l'aggregazione tra individui, che in effetti costituisce l'obiettivo di un social network, perciò essa è inclusa tra le funzionalità più comuni disponibili sui social network, ma anche su comunità dedicate.

Il concetto di automatismo nell'ambito dei social network fa pensare a una semplificazione delle operazioni più comuni e allo stesso tempo alla possibilità per gli utenti di realizzare procedure complesse di interazione sociale. Questo è ora possibile attraverso l'implementazione di sistemi multiagente: attraverso di essi, entità chiamate agenti, che rappresentano gli utenti, operano in modo autonomo portando avanti gli interessi degli utenti stessi, attraverso la collaborazione, la competizione e la negoziazione fra agenti.

I telefoni cellulari di ultima generazione offrono una tecnologia avanzata e moltissime potenzialità, grazie anche alla capacità di connettersi su diverse bande; è disponibile anche la connessione ad Internet. Inoltre tali dispositivi sono detti mobili perché sempre alla portata di mano degli utenti. Per questi motivi, i social network più utilizzati della rete hanno rivolto la loro attenzione ai telefoni cellulari come ad un importante mezzo d'interazione con la comunità.

La tesi si pone come obiettivo di sviluppare il prototipo di un social network per dispositivi mobili, basato su un sistema ad agenti, con particolare interesse verso l'organizzazione e la gestione di eventi: per questo motivo il prototipo è stato chiamato **MOEVENT**. La possibilità di realizzare questo sistema ci è stata offerta dall'unione di *JADE*, una piattaforma ad agenti creata da Telecom Italia, dal sistema operativo *Android* di Google, open e pensato per telefoni di ultima generazione, e dal plug-in *JADE LEAP for Android* per fondere le due architetture.

Nei capitoli successivi sono trattati i seguenti argomenti:

- **Social network e sistemi ad agenti**. Sono espresse le motivazioni che ci hanno spinto a ideare un sistema di pianificazione di eventi attraverso l'utilizzo di una sistema ad agenti, a partire da alcune analisi sullo stato dell'arte dei social network e sulle potenzialità offerte da un'implementazione dello stesso con un sistema multiagente.

- **Tecnologie per sistemi ad agenti rivolti ai dispositivi mobili**. Si effettua una panoramica delle tecnologie individuate per realizzare il prototipo (JADE, Android).
- **Progettazione di MOEVENT.** Si svolge un'analisi dei requisiti funzionali e non funzionali, e si descrive l'architettura logica del sistema.
- **Implementazione di MOEVENT.** Sono descritte le scelte di sviluppo del prototipo, attraverso l'uso di diagrammi delle classi e relativa descrizione in dettaglio, e diagrammi di flusso per la spiegazione del funzionamento nel suo insieme.
- **Testing e considerazioni generali.** Sono descritte le sperimentazioni svolte, riguardo al soddisfacimento dei requisiti e alcuni test sulle performance del sistema; seguono alcune considerazioni generali sul progetto.
- **Conclusione.**
- **Bibliografia.**

Capitolo 1
Social network e sistemi ad agenti

In questo capitolo svolgeremo dapprima un'analisi sui social network e sulla loro diffusione presso il pubblico di Internet; di seguito, tratteremo le tecniche di implementazione comumente utilizzate per creare un social network, l'estensione dei social network al mercato dei dispositivi mobili, quindi parleremo di sistemi ad agenti e delle potenzialità che questi potrebbero offrire ai social network.

1.1 Social network e tecnologie

La definizione attuale di social network è la seguente:

> *"Un'ambiente relazionale online, generato da un database, in cui sono raccolti dati descrittivi di persone (profili), e che permette ai membri che vi si sono iscritti di indicare ad altri con chi desiderano entrare (o sono già) in contatto. I "dati descrittivi" e i "dati relazionali" una volta analizzati permettono di ricostruire la mappa delle relazioni di un singolo individuo all'interno del network."*

Tale definizione è stata affibiata a un famoso social network, Friendster, da U.S. Patent & Trademark Office [1]; si tratta di una definizione tecnica, che si adatta perfettamente ai social network presenti sul mercato, ma è un'evoluzione del concetto originale: un gruppo di individui interconnessi da relazioni di diversa natura. In genere, la relazione è un rapporto di conoscenza nato ad esempio in ambiente scolastico o lavorativo, ma può anche essere dovuta alla condivisione d'idee o interessi. La rappresentazione di un social network è un grafo i cui nodi rappresentano gli individui che costituiscono il network e i cui archi rappresentano le relazioni fra gli individui, come si può vedere in Figura 1.

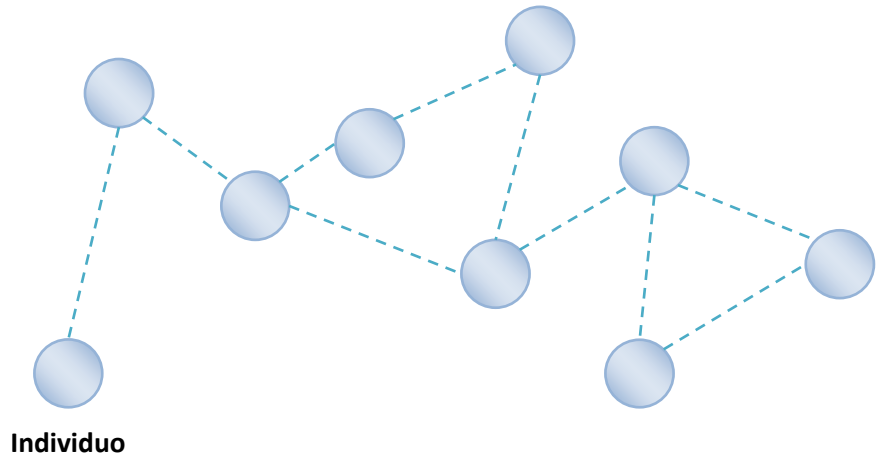

Figura 1. Rappresentazione di un social network

Tra i social network più noti ci sono MySpace, Facebook, LinkedIn, Friendster e Orkut. Facebook ha attualmente raggiunto le dimensioni di un continente: ne fanno uso più di trecento milioni di persone, di cui il 50% effettua quotidianamente l'accesso alla comunità; questi dati sono riportati dalle statiche dello stesso social network [2].

Le funzionalità più comuni offerte dai social network sono: la pubblicazione di un diario (*weblog*); la condivisione di contenuti multimediali come foto e video; la condivisione di idee personali e la partecipazione a cause ed argomenti di interesse pubblico; la ricerca di membri e contenuti del social network; la comunicazione con gli stessi membri; la creazione e la condivisione di liste di contatti personali; l'organizzazione di eventi.

Ogni social network implementa funzionalità specifiche, oppure offre a terze parti la possibilità di creare le proprie applicazioni, come sondaggi o giochi. Facebook permette l'integrazione di applicazioni di terze parti, come giochi Flash, con la comunità online: le statistiche riportano che correntemente esistono più di 350 mila applicazioni attive; le 250 applicazioni più utilizzate hanno un seguito di più di un milione di utenti ogni mese [2].

Alcuni social network si pongono degli obiettivi specifici e offrono funzionalità limitate a un ambito caratterizzante; si tratta di social network più vicini al concetto originale, diffusi ma rivolti a un pubblico ristretto. Ad esempio, esistono reti costituite di aziende e consumatori che hanno lo scopo di avvicinare le due parti con delle relazioni basate sulla reputazione e sulla fiducia.

In ogni caso, si può dedurre che i social network siano in un periodo di forte espansione.

L'implementazione di un social network si attua con la creazione di:

- una banca dati che contenga le informazioni riguardo gli utenti della rete e le relazioni che intercorrono fra di essi;
- un'interfaccia utente che permetta di consultare le informazioni disponibili presso la banca dati, di aggiungere ad esse nuovi contenuti o di modificare quelli esistenti.

Inoltre, una caratteristica fondamentale è l'accessibilità dei contenuti, poterli consultare e modificare in ogni momento e in ogni luogo. Perciò, la tecnologia intuitivamente più idonea a coprire i requisiti tipici di un social network è la *web community*: una comunità *online* è accessibile da qualunque locazione che abbia accesso a Internet e offre un'interfaccia di pagine a contenuto dinamico per svolgere le operazioni più comuni descritte in precedenza. Gran parte dei social network è realizzata come una comunità online.

Esistono alcune realtà alternative come Second Life [3], che utilizza un applicativo client tridimensionale in cui muovere il proprio *avatar* per socializzare con gli altri utenti; tuttavia, Second Life presenta un approccio più simile a quello di una chat, piuttosto che di un vero e proprio social network.

La diffusione sul mercato di telefoni cellulari di ultima generazione ha spinto i social network ad espandersi verso l'ambiente *mobile*. Le comunità più popolate del Web, come Facebook [4] o Myspace [5] hanno sviluppato delle applicazioni per cellulari di ultima generazione, come Apple iPhone oppure HTC Diamond, che ospita il sistema operativo Android. Questi dispositivi hanno ottime potenzialità per accedere ai social network online, essendo dotati di una connessione ad Internet costante, grazie alla capacità di trasmettere su diverse bande, ed uno schermo ampio e luminoso; per poter usufruire della connessione ad Internet è necessario sottoscrivere un abbonamento con un operatore telefonico, che in alcuni paesi, come in America o in Giappone è molto più economico che in Italia e nei paesi europei. Le applicazioni rilasciate dai social network vengono installate sui dispositivi mobili e conferiscono l'accesso a tutte le funzionalità offerte dalle pagine della web community, consultabili attraverso un *browser*. Ad esempio, si possono inviare messaggi agli utenti della comunità, si può modificare il proprio stato, si possono caricare nei propri album le foto scattate con il telefono, etc., poiché l'applicativo accede direttamente ai contenuti evitando l'operazione lenta e onerosa di scaricare per intero le pagine della comunità sul browser del dispositivo. La Figura 2 mostra un confronto tra una pagina pubblica di Facebook scaricata attraverso il browser Mozilla Firefox e una pagina visualizzata dall'applicativo Facebook per Google Android.

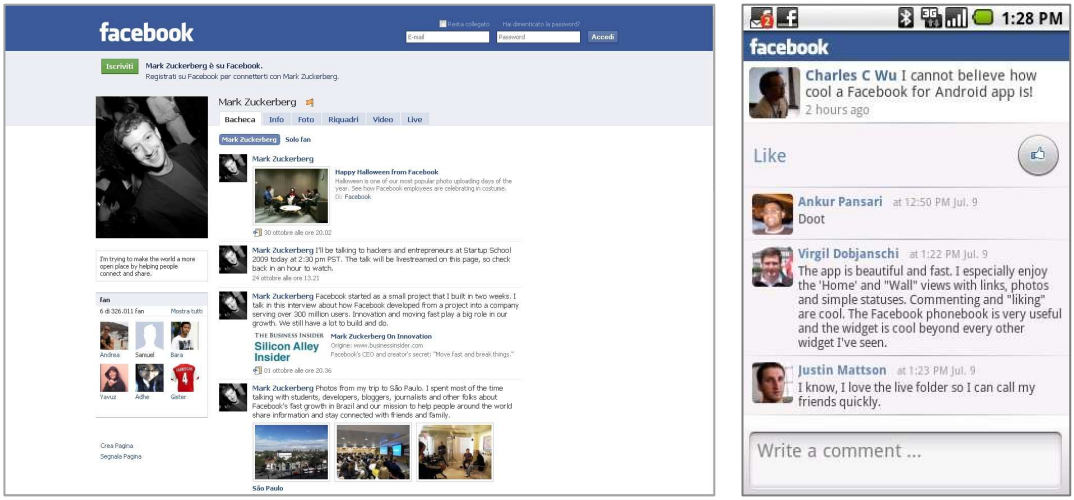

Figura 2. Facebook e Facebook *mobile*

1.2 Agenti e sistemi multiagente: concetti

La necessità di svolgere computazioni complesse in modo automatico è alla radice della nascita dell'informatica. L'uomo ha delegato alla macchina il compito di svolgere queste operazioni, mantenendone tuttavia il controllo o la supervisione. Successivamente, l'evoluzione della tecnologia ha portato al desiderio dell'uomo di lasciare alla macchina un maggiore carico di lavoro, una maggiore capacità decisionale e gestionale: è così che nasce il concetto di agente.

Un agente agisce in modo autonomo per conto di un individuo, al fine di venire incontro ai suoi obiettivi personali [6]. Gli interessi e gli obiettivi di un individuo possono incontrarsi o scontrarsi con gli interessi e gli obiettivi degli altri individui. I sistemi distribuiti, interconnessi fra loro, non erano predisposti a svolgere questa funzionalità, poiché le loro parti mancavano di capacità di collaborazione, di negoziazione o di competizione. Tutto ciò ha portato alla creazione di *sistemi multiagente*.

Che cos'è un agente?

> *"Agents are simply computer systems that are capable of autonomous action in some environment in order to meet their design objectives. An agent will typically sense its environment (by physical sensors in the case of agents situated in part of the real world, or by software sensors in the case of software agents), and will have available a repertoire of actions that can be executed to modify the environment, which may appear to respond non-deterministically to the execution of these actions."* (M. Wooldridge)

In altre parole, un agente è un sistema in grado di intraprendere in modo indipendente azioni basate sul comportamento dell'utente. In un sistema multiagente, diversi agenti collaborano o competono per raggiungere i propri obiettivi, talvolta comuni, esattamente come accade per gli individui che appartengono a una comunità.

I moderni sistemi multiagente si basano su sistemi distribuiti, eterogenei e scalabili, e rispettano i seguenti principi:

- ubiquità: i sistemi software moderni sono integrati nelle tecnologie più disparate;
- interconnessione: i sistemi distribuiti presuppongono che vi sia un'interconnessione tra gli apparati; oggi sono moltissimi gli apparecchi interconnessi grazie alla diffusione dell'accesso alla rete Internet;
- delega degli obiettivi: una conseguenza diretta dell'automazione; il fatto che esista del software che svolge in maniera automatica ed indipendente dei compiti permette che gli venga assegnato un sempre maggiore numero di ruoli;
- intelligenza: si tratta di una caratteristica legata alla delega degli obiettivi, specifica che gli obiettivi in questione sono d'interesse per l'utente;
- abilità sociali: poiché gli agenti dovrebbero portare avanti interessi di utenti diversi, spesso accade che tali interessi siano differenti oppure opposti, pertanto si pretende che l'agente abbia delle capacità di confronto con gli altri agenti, al fine di negoziare, collaborare/coordinarsi o competere con essi;
- coscienza del contesto (context-awareness): avere coscienza del contesto significa avere la conoscenza dei tratti caratterizzanti dell'ambiente, delle entità che si trovano in esso e delle azioni che possono essere intraprese verso queste entità;
- orientamento agli obiettivi (goal-oriented): duranti le prime fasi di sviluppo, il software era orientato alla macchina, andava interpretato dagli sviluppatori; successivamente il linguaggio software si è avvicinato a quello dello sviluppatore, permettendogli di esprimere dei concetti vicini alla sua percezione della realtà,

piuttosto che a quella della macchina; il primo passo in questa direzione è segnato dalla nascita della programmazione ad oggetti; un altro passo avanti coincide con la nascita della programmazione ad agenti - benché gli agenti non siano adatti ad implementare qualunque sistema applicativo – e lo sviluppo di un'analisi software goal-oriented.

La visione degli agenti è *human-oriented*: l'agente è percepito come un attore umano all'interno di uno scenario software. All'agente vengono infatti delegati dei compiti che sono svolti allo stesso modo in cui sarebbero svolti da un individuo, ma esso agisce indipendentemente seppure rappresentandone gli interessi. Interi sistemi ad agenti sono in grado di collaborare e competere con altri sistemi. Nel momento in cui un agente si comporta in modo corretto ma non previsto, esso è intelligente.

Gli agenti agiscono su due possibili tipi di ambiente: fisico o virtuale; nel primo caso, l'agente è dotato di sensori che raccolgono informazioni sull'ambiente stesso e di effettori che modificano l'ambiente in seguito ad un determinato comportamento dell'agente; nel secondo caso, si tratta di ambienti software, dove gli eventi sono registrati tramite dei processi in ascolto. In entrambi i casi, l'ambiente può essere comunque definito:

- accessibile o inaccessibile: con accessibile si intende un ambiente che l'agente può conoscere perfettamente (molti ambienti fisici non sono accessibili);
- deterministico o non deterministico: l'azione di un agente in un ambiente deterministico conduce sempre allo stesso stato;
- statico o dinamico: un ambiente dinamico subisce le modifiche apportate da un agente, un ambiente statico non ne risulta influenzato;
- discreto o continuo: un ambiente è definito discreto se su di esso è possibile svolgere al più un numero finito di azioni;
- grado di fairness: è difficile fare in modo che il comportamento di un agente non abbia effetto a lungo termine quando non è necessario (ossia non è sempre giusto intraprendere la stessa azione).

Una variabile importante è il tempo di ragionamento, soprattutto per tutte quelle piattaforme che integrano l'ambiente *mobile*; ci sono diverse tipologie di interazione in tempo reale: quelle in cui la decisione riguardo a quale azione debba essere intrapresa deve avvenire entro un limite di tempo; quelle in cui è l'agente che per realizzare al meglio il proprio obiettivo deve rispondere in tempo reale, cioè più rapidamente possibile; quelle in cui l'agente è costruito per ripetere un compito il maggior numero possibile di volte. Naturalmente le piattaforme estese all'ambiente mobile rivolgono la loro attenzione al primo caso.

Esistono diverse tipologie di agente, ma possiamo dire che, in base ai comportamenti che assume, un agente può essere:

- reattivo;

- proattivo;
- provvisto di capacità sociali.

  Un agente è reattivo se risponde agli stimoli causati da un evento che avviene nel suo ambiente: in altre parole, esso è in grado di rispondere ai cambiamenti del mondo intorno a sé. Tuttavia, un agente reattivo non prende mai autonomamente l'iniziativa di comportarsi in un certo modo per un certo scopo: un agente di questo secondo tipo è detto proattivo. Infine, un agente è provvisto di capacità sociali nel caso in cui tenti di collaborare con altri agenti presenti nell'ambiente per soddisfare un proprio obiettivo.

Poiché, come abbiamo visto, non esiste una definizione formale di cosa sia un agente, possiamo descriverne il concetto attraverso le sue caratteristiche principali:

- autonomia, controlla il proprio stato ed i propri comportamenti (*behaviour*);
- integrazione nell'ambiente, da cui trae informazioni con dei mezzi, detti sensori, e su cui opera tramite altri mezzi, detti effettori;
- flessibilità, reattività e pro-attività, in quanto reagisce dinamicamente agli eventi, e prende l'iniziativa per raggiungere un obiettivo ogni volta che ci siano le condizioni necessarie, senza che alcun evento stimolante sia avvenuto.

Altre caratteristiche di un agente sono:

- capacità di collaborazione, ossia di cooperare con altri agenti o entità per raggiungere un obiettivo comune;
- capacità di comunicare attraverso una forma di linguaggio specifica con altri agenti;
- capacità inferenziale;
- continuità temporale;
- personalità;
- adattabilità, ossia la caratteristica di acquisire esperienza con cui estendere e migliorare le proprie capacità di scelta;
- mobilità, ossia la capacità di migrare attraverso una piattaforma da una macchina all'altra.

Tra la programmazione a oggetti e quella ad agenti intercorrono delle differenze: innanzitutto, un oggetto è semplicemente costituito di campi e metodi che offrono determinati servizi, mentre un agente ha delle capacità intrinseche (servizi) e degli obiettivi. I servizi offerti da un software non orientato ad agenti ricoprono dei particolari *task*, difficilmente modificabili a posteriori senza interventi sul codice sorgente; al contrario, gli agenti offrono servizi ben determinati, ma che possono essere soddisfatti attraverso diversi tipi di implementazione.

Un sistema ad agenti non è solitamente costituito di un unico agente, ma di un insieme di agenti con diversi comportamenti, capacità e obiettivi, in grado di comunicare fra loro

per cooperare, coordinarsi, negoziare e competere al fine di raggiungere un obiettivo preposto. Talvolta, esistono degli obiettivi sociali, preposti dall'insieme degli agenti, che possono scontrarsi con gli obiettivi di un singolo agente.

Essendo una piattaforma ad agenti del tutto simile ad una comunità, un'organizzazione di agenti prevede il rispetto di determinate regole, la formazione di una gerarchia e l'imposizione o l'assunzione di ruoli.

L'architettura software di un sistema ad agenti può dunque essere:

- reattiva, nel caso in cui si tratti di sistemi semplici, in cui i comportamenti sono condizionati da regole e servono a raggiungere degli obiettivi; esistono tipi di comportamento più o meno complessi, che l'agente tenta di eseguire nel rispetto di una gerarchia; ad esempio, le azioni di un semplice robot dotato di sensori e di un motore possono essere regolamentate dai comportamenti di un agente reattivo, il quale raccoglie input dai sensori, progetta una modifica alla realtà che è in grado di percepire, identifica le entità e gli ostacoli presenti nell'ambiente, avverte cambiamenti dell'ambiente e ne disegna mappe, esplora ed evita gli ostacoli. Il vantaggio di utilizzare questo sistema sta a monte nella semplicità di sviluppo dello stesso, d'altra parte un sistema reattivo è limitato per quanto riguarda le capacità di apprendimento dell'agente;
- ibrida;
- deliberativa, o che appartiene alla categoria BDI (belief desire intention) ed introduce il concetto di *piano*, con cui si intende una sorta di *know-how* per raggiungere un obiettivo piuttosto che non un elenco di azioni da intraprendere.

Le *convinzioni* (*believes*) di un agente sono in sostanza le informazioni a sua disposizione, raccolte nel corso della sua esperienza; tali informazioni risultano utili in quanto un agente ha visione soltanto dell'ambiente in cui si trova, e perciò risulta essere più performante un agente che sfrutta informazioni già raccolte piuttosto che un agente che svolgere ripetutamente gli stessi calcoli. Tuttavia, le informazioni raccolte non sono sempre precise e corrette, perciò esse rappresentano più ciò che l'agente *crede* di ciò che l'agente *sa*. Il caching delle convinzioni può essere esteso ai piani.

I *desideri* (*desires*) di un agente rappresentano il suo scopo ultimo, perciò conferiscono significato e motivazione ad una specifica sezione di codice in esecuzione.

Le *intenzioni* (*intentions*) sono le istanze di un piano, ovvero un insieme di azioni che l'agente intraprende per coprire un obiettivo, e possono essere usate per coordinare gli agenti.

Un'architettura ad agenti è definita come una metodologia di creazione degli stessi, specifica come l'agente può essere scomposto in moduli e come tali moduli interagiscono fra loro. In genere, un'architettura deduttiva si basa su:

- un sistema di regole;

- un base di dati che contiene informazioni riguardo le regole stesse e le conoscenze degli agenti;
- un insieme delle azioni.

Un'azione modifica lo stato del sistema, e quindi viene eseguita se lo stato che essa produce si può provare induttivamente a partire dalle informazioni contenute nel database ed attraverso il sistema di regole.

Analizziamo il caso di un robot addetto alle pulizie: *regola* - se il robot si trova in un'area limitata della stanza definita come una casella - *informazioni* sull'ambiente ottenute da input - e non c'è polvere - altre *informazioni* raccolte da input - avanza - *azione* - se possibile, altrimenti si gira - *azione* - in senso orario.

In questo modo ci si para davanti il problema di tradurre in logica, o per meglio dire in deduzioni logiche, i dati provenienti dai sensori (nell'esempio del robot spazzino, i sensori determinano la presenza di polvere), ed il problema di dover risolvere problemi indecidibili.

Una soluzione è indebolire, semplificare la logica; utilizzare rappresentazioni simboliche non logiche; spostare il ragionamento da run-time a design-time: dato che gli agenti sono autonomi, ovvero ragionano in maniera indipendente, dovrebbero solitamente essere in grado di reagire durante l'esecuzione ad imprevisti non considerati nella fase di design.

Un agente assume i comportamenti che lo portano alla realizzazione dei propri obiettivi, e spesso collabora o compete con altri agenti dello stesso sistema; le abilità sociali degli agenti comportano che essi abbiano la capacità ed i mezzi per la comunicazione. La comunicazione avviene solitamente attraverso lo scambio di messaggi.

I messaggi appartengono a tre tipologie:

- richiesta (request): se un agente riceve questo messaggio, significa che un altro agente gli sta chiedendo di eseguire un'azione particolare;
- non-richiesta (unrequest): utilizzato da un agente che chiede ad un altro agente di rifiutare un'azione;
- informazione (inform): un agente invia questo messaggio ad un altro quando desidera semplicemente comunicargli un'informazione.

Passiamo ora ad analizzare le connessioni tra sistemi multiagente e social network.

1.3 Sistemi ad agenti per il social networking

Un social network in ambito informatico è la rappresentazione virtuale di una rete sociale costituita da individui umani; tuttavia, l'utente di un social network non è pienamente rappresentato in essa, perché è costretto a svolgere manualmente la quasi totalità delle procedure comuni. Al contrario, potrebbe rivelarsi utile lo svolgimento automatico di alcune operazioni: l'automatismo semplifica e velocizza i processi.

Supponiamo che un utente di Facebook desideri partecipare a un corso di fotografia, ma non conosca direttamente alcun utente disponibile a farlo, o che non sia in contatto con un gruppo che organizza dei corsi; per ottenere le informazioni che desidera, l'utente è costretto a chiedere a ciascuno dei propri contatti se lo possono aiutare. Se la richiesta fosse inoltrata in modo automatico fino a trovare l'utente o il gruppo che la soddisfa, tutti gli utenti coinvolti risparmierebbero tempo, in particolar modo gli intermediari che non hanno alcun interesse nell'operazione in questione.

In altre parole, c'è bisogno di *qualcuno* che possa svolgere per l'utente alcune operazioni in modo automatico: il miglior candidato ad attuare questa soluzione è un *agente*, poiché l'automatismo, il ragionamento e la negoziazione sono proprietà tipiche di un sistema multiagente. Il parallelismo tra agenti e utenti nel contesto del social networking si può osservare in Figura 3.

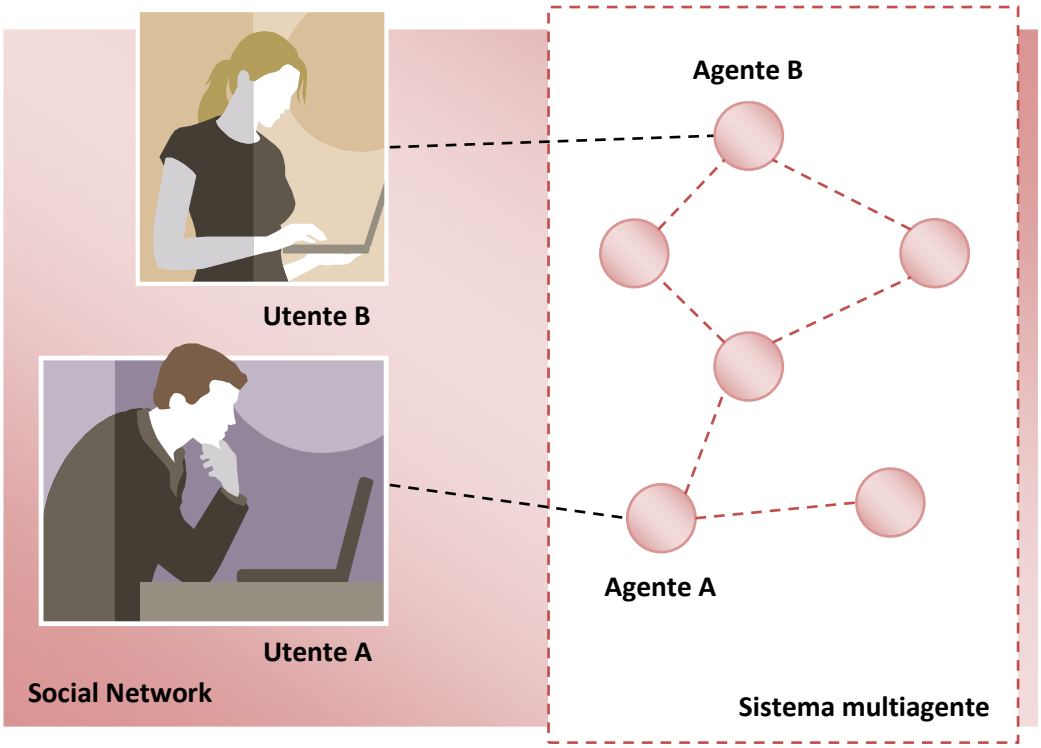

Figura 3. Sistema multiagente per il social networking

Come si può dedurre, gli utenti che non possiedono connessioni dirette non necessitano di contattare uno o più intermediari per poter comunicare perché l'operazione è svolta in modo automatico dagli agenti che li rappresentano. Nel nostro esempio, la richiesta di poter partecipare a un corso di fotografia viene trasmessa da Utente A in Figura 3 agli agenti che rappresentano i suoi contatti personali, i quali provvedono ad inoltrarla un passo alla volta fino a Utente B; il meccanismo è completamente trasparente agli occhi degli utenti.

Ne possiamo dedurre che un sistema multiagente può implementare un social network espandendo le tecnologie attualmente utilizzate con nuove potenzialità; nessun social network ha però implementato questa soluzione fino ad ora. Alcuni social network che operano in un ambito specifico hanno scelto di appoggiarsi su sistemi multiagente. Ad esempio, esistono sistemi di ranking votati al commercio elettronico, come Regret, il cui scopo è attribuire a ciascun agente un punteggio rappresentante la sua propria reputazione, calcolato in base ai dati raccolti dalla comunità secondo il punto di vista individuale degli agenti in relazione con esso [7, 8]. Esistono anche sistemi multiagente che simulano le reti sociali per studiare il profilo comportamentale della comunità in seguito ai cambiamenti registrati dal singolo individuo [9, 10]; talvolta questi simulatori sono legati a un ambito specifico, come quello militare [11]. Tra i social network e i sistemi multiagente intercorre un legame molto forte: alcuni sistemi multiagente fanno sì che i loro agenti possano sfruttare le proprietà tipiche dei social network per costruire coalizioni basate su princìpi di fiducia per realizzare i propri obiettivi [12]. In effetti, è intuitivo associare gli agenti dei sistemi multiagente agli utenti di un social network, poiché diverse caratteristiche li accomunano: entrambi assumono dei comportamenti specifici, perseguono degli obiettivi individuali e sociali, che cooperano, competono o negoziano fra loro per raggiungere, etc.

I nostri sforzi nella direzione di assimilare social networking e sistemi ad agenti si sono concentrati in particolare sull'organizzazione di eventi per dispositivi mobili. Organizzare eventi è una funzionalità tipicamente offerta dai social network e non solo, che abbiamo ritenuto importante in quanto concretizza la volontà di aggregazione degli individui di un social network. Vale la pena citare ancora una volta le statistiche di Facebook per testimoniarne l'importanza: ogni mese gli utenti della comunità organizzano più di tre milioni di eventi [2].

1.4 Organizzazione di eventi con strumenti web

Supponiamo di voler organizzare una partita di calcio a cinque. L'organizzatore sfoglia innanzitutto la propria agenda per verificare i propri impegni, quindi individua alcuni fra gli orari disponibili. In seguito, per formare due squadre di cinque persone ciascuna, deve invitare esattamente nove amici, fornendo loro indicazioni circa giorno, ora e luogo dell'incontro; inoltre, potrebbe desiderare rendere note altre informazioni, quali la lista temporanea dei partecipanti, la locazione del campo in cui si giocherà la partita, la quota di partecipazione pro capite, etc. Per invitare gli amici a partecipare alla partita, l'organizzatore può scegliere di chiamarli al telefono o inviare loro messaggi di posta elettronica, o utilizzare un qualsiasi altro metodo. A propria volta, i giocatori convocati verificano la propria disponibilità, relativa alle informazioni fornite, decidono se accettare o no di partecipare alla partita e infine, comunicano la loro decisione all'organizzatore. Se un invitato decide di non partecipare, deve essere rimpiazzato in modo da poter raggiungere il numero esatto di giocatori. Una volta che è stato raggiunto il numero totale di dieci partecipanti, compreso l'organizzatore, questi informa i propri giocatori che la partita avrà luogo; o in caso contrario, che è stata annullata, perché non

c'è più tempo a disposizione per cercare i giocatori mancanti, oppure perché c'è un campo disponibile, etc.

Nella realtà esistono casi molto più complessi di quello descritto nell'esempio: durante l'organizzazione di una partita alcuni partecipanti ritardano eccessivamente o ritirano la propria adesione, il che comporta un ulteriore sforzo da parte dell'organizzatore per raggiungere il numero esatto di giocatori convocati; l'organizzatore non ha un contatto diretto con alcuni partecipanti perciò deve fare affidamento su degli intermediari per avere informazioni su di essi e sulle loro adesioni; la partita è organizzata da dieci amici, i quali desiderano essere presenti per l'occasione, perciò uno o più portavoce svolgono un sondaggio per sapere in quale data tutti i giocatori sono disponibili. In altre parole, nel caso pessimo, ogni partecipante comunica con ciascuno degli altri, creando una situazione estremamente caotica, come si può vedere in Figura 4.

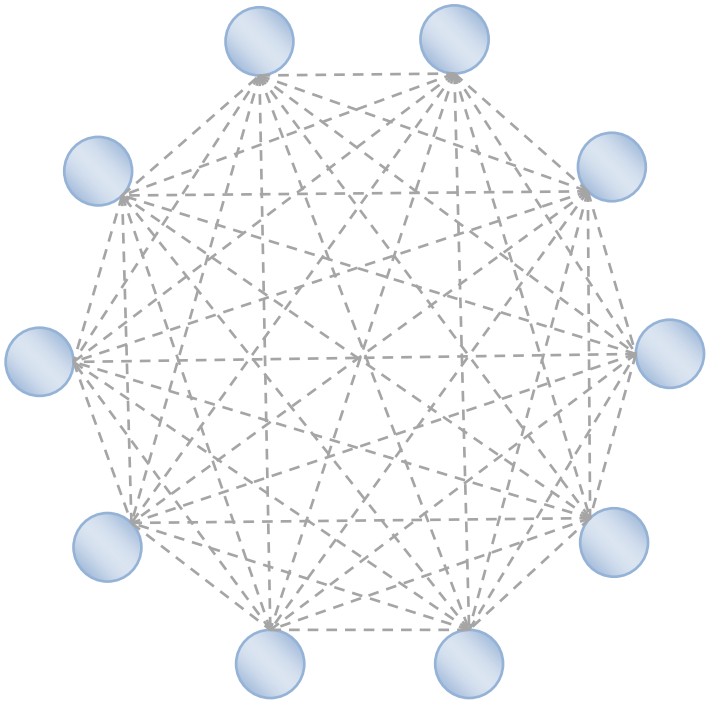

Figura 4. Interazione tra i partecipanti nel caso pessimo

Un sistema informatico offre all'organizzatore di un evento uno strumento potente per semplificare la gestione della fase di organizzazione; infatti, attraverso l'utilizzo di un sistema informatico:

- la comunicazione viene ridotta all'essenziale;
- le informazioni relative agli eventi e ai partecipanti sono conservati in una banca dati affidabile e accessibile in ogni momento;
- si possono offrire informazioni e funzionalità garantite dall'integrazione dello stesso sistema con altri servizi;

- alcuni passi della procedura di organizzazione possono essere automatizzati.

Nell'esempio dell'organizzazione di una partita di calcio a cinque, il promotore contatta il sistema per creare l'evento: durante la fase di creazione, il promotore specifica anche alcuni dettagli per la partita, come giorno ed ora, l'indirizzo del campo, etc. Il sistema stesso provvede a informare tutti i giocatori convocati, i quali confermano o negano la propria presenza al sistema stesso. Il promotore e i partecipanti possono visualizzare le presenze per conoscere in ogni momento l'avanzamento della fase di organizzazione. Se il numero di presenze è eguale a dieci, il sistema informa promotore e partecipanti che la partita avrà luogo il giorno e ora specificati. L'interazione tra gli utenti è ridotta all'essenziale, come si può vedere anche in Figura 5.

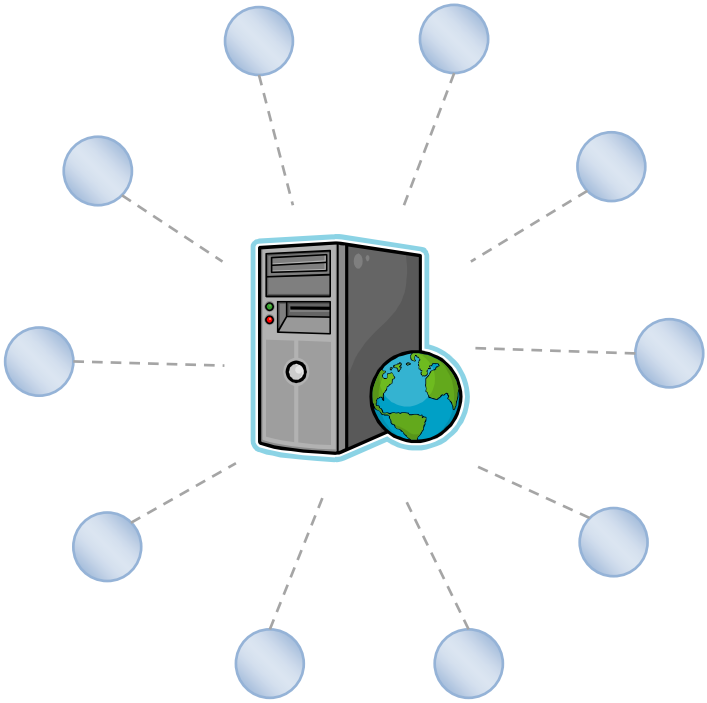

Figura 5. Interazione tra i partecipanti tramite un sistema informatico

Sulla rete Internet esistono diversi servizi di cui il promotore dell'evento può usufruire per migliorare la qualità delle informazioni che fornisce ai partecipanti. Riprendendo l'esempio di organizzazione di una partita di calcio a cinque, il promotore può associare all'evento un link al servizio Google Maps che è un segnalibro della posizione geografica del campo, allo scopo di fornire più informazioni su come raggiungerlo.

Alcuni sistemi informatici, ad esempio i sistemi ad agenti, possono rendere automatiche alcune procedure. Nel nostro esempio, non appena il conteggio delle adesioni effettuato dal sistema raggiunge la soglia di valore pari a dieci, il sistema comunica in modo automatico al promotore e ai partecipanti che la partita può essere giocata.

In particolare, supponendo che gli agenti rappresentino i partecipanti, questi possono rendersi utili per realizzare lo scopo comune: portare a termine l'organizzazione della partita. Se la soglia dei giocatori non è stata ancora raggiunta, e mancano ormai poche ore al giorno e all'ora prestabiliti, gli agenti che rappresentano i partecipanti contattano altri agenti del sistema nell'intento di coprire i posti mancanti.

L'organizzazione di eventi, come abbiamo detto, è una funzionalità comune tra i social netwok, tuttavia esistono delle comunità web dedicate; tra di esse, alcune si integrano con i social network più diffusi, come Facebook o Twitter [13]. Anyvite [14] permette di organizzare eventi scegliendo il design del biglietto d'invito, e impostando un titolo, una descrizione, una locazione, un'ora e un luogo in cui si svolgerà l'evento; una volta creato, l'invito può essere spedito tramite posta elettronica o messaggi brevi (SMS). Anyvite offre anche altre opzioni di gestione: ad esempio, per ogni evento si può stabilire se gli invitati possono essere accompagnati da una o più persone, se la lista degli invitati è pubblica, quando deve essere spedito un promemoria agli invitati, etc.; dopo che l'evento ha avuto luogo, si possono aggiungere contenuti multimediali riferiti ad esso, come foto e video. Evite[15], Pingg [16], Meeting Wizard [17] o MyPunchBowl [18] offrono dei servizi analoghi al precedente seppure con qualche differenza; ad esempio, Pingg permette di spedire gli inviti anche su carta stampata; MeetingWizard, che ha un tono più formale, è dedicato specificatamente all'organizzazione di meeting aziendali, mentre MyPunchBowl è stato ideato principalmente per organizzare feste o eventi informali.

Le due comunità MeetingWizard e MyPunchBowl introducono un concetto interessante: la possibilità di negoziare la data finale dell'evento tramite sondaggio pubblico, che è il punto di forza del sistema di pianificazione online di eventi Doodle [19]. In Figura 6 è mostrata la pagina di amministrazione per un evento creato in Doodle; come si può vedere, il sondaggio tra diverse proposte per la decisione di una data finale dell'evento è esposto in primo piano nella pagina.

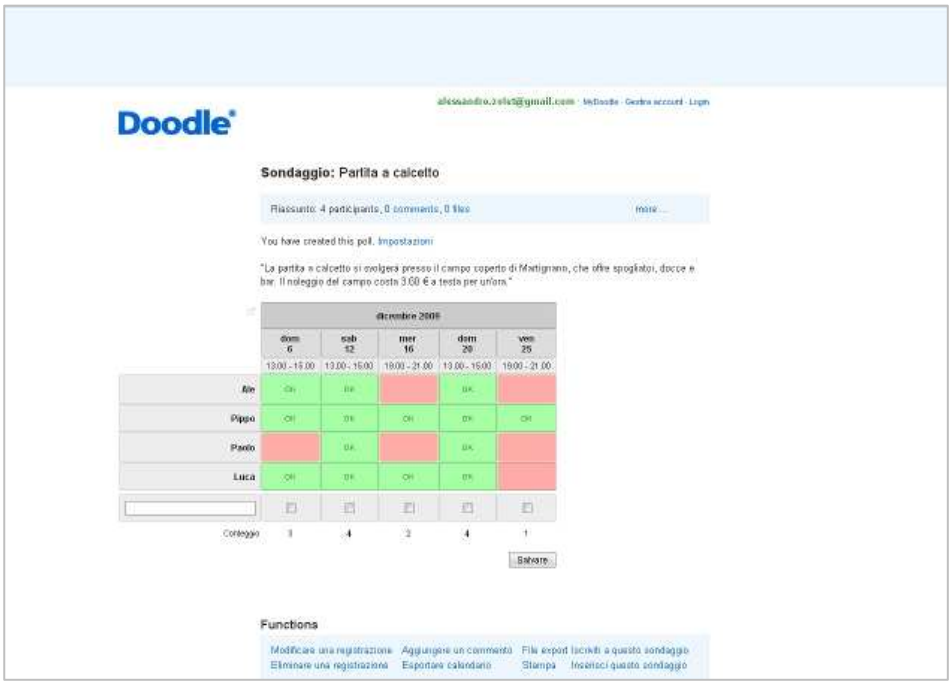

Figura 6. La pagina di amministrazione di un evento creato su doodle.com

Doodle è la nostra maggiore fonte d'ispirazione per ciò che riguarda l'implementazione, perciò analizzeremo il servizio più in dettaglio.

La fase di registrazione è facoltativa, tuttavia indispensabile se si desiderano ricevere aggiornamenti tramite la posta elettronica sulle preferenze espresse dagli invitati riguardo le date proposte. Con Doodle è possibile creare eventi, per cui è necessario fornire alcuni dettagli (titolo e descrizione dell'evento) e stabilire un insieme di date in cui l'evento può avere luogo. Il promotore può invitare chi desidera semplicemente inviando in un messaggio di posta elettronica il collegamento ipertestuale che identifica in modo univoco la pagina di partecipazione dell'evento (*Participation Link*); gli indirizzi e-mail degli invitati possono essere importati direttamente in Doodle da alcuni client di posta elettronica o da Gmail. Ogni invitato che accede alla pagina di partecipazione può esprimere le proprie preferenze tra le date proposte. I partecipanti possono visualizzare le preferenze altrui.

L'utilizzo di Doodle è intuitivo, poiché l'interfaccia utente è essenziale. Inoltre, Doodle offre la possibilità di utilizzare il servizio gratuitamente, e senza sottoporsi a complesse procedure di registrazione. L'utente che utilizza la versione a pagamento di Doodle ha accesso ad alcuni privilegi, come l'assenza di *banner* pubblicitari o l'accesso crittografato ai dati personali. Gli eventi organizzati con Doodle possono essere pubblicati anche sul profilo utente di alcuni social network, come Facebook. Infine, Doodle offre la possibilità di visitare le proprie pagine in versione *mobile*: si tratta semplicemente di una versione ridotta delle pagine, in termini di byte, rispetto alla versione visualizzata in un comune browser.

Diarised [20] offre un servizio analogo a quello di Doodle, più l'automatismo nella scelta della data finale: viene selezionata la data con il maggiore numero di adesioni tra quelle proposte. Zoji [21] è una comunità per la pianificazione di eventi che offre un servizio simile ai precedenti, con la differenza che gli utenti devono appartenere a dei gruppi per creare eventi; Zoji fornisce ai suoi utenti alcune funzionalità comuni tra i social network, che però in questo contesto risultano più dispersive che efficaci.

Le piattaforme di cui abbiamo parlato sopra, i social network e i servizi dedicati alla pianificazione di eventi, presentano innanzitutto una carenza di mezzi di comunicazione. Nei social network solitamente la comunicazione è gestita dalla comunità stessa tramite un servizio interno di messaggistica; probabilmente, il motivo è che i gestori dei social network desiderano che gli utenti navighino il più possibile tra le pagine della comunità per avere maggiori incassi con i banner pubblicitari. Le piattaforme dedicate spesso non offrono alcun servizio interno di messaggistica, e si appoggiano alla posta elettronica. In alcuni casi, le piattaforme dedicate offrono agli utenti la possibilità di comunicare tramite messaggi di testo brevi (SMS), ma il servizio naturalmente è a pagamento.

In effetti, il solo supporto che - non tutte - le piattaforme dedicate offrono ai dispositivi mobili è la visualizzazione di una versione più leggera delle pagine della community, diversamente da quanto accade ad esempio per Facebook, i cui sviluppatori hanno realizzato dei veri e propri applicativi client che si connettono direttamente alla piattaforma per trasmettere i dati. Il ridimensionamento della pagina può essere implementato in questo modo: il server HTML che riceve la richiesta di un browser web, analizza il browser stesso per ricavare informazioni sul dispositivo utilizzato; se questo è identificato come un dispositivo mobile, il server associa alla pagina, prima di trasmetterla, un foglio di stile che ne riduce le dimensioni in termini di esposizione e di contenuti; così il browser del dispositivo mobile evita di scaricare immagini di grandi dimensioni o filmati, mentre la struttura della pagina stessa è ridotta all'essenziale.

Una piattaforma realmente integrato con i dispositivi mobili dovrebbe fornire ad essi un applicativo client in grado di scambiare dati direttamente con il sistema. In questo modo, tra l'altro, si evita lo *overhead* dovuto al codice HTML perché il dispositivo mobile riceve soltanto i dati cui è interessato.

La capacità decisionale degli agenti può essere sfruttata durante la pianificazione di eventi, così come per le altre funzionalità tipiche di un social network, per automatizzare determinate procedure. Ad esempio, se tutti i partecipanti hanno espresso la propria preferenza su una possibile data in cui si deve svolgere un evento, l'organizzatore seleziona come data finale quella con il maggiore numero di preferenze ricevute: di conseguenza, l'agente che svolge il ruolo di organizzatore potrebbe selezionare la data finale in automatico e inviare la conferma della data stessa a tutti gli invitati o almeno ai partecipanti. Riprendendo un esempio fatto in precedenza, supponiamo che un evento richieda una quota minima di partecipanti per avere luogo, e che al giungere di una scadenza prefissata, la quota non sia stata raggiunta; una soluzione può essere quella di chiedere ai partecipanti di invitare i propri conoscenti che sono disponibili: l'agente che

rappresenta il promotore invia ai partecipanti una richiesta di estendere l'invito ai propri contatti. Come ulteriore esempio di automatismo, supponiamo che a ciascun evento sia associato un valore che rappresenta la priorità dell'evento stesso; quando un utente riceve un invito per un evento a bassa priorità in un orario in cui ha già un altro impegno con priorità maggiore, l'agente rifiuta immediatamente l'invito.

Infine, supponiamo che diversi campi da calcio a cinque siano disponibili per il giorno e l'orario prestabiliti per la prossima partita; un partecipante può desiderare di convincere gli altri a giocare la partita nel campo a lui più vicino, oppure nel campo più economico. Per fare ciò, l'agente che rappresenta quell'utente può avviare una negoziazione con gli agenti rappresentanti gli altri utenti sui termini stabiliti, al fine di raggiungere il proprio scopo, che è lo stesso che l'utente si è prefissato: minimizzare le spese. Gli agenti, come detto in precedenza, possono collaborare, competere o negoziare con gli altri agenti per raggiungere il proprio obiettivo.

1.5 Conclusioni

Allo stato dell'arte attuale, non esiste alcun social network *intelligente*, ossia una piattaforma in cui ogni utente sia rappresentato da un agente che opera attivamente o reattivamente per raggiungere gli obiettivi dell'utente stesso, quindi abbiamo deciso di costruire un prototipo in quest'ottica.

Considerata inoltre l'importanza dell'utenza mobile sul mercato, non abbiamo potuto trascurare questo aspetto: una parte del prototipo sviluppato è rivolta ai cellulari di ultima generazione, in particolare ai disposivi che installano il sistema operativo Google Android.

In conclusione, considerando il carico di lavoro necessario per sviluppare un intero social network, abbiamo preso in esame il caso specifico della pianificazione di eventi.

Capitolo 2
Tecnologie per sistemi ad agenti rivolti ai dispositivi mobili

Di seguito presentiamo le tecnologie JADE, JADE-LEAP, Google Android e JADE-LEAP for Android, esponendo la graduale estensione di JADE fino all'integrazione con i telefoni cellulari di ultima generazione.

2.1 Il modello FIPA

Nella presente sezione introduciamo il modello di piattaforma ad agenti proposto da Foundation for Intelligent, Physical Agents (FIPA).

La comunicazione proposta dal modello FIPA si basa sulla teoria degli atti linguistici.

FIPA offre un layer diviso in sub-layer:

- trasporto: IIOP o TCP;
- encoding: XML, String, Bit;
- messaging: struttura con envelope e payload con parametri obbligatori;
- ontology;
- content: il contenuto può essere di qualunque tipo anche se è preferibile una rappresentazione in formule logiche;
- comunicative act: una classificazione del messaggio in base al suo scopo, informativo, richiesta, etc.;
- interaction protocol: riguarda le implicazioni del messaggio sull'agente e sull'interazione con altri agenti; ad esempio, una richiesta (un messaggio di tipo request) potrebbe essere risolta con la compartecipazione di più di un agente - non solo di quello che la ha ricevuta.

JADE [22] è una piattaforma ad agenti implementata da Telecom Italia, fedele al modello proposto da FIPA; in seguito parleremo di JADE in dettaglio. Il modello FIPA è riportato in Figura 9.

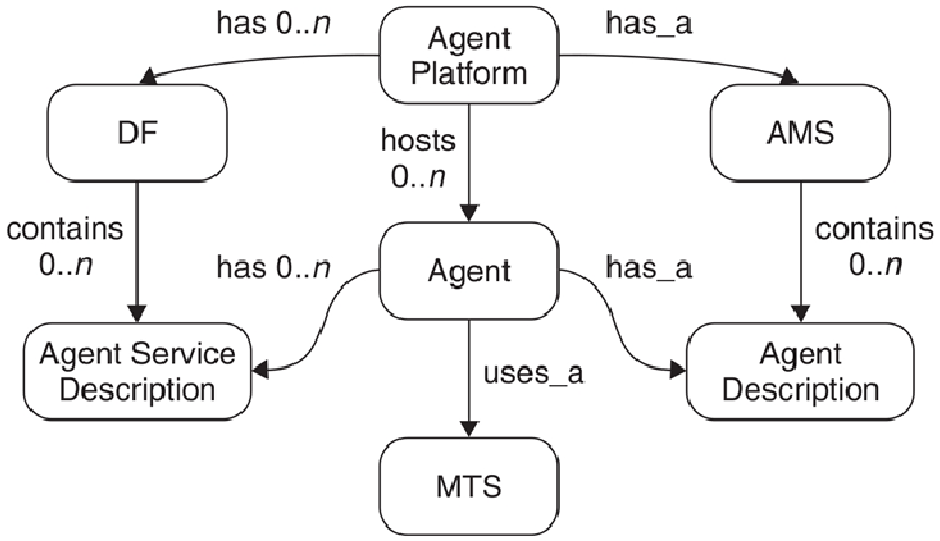

Figura 7. Il modello FIPA

Gli agenti vengono dispiegati nella piattaforma, la quale è un concetto astratto, pertanto può essere fisicamente dislocata su più macchine con diversi sistemi operativi. Un agente è un'entità che offre un servizio, e qualunque esso sia deve rispettare soltanto il requisito di avere un padrone (piattaforma, o un contenitore come vedremo nel caso di JADE), essere identificabile con un Agent Identifier (AID) univoco ed un indirizzo di trasporto a cui può essere raggiunto.

Il Directory Facilitator (DF) è un componente opzionale di una piattaforma ad agenti che offre la possibilità di effettuare una ricerca degli agenti registrati per tipologia di servizio. Uno o più DF sono solitamente presenti all'interno della piattaforma, JADE ne istanzia uno di default. Un agente richiede la registrazione al DF nel caso in cui desideri rendere noto il proprio servizio; allo stesso modo, un agente può anche richiedere la modifica dei parametri di registrazione o la deregistrazione. La ricerca di un servizio viene effettuata in qualunque momento secondo dei parametri che vengono confrontati con quelli degli agenti registrati.

Lo Agent Management System (AMS) è un componente necessario alla piattaforma che serve a svolgere tutte le operazioni di gestione degli agenti: creazione, cancellazione, migrazione (da e verso la piattaforma). Inoltre, AMS tiene traccia perciò di tutti gli agenti attivi sulla piattaforma, assegnando a ciascuno un identificatore univoco (AID) che può essere riassegnato nel caso in cui l'agente che lo possedeva sia terminato.

La piattaforma infine possiede un sistema di messaggistica che permette agli agenti di comunicare con altri agenti appartenenti alla stessa piattaforma o anche a piattaforme diverse. Il componente che implementa questo meccanismo è detto Message Transport Service (MTS).

I componenti sopra descritti possono essere implementati come agenti della piattaforma stessa, come avviene in JADE, anche se si differenziano dai normali agenti per determinate caratteristiche, oltre che per le funzionalità specifiche: ad esempio, i componenti AMS e DF sono costretti a rispondere ad un'interrogazione.

In ogni caso, i servizi che questi componenti devono offrire sono:

- servizio di trasporto dei messaggi, per lo scambio di messaggi tra gli agenti;
- servizio di registro degli agenti, ossia un repository in cui gli agenti registrano le proprie informazioni;
- servizio di registro dei servizi stessi, ossia un repository in cui agenti e servizi possono registrare e trovare servizi.

Un messaggio FIPA è strutturato come in Figura 8.

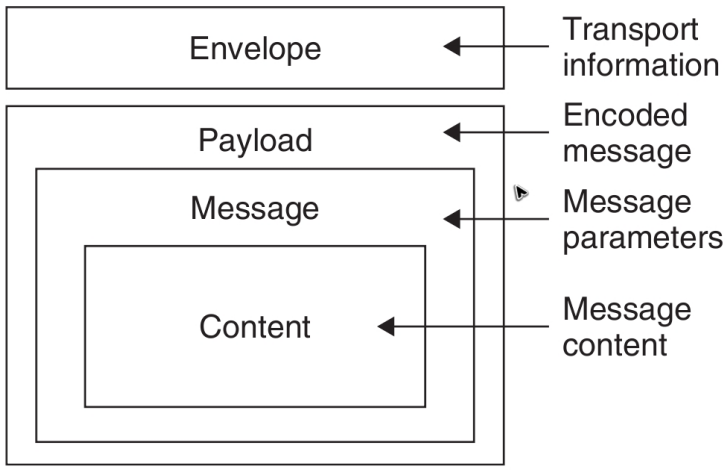

Figura 8. Struttura di un messaggio FIPA

All'interno del messaggio vi sono diversi parametri, dei quali soltanto *performative,* che specifica la tipologia di atto comunicativo del messaggio stesso, è obbligatorio. I restanti parametri sono facoltativi (anche se alcuni di essi sono utilizzati praticamente in ogni caso pratico). Ecco un elenco dei parametri:

- performative: l'atto comunicativo del messaggio;
- sender: il mittente del messaggio;
- receiver: il ricevente del messaggio;
- reply-to: l'agente destinato a ricevere i messaggi di una particolare conversazione;
- content: il contenuto del messaggio;
- language: il linguaggio in cui il contenuto viene espresso;
- encoding: encoding del messaggio (XML, stringa, etc.);
- ontology: il riferimento ad un'ontologia che dà significato ad eventuali simboli all'interno del contenuto;

- protocol: il protocollo d'interazione;
- conversation-id: un identificatore della conversazione;
- reply-with: l'espressione utilizzata da un agente ricevente per identificare il messaggio;
- in-reply-to: per i messaggi di risposta, è un riferimento alla precedente azione;
- reply-by: un timestamp dell'ora in cui il messaggio doveva essere ricevuto.

Diamo uno sguardo in dettaglio alle *performative*: si tratta di espressioni volte a definire l'atto comunicativo, ossia l'aspetto interpretativo che ci si aspetta che sia dato ad un messaggio, in altre parole l'effetto che si spera di ottenere inviando un messaggio; in qualche modo, l'atto comunicativo rappresenta la semantica del messaggio. Esistono diversi tipi di atto comunicativo, derivanti dalla teoria degli atti linguistici, che si possono riassumere in alcune categorie non esclusive:

- interrogative per la richiesta di informazioni;
- exercitive direttive per la richiesta di eseguire un'azione;
- referential per la condivisione di informazioni riguardo l'ambiente circostante;
- phatic per la richiesta di stabilire, prolungare o interrompere una conversazione;
- paralinguistic ovvero l'espressione di una relazione tra diversi messaggi;
- expressive ovvero asserzioni che esprimono obiettivi, convinzioni e atteggiamenti.

Gli atti comunicativi sono molti; di seguito, riportiamo a titolo di esempio alcuni tra gli atti più comuni e le loro rispettive categorie:

- inform: enuncia la veridicità di una proposizione, ovvero riporta una informazione; può appartenere sia alla categoria referential, che a expressive, perché i dati riportati da questo tipo di messaggio possono arricchire la conoscenza dell'ambiente così come contenere altri tipi di informazione;
- request: chiede al ricevente di fare qualcosa, di svolgere un'azione, perciò è chiaramente di tipo exercitive.

Il contenuto del messaggio può essere espresso liberamente come FIPA-SL tramite logica modale, o in qualsiasi altro modo.

FIPA Request Interaction Protocol Specification invece è un protocollo utilizzato per definire come avviene l'interazione tra due agenti; uno di essi, detto *Initiator*, effettua una richiesta ad un altro di eseguire una determinata azione. L'agente interrogato, detto *Participant*, può decidere se accettare o rifiutare; nel caso in cui accetti (cosa che non sempre è costretto a comunicare), esegue l'azione prescritta e risponde al primo con un messaggio d'informazione che può essere *failure*, se non è riuscito a portare a termine l'azione, *inform-done*, se deve informare che è andato tutto a buon fine, *inform-result*, se deve informare che l'azione è stata eseguita e ha anche un risultato da consegnare. Il messaggio *not-understood* è utilizzato nel caso in cui l'agente interrogato non abbia compreso il significato della richiesta che gli è stata inviata.

Nel caso in cui l'agente Initiator desideri annullare la richiesta, può farlo avviando il protocollo di *FIPA Cancel Meta Protocol*, in cui l'iniziatore non è più interessato ai risultati di una precedente richiesta perciò chiede l'annullamento della conversazione (identificata con un particolare id), e può ricevere una risposta affermativa (inform) o negativa (failure).

Due o più agenti possono inoltre iniziare un negoziato seguendo il *Contract Net Interaction Protocol*; secondo le specifiche, l'agente Initiator contatta un agente Participant per richiedere a questo di eseguire una determinata funzione, tuttavia lo fa specificando che alcuni parametri devono essere ottimizzati. Per fare questo l'agente Initiator deve innanzitutto inviare a ciascun agente Participant un messaggio di *call for proposa*l con annessi parametri (costo, tempo, etc.), per i quali desidera che la sua richiesta sia presa in considerazione; alcuni partecipanti rifiutano la proposta, quindi l'iniziatore riceve da essi un messaggio di rifiuto (refuse), altri rispondono con un messaggio di proposta (propose). In questo secondo caso, l'iniziatore a propria volta può rifiutare o accettare la controproposta; se la accetta, dopo aver ricevuto un messaggio accept-proposal, il partecipante esegue la funzionalità richiesta ed informerà l'iniziatore del fallimento o del successo dell'operazione come nel caso di una richiesta (request).

Il servizio che offre alla piattaforma ad agenti la funzionalità di comunicazione tramite scambio di messaggi ACL (Agent Communication Language) è detto *Message Transport Service* (MTS), ed è implementato tramite un Agent Communication Channel (ACC). Ogni messaggio è costituito di un'intestazione (envelope) in cui sono memorizzate le coppie (parametro, valore) per il mittente, il ricevente, la data, la tipologia di atto comunicativo, ed altri parametri facoltativi: questi dati - e solo questi - sono letti da ACC, che provvede ad inoltrare il messaggio alla piattaforma corretta. All'interno del recipiente del messaggio sono contenuti gli identificatori univoci (AID) degli agenti destinatari del messaggio stesso, perciò ad ACC è sufficiente estrapolare le informazioni riguardanti l'indirizzo di trasporto da questi identificatori a quale agente o piattaforma deve essere spedito.

2.2 JADE

JADE [22] implementa il modello architetturale proposto da FIPA, per quanto riguarda i suoi componenti, il sistema di comunicazione ed i protocolli di interazione tra gli agenti; ciò che non è implementato può essere ricavato secondo le necessità di sviluppo attraverso estensioni del framework esistente. Inoltre, JADE offre diverse funzionalità aggiuntive tra cui vale la pena citare un'architettura distribuita su *container* che garantisce robustezza, sicurezza, supporta la mobilità degli agenti (cioè la migrazione degli agenti verso altre piattaforme) e l'interazione tra agenti e web service (attraverso un plugin dedicato).

La politica *open source* si è rivelata importante per ciò che riguarda dare la possibilità agli sviluppatori più volonterosi di fornire nuovo materiale ed estensioni; per questo motivo è stata assunta anche dagli autori di JADE.

Seguono alcune caratteristiche di JADE.

- Ogni agente viene eseguito da JADE in un *thread* separato. Al fine di risparmiare risorse, viste le necessità tipiche dell'ambiente mobile, si associa ad ogni comportamento assunto da un agente un semplice oggetto; ciò porta ovviamente a delle conseguenze: ad esempio, quando viene eseguita l'azione di un behaviour essa non può essere interrotta, finché non termina; il ciclo successivo, a meno che uno specifico metodo di controllo non ritorni uno stato necessario a terminare il behaviour, viene intrapreso soltanto dopo aver eseguito l'azione di un altro behaviour, se ne esistono altri associati allo stesso agente.
- La comunicazione avviene in modo asincrono, e all'agente mittente è sufficiente conoscere soltanto l'indirizzo di trasporto dell'agente destinatario.
- Gli agenti che vengono eseguiti in una piattaforma JADE sono da considerarsi dei *peer*. JADE è un framework distribuito, in grado di eseguire diversi agenti che comunicano fra loro in trasparenza.
- JADE presenta completa compatibilità con il modello presentato da FIPA.
- JADE fornisce un servizio di trasporto di messaggi asincroni, attraverso delle API, che selezionano in modo trasparente il mezzo di comunicazione più efficiente.

La piattaforma ad agenti JADE è costituita da oggetti speciali detti *container*, che sono dislocati su differenti Java Virtual Machine (JVM), ossia su differenti macchine: dentro ogni container sono eseguiti gli agenti.

Tra i container, vi è un *main container*, in cui vengono eseguiti anche AMS e DF, che forniscono le funzionalità di registrazione degli agenti e ricerca dei servizi offerti; il main container viene creato per primo, e gli altri container sono costretti a registrarsi ad esso per rendersi parte della stessa piattaforma. Il main container apparentemente potrebbe sembrare il collo di bottiglia del modello, ma il traffico viene indirizzato ai rispettivi container in cui sono dislocati gli agenti in conversazione, e gli indirizzi dei container sono memorizzati in una cache mantenuta da ciascun container (LADT). Nel caso in cui la cache non possieda le informazioni necessarie, scartate in seguito all'assunzione di altre informazioni secondo la politica Least Recently Used (LRU), esse vengono acquisite dal main container. Il main container risulta comunque essere il punto debole del sistema: nel caso in cui venga messo giù, la piattaforma non è più in grado di funzionare correttamente. Questo problema può essere risolto con la tecnica di Main Replication Service.

Un identificatore AID identifica un agente in modo univoco; esso è costituito dal nome dell'agente (la concatenazione di un appellativo univoco e del nome della piattaforma ospite), utilizzato per la disambiguazione intrapiattaforma, e l'indirizzo dell'agente, ossia l'indirizzo di trasporto ereditato dalla piattaforma ed utilizzato per la ricezione di messaggi.

JADE è una piattaforma scritta in Java. Ogni agente estende la classe `jade.core.Agent`. Ogni agente può avere diversi comportamenti, eseguiti in sequenza o in parallelo, in uno o più passi oppure ciclici; ogni comportamento estende la classe `Behaviour`, o più spesso una delle sue sottoclassi. Ad esempio, per eseguire un comportamento ciclico è sufficiente implementare una istanza della classe `CyclicBehaviour` (o anche estendere la classe stessa); invece, per eseguire un comportamento che svolge una singola azione è sufficiente implementare `OneShotBehaviour`. Un'azione, non importa se essa sia singola o ripetuta, è implementata dalla funzione `action` di `Behaviour`.

In JADE, anche i servizi DF e AMS sono implementati come degli agenti speciali, come spiegato in precedenza. In più, per facilitare la gestione della piattaforma esistono alcuni agenti speciali: ad esempio, un agente che offre un'interfaccia grafica per la gestione della piattaforma (RMA), che visualizza i container che compongono la piattaforma stessa, gli agenti in esecuzione nei container, il registro del Directory Facilitator ed altre informazioni; un agente utilizzato per lo sniffing dei messaggi (Sniffer).

JADE implementa tutti i protocolli MTP standard definiti da FIPA; in questo modo permette tra l'altro la comunicazione inter-piattaforma. Da un punto di vista tecnico, viene aperta una *socket* HTTP in ricezione, cioè un indirizzo costituito di protocollo utilizzato, macchina e porta che identifica il servizio (come `http://localhost:1099`), sul main container per la ricezione di tutti i messaggi che sono poi destinati (solitamente) agli agenti dislocati sulla piattaforma. Il protocollo utilizzato per la comunicazione interna alla piattaforma è detto Internal Message Transport Procotol (IMTP). La tabella di routing utilizzata per l'inoltro dei messaggi è single-hop.

IMTP (protocollo di trasporto di messaggi interni) è utilizzato per lo scambio di messaggi fra container che appartengono alla stessa piattaforma. Non necessitando di interagire con l'esterno, e di essere perciò compatibile con le specifiche FIPA, IMTP è stato progettato come un protocollo proprietario, ed è stato ottimizzato per l'utilizzo interno alla piattaforma, ed esteso a funzionalità non legate alla messaggistica, come ad esempio il monitoraggio delle condizioni di vita di un container remoto. Esistono due varianti di IMTP, basate su RMI e su socket TCP.

Il contenuto di un messaggio contiene informazioni riguardo l'ambiente in cui gli agenti operano; un'ontologia definisce proprio l'ambiente specifico in cui gli agenti sono calati, una realtà che viene definita tramite una serie di predicati, concetti ed azioni che gli agenti stessi intraprendono se richiesto. Ci sono vari mezzi per passare gli oggetti: JADE NON usa la serializzazione Java ma piuttosto il linguaggio XML.

Per definire un'ontologia bisogna innanzitutto definire concetti, predicati ed azioni. Ad esempio, dei concetti sono *utente*, *libro*. Dei possibili predicati per un utente sono *nome*, *cognome*, *indirizzo*, *numero di telefono*, oppure *titolo*, *autore*, *casa editrice* per un libro; in altre parole i predicati sono le proprietà caratteristiche di un concetto. Delle azioni possibili sono *vende*, *acquista* e hanno dei concetti per soggetti e per oggetti: *un utente vende un libro*.

JADE offre un sussidio per la creazione di ontologie attraverso l'implementazione di alcune classi relazionate a `jade.content.onto.Ontology`.

2.3 Google Android

Come gli autori lo definiscono sulla pagina principale della sezione dedicata agli sviluppatori

> "*Android è la prima piattaforma mobile gratuita, open-source ed assolutamente personalizzabile. Android offre uno stack completo di sistema operativo, middleware, e applicazioni chiave. Inoltre, Android contiene un vasto insieme di API che permettono agli sviluppatori di terze parti di realizzare le proprie applicazioni.*" (Android developers)

Android [23] è il primo obiettivo di Open Handset Alliance [24], un gruppo di grandi aziende che operano nei campi dell'industria manifatturiera di dispositivi mobili (come Acer, Asus, LG, HTC, Toshiba) e di componenti hardware (come Intel), software house (come eBay Inc. e naturalmente Google Inc.), compagnie telefoniche (tra cui Telecom Italia, Vodafone). Attualmente, le compagnie che fanno parte di Open Handset Alliance sono quarantasette.

L'obiettivo è quello di realizzare uno standard aperto, che offra un'alternativa ai sistemi proprietari, e pertanto estende il kernel Linux (esattamente come per i sistemi desktop o laptop, questa è l'alternativa), offrendo la strada dell'innovazione agli sviluppatori più operosi, che devono semplicemente fare pratica con le API offerte dal pacchetto Android SDK e quindi possono utilizzare qualunque parte hardware del dispositivo mobile (bluetooth, camera, etc.) e qualunque applicazione a basso livello (messaggi, chiamate, rubrica, etc.) direttamente dalla propria applicazione. L'unico limite è la fantasia degli sviluppatori. L'insieme di sviluppatori cui si rivolge Android è tutto il mondo, e per incrementare l'interesse Google ha organizzato dei *contest* con ricchi premi. Ad oggi, sul sito *Androidlib.com* esistono più di diecimila applicazioni [25] (gratuite e a pagamento) sviluppate da terze parti.

Tutte le applicazioni, sia quelle proprie del sistema sia quelle create da terze parti, hanno le stesse capacità di accesso alle funzionalità offerte da Android stesso.

Il seguente diagramma riporta i componenti principali del sistema operativo Android.

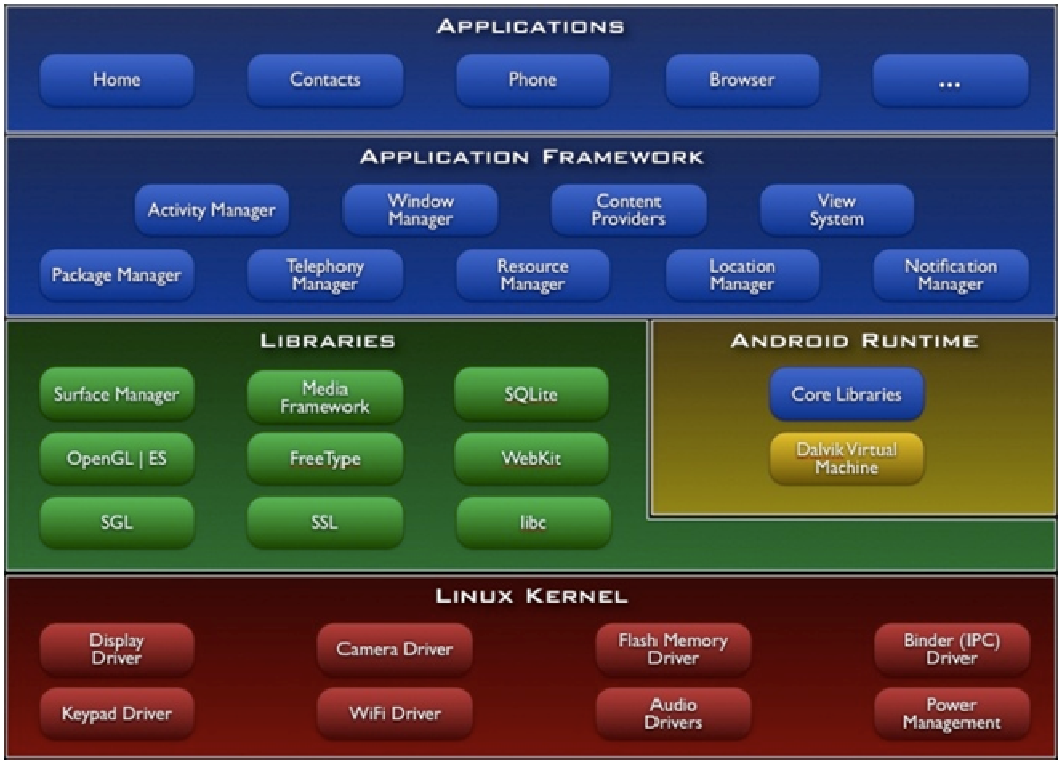

Figura 9. Lo stack di Android

Partendo dal livello più basso della *pila* (o *stack*), troviamo il cuore del sistema operativo, basato sul Linux Kernel. Esso fornisce servizi di sistema come sicurezza, gestione della memoria, gestione dei processi, pila dei protocolli di rete, l'architettura dei driver: in altre parole, il kernel costituisce lo strato di comunicazione tra l'hardware incorporato nel dispositivo portatile e i livelli più alti della pila. Il kernel di Linux è stato scelto per essere *open source*, oltre che per la sua solidità.

Al livello superiore giacciono le librerie, scritte in C/C++ ed utilizzate dai diversi componenti del sistema. Lo sviluppatore non vi accede direttamente, ma attraverso il livello superiore (Application Framework). Alcune tra le librerie più importanti sono:

- System C Library: un'implementazione della libreria standard di sistema *libc*, adattata per dispositivi integrati basati su Linux;
- Media Libraries: una libreria per il supporto e la riproduzione dei formati audio e video più popolari, tra cui MPEG4, H.264, MP3, AAC, AMR, JPG e PNG;
- SQLite: un motore di basi di dati relazionali potente e leggero.

Le librerie sono affiancate ad un'implementazione specifica della Java Virtual Machine (JVM), chiamata Dalvik. Le applicazioni sono scritte in Java, perciò l'esistenza di Dalvik non è un fattore trascurabile per il successo di Android. Le ragioni per cui i creatori di Android hanno preferito implementare una propria JVM piuttosto che utilizzare un'implementazione standard come quella di Sun, sono le seguenti: per prima cosa, ogni applicazione è eseguita da Android in un processo singolo che possiede la propria

istanza di Virtual Machine; in secondo luogo, Dalvik ottimizza il consumo della memoria - una risorsa sempre limitata sui dispositivi portatili - attraverso un compilatore Java specifico che trasforma i sorgenti in file con un formato ottimale con suffisso .dex, anziché in file .class.

Gli sviluppatori hanno accesso alle stesse librerie utilizzate dalle applicazioni di sistema. L'architettura delle applicazioni è stata progettata in modo da semplificare il riutilizzo dei componenti; ogni applicazione può pubblicare le proprie funzionalità, ed ogni altra applicazione può fare uso di queste capacità. Lo stesso meccanismo permette che i componenti vengano sostituiti dall'utente. Tra i servizi di sistema ricordiamo:

- un insieme di elementi di visualizzazione (`View`) che possono essere utilizzati per costruire l'interfaccia utente di un'applicazione, e pertanto comprendono esposizione dei contenuti a lista, a griglia, ma anche campi di testo, pulsanti, e perfino un web browser integrabile;
- dei Content Provider, che permettono alle applicazioni di accedere a dei dati da altre applicazioni (l'oggetto `Contacts` ne è un esempio) o di condividere i propri dati;
- un Resource Manager, che fornisce l'accesso a risorse che non siano codice, come ad esempio stringhe, layout (esposizione dei componenti grafici a schermo);
- un Notification Manager che abilita tutte le applicazioni a mostrare dei messaggi personalizzati nella barra di stato;
- un Activity Manager che gestisce il ciclo di vita delle applicazioni e fornisce una pila di navigazione tra di esse.

Un'applicazione Android è contenuta in un file di formato *.apk*, utilizzato di fatto per l'installazione sul dispositivo, che comprende binari e risorse utilizzate dall'applicazione stessa. Un'applicazione Android viene solitamente eseguita in un processo separato dagli altri; tuttavia, in caso che l'implementazione lo richieda, è possibile eseguire più applicazioni in un unico processo, come thread sulla stessa virtual machine che viene condivisa esattamente come le risorse delle applicazioni stesse. Va precisato che è possibile passare facilmente da un'applicazione Android all'altra, perché di fatto Android è un sistema operativo multitasking. Solitamente invece un'applicazione viene eseguita in un processo Linux, che inizia l'esecuzione nel momento in cui l'utente richiede l'esecuzione dell'applicazione e lo uccide nel momento in cui l'applicazione non lo utilizza più, ed il sistema necessita il recupero di risorse; ogni applicazione esegue il proprio codice indipendentemente dalle altre applicazioni su una propria istanza della JVM; ad ogni applicazione è associato un identificativo utente univoco che la rende visibile soltanto all'utente corrente per quell'applicazione stessa. La condivisione di cui si parlava poco fa è possibile proprio grazie all'utilizzo dello stesso identificativo per più applicazioni.

Ogni applicazione è costituita di diverse parti, che svolgono delle funzionalità precise al suo interno; è sufficiente che una soltanto di queste parti sia richiesta perché l'applicazione sia inizializzata o riesumata. Le parti di cui si compone un'applicazione sono:

- *activity*: ogni activity rappresenta un pezzo dell'applicazione che svolge una determinata funzione. Ad esempio, un'applicazione di chiamata potrebbe contenere una activity per visualizzare i tasti del telefono e comporre un numero, un'altra potrebbe avere un elenco di contatti, un'altra potrebbe mostrare la chiamata in corso e la sua durata. Una activity viene creata ed inizializzata ogni qual volta c'è bisogno di svolgere una funzionalità, quindi viene messa in pausa o distrutta nel caso in cui altre attività o applicazioni siano eseguite in primo piano. Ogni activity viene presentata all'utente attraverso una gerarchia di viste cui è associata. La activity che nell'esempio precedente permette di aggiungere un contatto alla rubrica, offre diversi campi di testo che conterranno i dati del futuro nuovo contatto;
- *service*: un'attività che deve essere svolta in background è chiamata service: in sostanza, si tratta di funzionalità che durano nel tempo ma prive di interfaccia grafica;
- *broadcast receiver*: un broadcast receiver è un processo in background che porta delle notifiche alle applicazioni (e alle loro attività); vi sono diversi metodi per notificare all'utente che è avvenuto qualcosa indipendentemente dall'attività in svolgimento (ad esempio è stato ricevuto un messaggio, oppure la batteria si è abbassata oltre la soglia minima): un suono, una vibrazione o una notifica sulla barra di stato;
- *content provider*: mette a disposizione di un'applicazione un insieme di dati appartenenti ad un'altra applicazione; i dati sono conservati ed estrapolati da una sorgente che può essere il file system oppure un database SQLite; in ogni caso, le applicazioni fanno uso di un Content Provider in maniera indiretta, attraverso l'uso di un *Content Resolver*.

Un elemento caratteristico della programmazione in Android è lo *intent*. Uno intent è un messaggio asincrono che serve ad avviare una activity, un service o un broadcast receiver: esso contiene un identificativo univoco dell'oggetto che si desidera istanziare, ed eventuali dati che l'attività chiamante vuole passare alla nuova attività.

Un'applicazione può essere identificata come un *task*. Tutte le attività comprese all'interno di uno stesso task vengono spostate in blocco in testa allo stack nel momento in cui l'applicazione è avviata o riportata in primo piano (foreground); al contrario, nel momento in cui l'applicazione viene chiusa, tutte le attività che compongono il task vengono scaricate dalla memoria; le uniche operazioni che si possono svolgere sullo stack sono dunque *push* e *pop*, ossia caricare un task in cima allo stack e rimuoverlo dalla stessa.

Nel caso in cui l'utente prema il tasto *HOME* dell'emulatore o del dispositivo, il task in primo piano, se ne esiste uno, passa in secondo piano (background) lasciando l'applicazione denominata *home* in testa allo stack; attraverso il menu delle applicazioni, che si apre con il tasto MENU, è possibile riportare l'applicazione precedente di nuovo in foreground.

Nel caso in cui l'utente prema il tasto *BACK* dell'emulatore o del dispositivo, il task viene rimosso dalla memoria. L'applicazione può essere lanciata nuovamente attraverso il menu delle applicazioni.

All'interno di uno stesso task, un'attività passa in primo piano nel momento in cui si trova in testa allo stack. Ciò può accadere nel caso in cui un'altra attività del task ne abbia richiesto l'esecuzione tramite un intent, utilizzando il metodo startActivity() o startActivityForResult(), a seconda che l'attività chiamante si aspetti o meno di ricevere un risultato dalla nuova attività; oppure nel caso in cui l'attività in testa allo stack venga rimossa, utilizzando il metodo finish().

Quanto detto vale anche per i service.

Attraverso l'impostazione di alcune flag specifiche, è possibile modificare il comportamento standard delle attività, ad esempio facendo in modo che una nuova attività sia associata ad un task diverso da quello che la ha generata, oppure ad esempio facendo in modo che sul dispositivo possa essere lanciato una sola volta un particolare task o una particolare attività, o ancora facendo in modo che un intent non generi, come di default, una nuova istanza della classe rappresentante l'attività, se essa si trova in testa allo stack.

Nel nostro caso, sono state utilizzate le impostazioni di default.

Un'applicazione Android normalmente viene eseguita su un singolo thread all'interno di un singolo processo Linux; un thread si identifica con un oggetto Thread in Java. Nel caso in cui un thread debba svolgere operazioni piuttosto lunghe, come scaricare un file da remoto, può lanciare un secondo thread dedicato a svolgere questo compito. Tra le tipologie di classi che Android offre come possibili implementazioni di un thread va citata *Handler*, che, come vedremo in seguito, viene utilizzata per inoltrare messaggi nella comunicazione dall'agente al dispositivo mobile.

Come visto in precedenza, un'attività viene creata e distrutta. Durante il proprio ciclo di vita, essa può trovarsi in tre stati:

- *active*: l'attività si trova in primo piano sullo schermo (ossia in cima allo stack delle attività in esecuzione); in questo stato, l'attività cattura le azioni dell'utente;
- *paused*: l'attività non possiede il focus ma è ancora visibile all'utente, perciò questi non può agire direttamente su di essa, ma essa mantiene il proprio stato attuale di esecuzione;

- *stopped*: l'attività è completamente oscurata da un'altra attività, non risulta visualizzabile all'utente, tuttavia mantiene il proprio stato.

Nei casi in cui l'attività è in stato paused o stopped può essere rimossa dal sistema nel caso in cui esso necessiti di memoria, con la conseguente perdita dello stato di esecuzione in cui si trova. Ogni volta che un'attività cambia il proprio stato, viene invocata una delle seguenti *callback*:

- `onCreate(Bundle saveInstanceState)`: il metodo viene invocato durante l'inizializzazione dell'attività, e deve essere implementato da una sottoclasse di Activity; solitamente, onCreate() associa una visualizzazione (View) all'attività; il parametro contiene lo stato dell'attività chiamante, perciò viene comunemente utilizzato per il passaggio di parametri da un'attività all'altra. Tuttavia, nel caso in cui un'attività debba passare ad un'altra degli oggetti complessi può farlo impostando il valore di una variabile globale statica di questa seconda;
- `onStart()`: l'attività viene inizializzata (oppure in seguito alla chiamata di `onRestart()`); questo metodo determina l'inizio del tempo di visibilità di un'attività, sia che essa si trovi in foreground oppure no;
- `onResume()`: l'attività viene portata in foreground;
- `onPause()`: l'attività viene portata in background, ma è ancora visibile all'utente;
- `onStop()`: l'attività diventa invisibile all'utente;
- `onDestroy()`: questo metodo determina il termine del ciclo di vita di un'attività: essa viene rimossa dallo stack e la porzione di memoria da essa occupata viene liberata dal sistema; pertanto, eventuali processi in esecuzione parallela devono essere fermati nel corso dell'esecuzione di questa callback.

Il ciclo di vita di un'attività si presenta come in Figura 10.

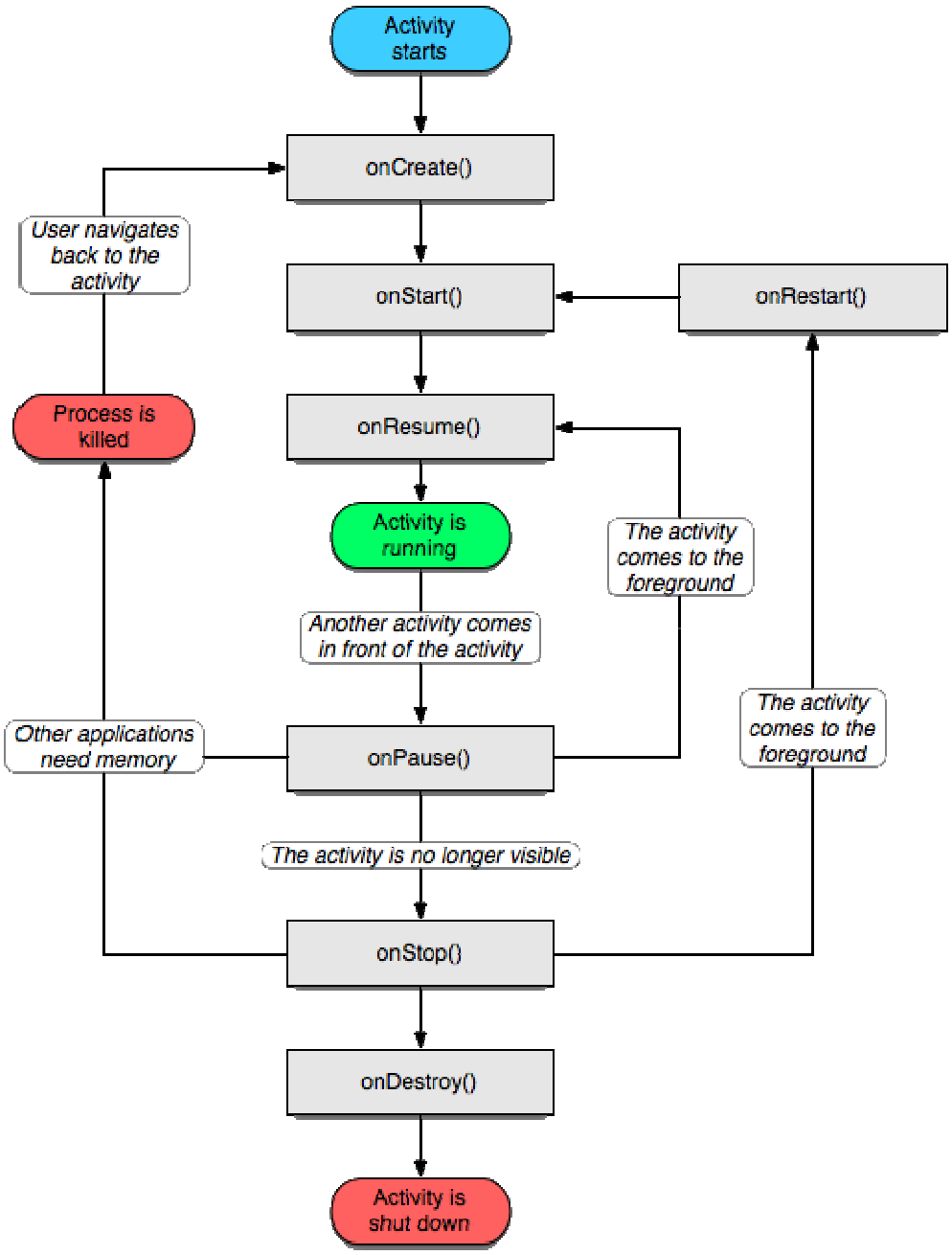

Figura 10. Ciclo di vita di una activity

In maniera del tutto simile a quanto avviene per le attività, il ciclo di vita di un servizio inizia e si conclude rispettivamente con le chiamate di `onCreate()` e `onDestroy()`. Il periodo di attività di un servizio può iniziare in due modi differenti: con la chiamata del metodo `Context.startService()` oppure del metodo `Context.bindService()`; nel primo caso viene invocata la callback `onStart()`, nel secondo `onBind()`; la seconda ritorna un canale di collegamento con il servizio, che se rilasciato invoca la callback `onUnbind()`. Se un nuovo client si connette al servizio, quest'ultima callback invoca a propria volta `onRebind()`.

Il ciclo di vita di un servizio è rappresentato nel diagramma in Figura 11:

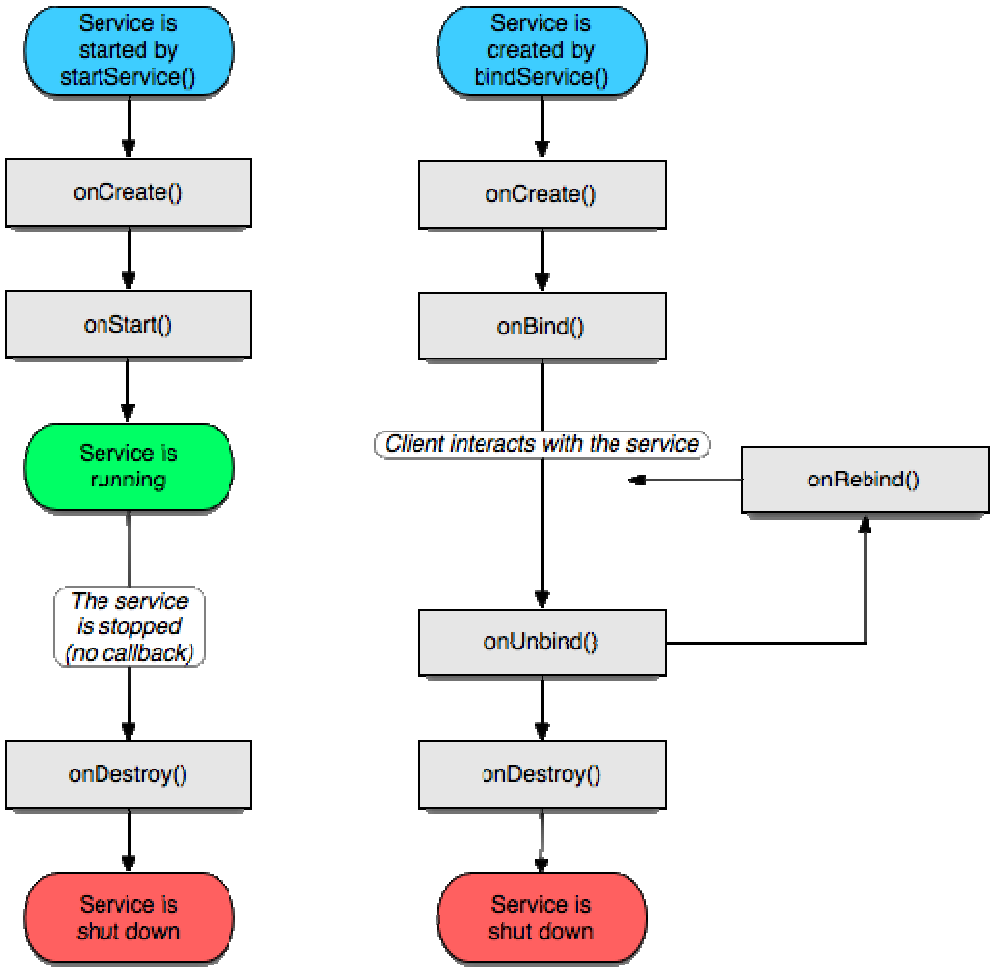

Figura 11. Ciclo di vita di un service

2.4 JADE-LEAP

Tra le implementazioni di sistemi multiagente, estesi all'ambito mobile, ci sono ad esempio IBM Aglets, Mitsubishi Concordia e JADE-LEAP Platform di Telecom Italia Lab. Nella presente sezione e nelle successive, trattiamo l'espansione di JADE all'ambiente mobile con LEAP, chiamata appunto JADE-LEAP, e conseguentemente con LEAP-Android.

La diffusione delle connessioni wireless costantemente attive (GPRS, UMTS) e lo sviluppo dei dispositivi mobili hanno portato una maggiore richiesta di integrazione degli ambienti mobili con le piattaforme fisse: ciò è quanto avvenuto per JADE e che ha portato alla nascita di JADE-LEAP [26].

L'integrazione della piattaforma ad agenti in un ambiente mobile è stata ostacolata dai seguenti punti: innanzitutto, la memoria dei dispositivi mobili, essendo molto limitata non è in grado di eseguire JADE, in alcuni casi nemmeno un container; inoltre, JADE è compatibile con la distribuzione Java Development Kit (JDK) 1.4 o superiore, mentre i

dispositivi mobili supportano eventualmente le distribuzioni Connected Device Configuration (CDC), Personal Java oppure Mobile Information Device Profile (MIDP); infine, le connessioni wireless hanno delle caratteristiche ben diverse da quelle delle connessioni cablate: rispetto a queste ultime hanno maggiore latenza, banda meno ampia, connettività incostante, etc.

Questi motivi sono le origini dell'implementazione di JADE-LEAP.

Il pacchetto LEAP, unito a JADE, costituisce l'ambiente definito come JADE-LEAP. Alcune parti del kernel di JADE sono state adattate all'occorrenza, in modo tale da costituire alcune distribuzioni:

- j2se, che serve ad eseguire JADE-LEAP su computer o macchine server;
- pjava, che può essere eseguita su dispositivi che supportino CDC (palmari);
- midp, che può essere eseguita su dispositivi con capacità molto limitate (la maggior parte dei telefoni cellulari).

Una piattaforma JADE-LEAP può comprendere, grazie alle proprie caratteristiche, diverse tipologie di dispositivi; in particolare, il container principale della piattaforma ad agenti (main container) viene eseguito su una macchina che supporti la distribuzione j2se, mentre pjava e midp possono eseguire dei container specifici, anche se in modalità differenti, *stand-alone* e *split*. Un container in modalità stand-alone è a tutti gli effetti un container comune.

Un container in modalità split è costituito di un *front end* e di un *back end*; il front end è molto più leggero di un container comune e viene eseguito sul dispositivo mobile, mentre il back end viene eseguito su una macchina fissa, detta *mediator*. Un container in modalità split suppone l'esistenza di un container j2se sul mediator, inoltre non può essere il container principale, e non supporta la mobilità e la clonazione degli agenti.

La modalità split presenta i seguenti vantaggi:

- il front end è molto leggero e può essere ospitato da qualunque dispositivo mobile;
- la comunicazione con il container principale è in effetti svolta dal back end, perciò la connessione wireless non ne risulta appesantita;
- le code dei messaggi a lato front end e back end permettono alla connessione di risultare discontinua, poiché i messaggi in entrambe le direzione da e verso l'agente vengono scaricate dalla coda soltanto quando la connessione è attiva;
- l'indirizzo IP del dispositivo è noto soltanto al back end, non ad altri container.

In conclusione, la distribuzione midp si integra perfettamente con le *midlet*, ossia le applicazioni Java per l'ambiente J2ME.

IMTP, il protocollo incaricato della comunicazione tra i container, costituisce un'altra differenza tra JADE e JADE-LEAP: esso si basa su Remote Method Invocation (RMI), e

perciò non è adatto all'ambiente mobile. LEAP IMTP è costituito di un modulo detto *command dispatcher*, che si occupa della serializzazione e dell'instradamento dei messaggi verso i container, e di diversi Internal Communication Peer (ICP), responsabili dell'invio e della ricezione dei messaggi sulla rete secondo diversi protocolli, quali ad esempio HTTP o Jade Inter Container Protocol (JICP).

Nel paragrafo successivo, vedremo come l'evoluzione sia continuata da JADE-LEAP verso JADE-Android.

2.5 JADE-LEAP for Android

La nascita e lo sviluppo della piattaforma Android hanno costretto gli autori di JADE a implementare un add-on per permettere ad un dispositivo Android di eseguire un agente sulla piattaforma. La possibilità di combinare le potenzialità della comunicazione FIPA supportata da JADE con il sistema operativo Android offre l'opportunità di sviluppare sistemi innovativi basati su modelli di social network peer-to-peer: l'add-on JADE ANDROID [27, 28] realizza questa possibilità.

Lo add-on JADE ANDROID consente ad un'applicazione Android di eseguire un agente su una piattaforma, e di richiedere all'agente l'esecuzione di comportamenti specifici (behaviour) attraverso l'uso di un metodo di scambio di oggetti generici. Tutti i vantaggi delle applicazioni JADE vengono estesi anche ai dispositivi Android. Inoltre, la piattaforma può così risultare costituita di agenti eseguiti su dispositivi differenti (non soltanto con sistema operativo Android).

La versione delle librerie di JADE su cui si basa l'add-on JADE ANDROID è JADE-LEAP CDC, utilizzata sempre in modalità *split*. L'implementazione dello add-on ha richiesto:

- la rimozione delle classi che fanno uso della librerie Swing e AWT, non supportate da Dalvik Virtual Machine; tali classi sono utilizzate per gestire l'interfaccia grafica della piattaforma ad agenti, che permette di visualizzare i container e gli agenti presenti in essa, le pagine del Directory Facilitator, ed offre alcuni tool come lo sniffer per visualizzare i messaggi scambiati tra gli agenti della piattaforma;
- l'implementazione della funzionalità di logging attraverso l'uso di android.util.Log piuttosto che di jade.util.Logger;
- implementazione delle classi descritte nei paragrafi seguenti.

`JadeGateway` fornisce l'interfaccia con il servizio `MicroRuntimeService` che effettua la connessione con la piattaforma ad agenti. L'istanza di `JadeGateway` viene creata ed inizializzata nel momento in cui avviene la connessione al servizio: in altre parole, quando il collegamento con il servizio è completato, viene passata alla activity che fa da `ConnectionListener` un'istanza di `JadeGateway` che deve essere poi salvata in una variabile locale per potere essere riutilizzata (ad esempio, per il passaggio di oggetti: non tutti i metodi sono statici); inoltre il costruttore parametrico di `JadeGateway` prevede che venga passata un'istanza di `JadeBinder` come argomento, e questo può essere fatto

soltanto dal servizio quando esso è attivo. In altre parole fornisce i seguenti metodi (alcuni dei quali sono statici):

- `executeCommand`: consegna un oggetto all'agente, utilizzato per passare i behaviour perché siano eseguiti; se l'agente non è stato ancora avviato allora vengono inizializzati container e agente; inoltre tale funzione è bloccante, e termina soltanto quando l'oggetto (comando) trasmesso viene rilasciato per mezzo della funzione `releaseCommand`; inoltre, se la piattaforma ad agenti non è attiva, dopo un lungo timeout la funzione va in timout e termina. A sua volta, questo metodo invoca il metodo `execute` di `JadeBinder`.
- `connect`: questo metodo inizializza il gateway passandogli i parametri di configurazione e si connette al servizio Android `MicroRuntimeService`;
- `shutdownJADE`: invoca il metodo omonimo di `JadeBinder`, per terminare il container locale; genera un'eccezione nel caso che non esista alcun collegamento al servizio JADE.

La classe `MicroRuntimeService` implementa un servizio Android che effettua la connessione alla piattaforma JADE attraverso la classe `jade.core.MicroRuntime` delle librerie JADE-LEAP Android.

La classe che implementa un servizio Android deve estendere `android.app.Service`, ed implementare i suoi metodi `onCreate`, `onStart` e `onDestroy`, utilizzati rispettivamente durante l'inizializzazione, l'avvio e la terminazione del servizio come descritto all'interno della sezione 3.1.3. Durante l'avvio del servizio vengono impostati tutti i parametri di connessione necessari all'interno di una mappa; tra di essi, indirizzo e porta della piattaforma, nome e classe degli agenti da eseguire ed eventuali argomenti da passare ad essi in fase di avvio.

Una classe può effettuare un *binding* con un servizio Android. Nel momento in cui la classe invoca il metodo `Context.bindService` su di un servizio, crea una dipendenza con esso: la classe riceve l'oggetto che rappresenta la connessione quando il servizio viene creato, ed il servizio rimane attivo fintanto che anche la classe cliente del servizio stesso lo è. La classe che descrive la connessione e viene ritornata all'oggetto cliente è di tipo `IBinder`, o meglio è un'implementazione di essa: nel caso specifico, si tratta di un oggetto di tipo `JadeBinderImpl`, una classe interna a `MicroRuntimeService` che estende `android.os.Binder`.

La classe `JadeBinderImpl` possiede alcuni metodi interessanti: uno di essi è `checkJADE`, che verifica se il container è in esecuzione, ed in caso negativo lo crea e poi avvia gli agenti chiamando consecutivamente i metodi statici `MicroRuntime.startJADE` e poi `MicroRuntime.startAgent` utilizzando i parametri di connessione impostati in precedenza; un altro metodo importante è `execute`, che consente di inviare oggetti all'agente attraverso l'invocazione del metodo `AgentController.putO2AObject` dopo averli incapsulati dentro oggetti di tipo `jade.util.Event`: questo sarà rilasciato soltanto esplicitamente in seguito alla chiamata di `releaseCommand` (che invoca

`Event.notifyProcessed`), poiché sull'evento è invocato il metodo `waitUntilProcessed` che blocca il *thread* chiamante; infine un metodo da citare è `shutdownJADE` che invoca in successione `MicroRuntime.killAgent` e `MicroRuntime.stopJADE` per terminare l'esecuzione dell'agente e del suo container.

Capitolo 3
Progettazione di MOEVENT

Riprendiamo quanto concluso nel Capitolo 2: lo scopo della tesi è realizzare il prototipo di un social network ad agenti per cellulari; l'implementazione del prototipo è limitata alla pianificazione di eventi per ragioni di carico di lavoro, perciò maggiore priorità viene data ai requisiti direttamente coinvolti in questa funzionalità.

3.1 Analisi dei requisiti

I requisiti funzionali del prototipo di un social network sono i seguenti:

- un ospite può diventare un membro della comunità in qualunque momento effettuando una procedura di registrazione;
- un membro del social network definisce il proprio profilo come un insieme di informazioni personali; alcune di queste informazioni identificano l'utente e non possono essere modificate, altre possono essere modificate in ogni momento; gli utenti della comunità possono accedere al profilo di un utente (o ad una parte di esso), ma non possono portarvi modifiche;
- un membro del social network può comunicare con gli altri membri del social network;
- un membro del social network può indicare alcuni membri del social network come persone che gli sono relazionate, creando in questo modo una lista di contatti personali.

I requisiti funzionali specifici della pianificazione di eventi sono i seguenti:

- il sistema permette ad un utente di pianificare un evento attraverso il cellulare, effettuando un sondaggio (*poll*) sulla data finale in cui si svolgerà l'evento;
- un utente può promuovere un numero illimitato di eventi; ciascun evento è caratterizzato da alcuni parametri fissi (titolo e descrizione), una lista di date proposte, che costituiranno le alternative del sondaggio, una lista di invitati, che è un sottoinsieme dei contatti personali del promotore;
- l'utente promotore di un evento può tenere costantemente sotto controllo l'andamento dei sondaggi sulle date proposte e decidere in qualunque momento quale tra di esse è la data in cui l'evento avrà luogo, ossia la data finale dell'evento;
- un utente invitato riceve un messaggio che specifica i dettagli sull'evento, e può esprimere in ogni momento le sue preferenze nel sondaggio sulle date, scegliendo tra di esse quelle più convenienti per sé come date finali; la scelta non è condizionata, perché l'invitato non ha nessuna informazione sugli altri invitati.

Allo scopo di semplificare ulteriormente i requisiti funzionali di un social network aggiungiamo che:

- un ospite effettua la registrazione e conseguentemente l'accesso al sistema in modo semplice, attraverso un solo clic;
- il profilo di un utente della comunità contiene solo informazioni che lo identificano e non possono essere modificate;
- un utente della comunità può comunicare con gli altri utenti soltanto in merito alla pianificazione di eventi;
- un utente ha una lista di contatti personali non limitata; un utente può aggiungere un qualsiasi altro utente del sistema alla lista dei suoi contatti senza richiedere a quest'ultimo il permesso di farlo.

In conclusione, il sistema deve prevedere che gli utenti utilizzino dispositivi mobili:

- la procedura di autenticazione di un utente avviene attraverso il suo numero di telefono.

Analizziamo ora in dettaglio le entità coinvolte:

- utente;
- evento;
- partecipazione.

Un utente è un individuo che fa uso del sistema; un utente può assumere i ruoli di promotore di un evento o di partecipante. Ogni utente ha un identificativo univoco e dei dati personali (nome, cognome e numero di telefono). Un utente è solitamente legato ad altri utenti attraverso una relazione, proprio come nei social network: tra di essi, quelli che appartengono al sistema sono i suoi contatti. Tutti gli utenti sono uguali all'interno del sistema.

Un evento è un avvenimento o un'iniziativa futura caratterizzata da proprietà fisse: titolo e descrizione. La descrizione può fornire dettagli sull'evento quali il programma, la locazione, il costo pro capite, etc.

Abbiamo identificato diverse tipologie di eventi, che possono essere racchiuse in due essenziali categorie:

- *evento singolo*: ha luogo una sola volta; la ripetizione di questo evento non è esclusa a priori, ma non è nemmeno prevista durante la fase di pianificazione; per questo tipo di evento esiste una sola data selezionata come data finale dello stesso;
- *evento composto*: si svolge in più sessioni non necessariamente periodiche; per questo tipo di evento esistono più date finali, quelle in cui le diverse sessioni si svolgeranno.

Nel mondo reale esistono scenari più complessi; tuttavia, un evento non può essere rappresentanto virtualmente se non con delle semplificazioni. Esistono svariate tipologie di evento che richiedono una gestione molto differente: ad esempio, organizzare una

partita di calcio a cinque è differente da organizzare un seminario sulla letteratura o una festa di compleanno; una partita di calcio ha un numero esatto di partecipanti, e il promotore deve trovare un campo libero nel giorno e nell'orario desiderati; un seminario è un evento pubblico cui tutti possono partecipare, esistono un minimo numero di partecipanti, almeno le spese devono essere coperte, un massimo numero di partecipanti, perché la sala designata per l'incontro ha un numero di posti limitato, e può aver luogo nel momento in cui si trova una sala disponibile e un relatore; una festa di compleanno è un evento destinato ad un gruppo ristretto di persone, non ha dei limiti precisi sul numero minimo e massimo di partecipanti. Generalizzando si può affermare che un evento:

- ha un insieme di requisiti che devono essere soddisfatti perché esso si possa svolgere; ad esempio, un numero minimo di partecipanti o la disponibilità di un locale;
- ha una visibilità pubblica, privata o ibrida: può essere aperto a tutti, soltanto a un gruppo ristretto di persone, oppure può essere privato ma permettere che tra i partecipanti figurino anche persone che non sono a diretto contatto con il promotore; ad esempio, l'invito a una partita di calcio a cinque, che necessita esattamente di dieci giocatori può essere esteso ai conoscenti dei partecipanti pur di completare la rosa di giocatori;
- un evento può avere diverse categorie di partecipanti, associate a una gerarchia; ad esempio, potrebbero esistere dei partecipanti privilegiati, che affiancano il promotore nella fase di organizzazione.

Per semplicità, abbiamo scelto di trattare eventi *senza requisiti*, la cui soddisfazione è lasciata all'organizzatore dell'evento attraverso mezzi esterni al sistema, rivolti a *gruppi ristretti di persone* e dove *non esistono invitati privilegiati*: ciò non implica che il servizio offerto sia insufficiente o incompleto.

Un evento è organizzato da un utente *promotore*, e ad esso possono prendere parte altri utenti da esso *invitati*. L'insieme di promotore e invitati è detto insieme dei *partecipanti*.

La relazione che associa utenti ed eventi è detta *partecipazione*; in particolare, una partecipazione associa la presenza (o l'assenza) di un utente ad un evento specifico, la preferenza di un utente relativamente ad una data proposta per un evento specifico.

La conclusione positiva della fase di organizzazione di un evento dipende dalla soddisfazione di alcuni requisiti, e ad essa segue la conferma a tutti i partecipanti che l'evento avrà luogo; i requisiti sono formule semplici che contengono delle variabili specifiche. Tuttavia, questa funzionalità esula dai nostri obiettivi di creazione di un semplice prototipo, perciò non è stata implementata.

I concetti appena descritti costituiscono il vocabolario (ontologico) del nostro sistema di organizzazione di eventi: MOEVENT.

I casi d’uso esprimono le funzionalità di cui il nostro sistema deve essere dotato, quindi trattano le procedure di creazione e di promozione di un evento, di gestione dell’evento creato e di reazione ad un invito con un’accettazione o un rifiuto della partecipazione. I casi d’uso formalizzano le azioni che coinvolgono le entità presenti nel sistema.

Nel linguaggio UML, i diagrammi dei casi d’uso sono composti dai seguenti elementi:

- gli attori, che rappresentano gli utenti, o più in generale, i soggetti delle funzionalità sviluppate;
- i casi d’uso, che rappresentano le funzionalità sviluppate;
- le relazioni che associano gli attori ai casi d’uso e i casi d’uso fra loro; in questo secondo caso, esistono relazioni di inclusione, estensione, generalizzazione e raggruppamento, per indicare rispettivamente che un caso d’uso: include nei propri passi un altro caso d’uso; estende con i propri passi un altro caso d’uso preesistente ed indipendente; eredita (cioè estende) un altro caso d’uso; infine che i casi d’uso hanno qualcosa in comune.

Di seguito, presentiamo la figura che illustra il diagramma dei casi d’uso di MOEVENT (Figura 12. Analisi dei casi d'uso).

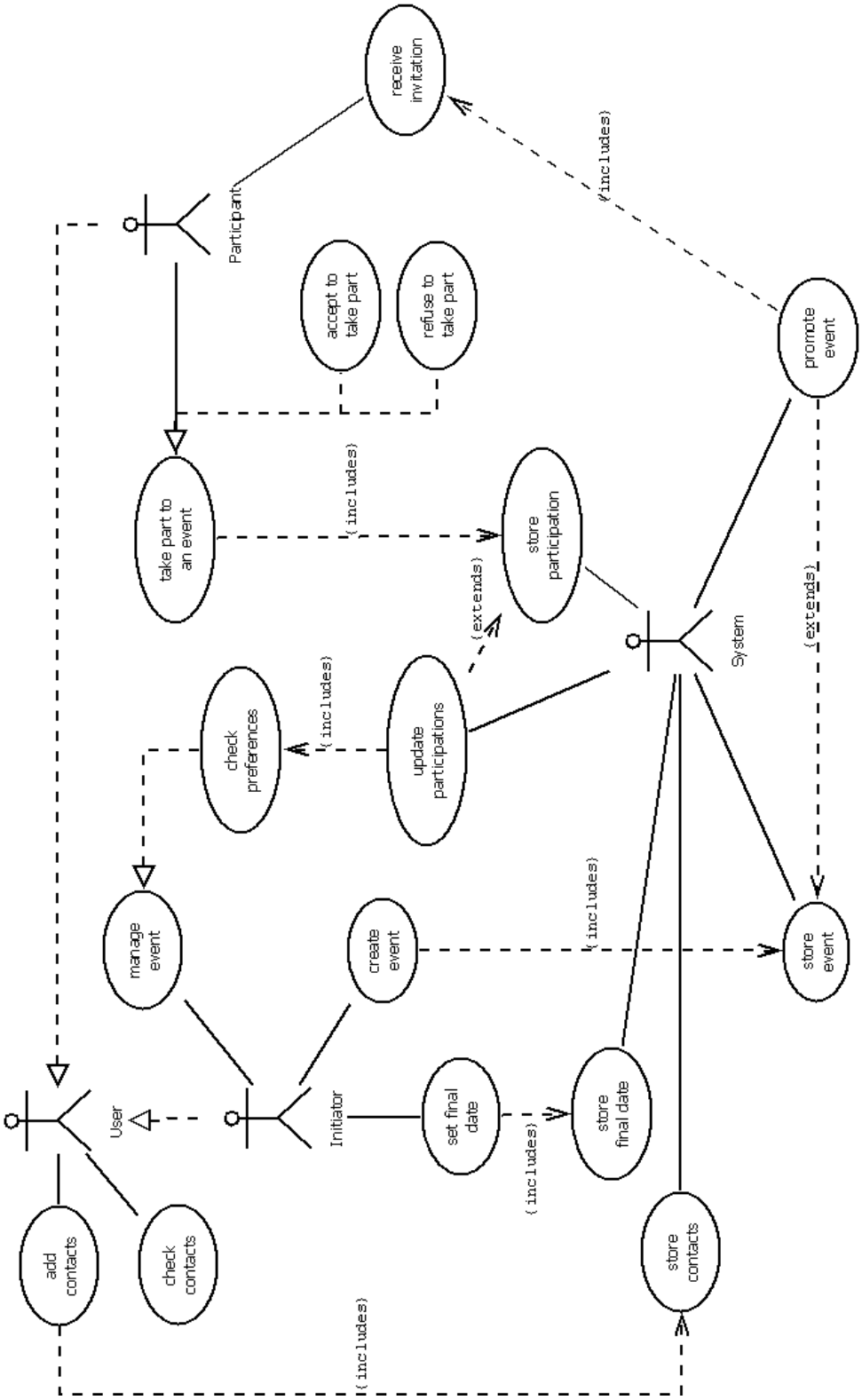

Figura 12. Analisi dei casi d'uso

- *initiator*: il promotore dell'evento, che crea e gestisce l'evento stesso;

- *participant*: l'invitato ad un evento, che può accettare o rifiutare un invito ricevuto; entrambi questi attori si identificano con l'utente reale (rappresentato nella figura dall'attore *user*);
- *system*: il sistema che gestisce la creazione degli eventi e l'accettazione o il rifiuto degli inviti, provvedendo ad informare l'initiator o i participant a seconda del caso.

Nel seguito del paragrafo, indicheremo gli attori con il proprio nome o con ciò che rappresentano: ad esempio *utente* o *promotore* indicano indifferentemente l'attore *initiator*. I casi d'uso sono:

- *create event*: il promotore può creare un nuovo evento, per cui fornisce una descrizione, un insieme di date ed una lista di partecipanti;
- *store event*: nel momento in cui il sistema riceve la richiesta di creare un nuovo evento reagisce memorizzando le informazioni appropriate, pertanto la creazione dell'evento si concretizza soltanto quando esso viene registrato dal sistema (da cui la relazione di inclusione);
- *promote event*: la promozione di un evento presso gli invitati avviene nel momento in cui system conclude la validazione di un nuovo evento e la sua memorizzazione nella banca dati;
- *receive invitation*: un partecipante riceve un invito in seguito alla promozione di un evento (promote event) da parte del sistema; naturalmente, l'operazione di promozione può dirsi conclusa soltanto nel momento in cui tutti i partecipanti all'evento hanno ricevuto un invito per lo stesso;
- *take part in an event*: in seguito alla ricezione di un invito, un partecipante può decidere di prendere parte o meno all'evento; manifestare la propria preferenza è un'operazione che si conclude soltanto nel momento in cui il sistema convalida e memorizza la preferenza stessa;
- *accept to take part* o *refuse to take part*: accettare o rifiutarsi di partecipare ad un evento possono essere considerate funzionalità generalizzate dal manifestare la propria preferenza (il caso d'uso precedente);
- *update participations*: dopo che un utente ha manifestato la propria preferenza, il sistema può rendere noto l'aggiornamento al promotore dell'evento relativo;
- *check preferences*: l'aggiornamento delle partecipazioni da parte del sistema è visualizzato nel momento in cui il promotore verifica le preferenze relative ad un evento;
- *manage event*: gestire un evento significa monitorare la lista dei partecipanti e le loro preferenze.
- *set final date*: contrassegnare una data come *final* significa confermare ai partecipanti che l'evento avrà luogo in quella data precisa.

I requisiti non funzionali del progetto, che la logica architetturale mira a soddisfare, sono i seguenti:

- *mobilità*: gli utenti del sistema fanno uso di dispositivi mobili;
- *sicurezza*: l'accesso al sistema richiede all'utente di sottoporsi ad una procedura di autenticazione basata sul proprio numero di telefono;
- *riservatezza, affidabilità e accessibilità dei dati*: le informazioni sullo stato del sistema sono conservate in una banca dati centrale, accessibile direttamente soltanto agli operatori del sistema, ma consultabile in ogni momento e in ogni luogo dagli utenti;
- *affidabilità della comunicazione*;
- *scalabilità*: il sistema è basato su una piattaforma ad agenti scalabile.

3.2 Architettura logica del sistema

Disegnamo un'architettura in grado di soddisfare i requisiti funzionali e non funzionali che ci siamo posti.

In base all'analisi dei casi d'uso svolta in precedenza, si possono identificare due entità che agiscono all'interno della piattaforma:

- l'utente, che può svolgere il ruolo di promotore o di partecipante ad un evento;
- il sistema, che riceve messaggi di promotori e partecipanti, li convalida e li analizza, e secondo la tipologia di richiesta, accede alla banca dati per memorizzare nuove informazioni, modificare quelle esistenti o per restituirle agli utenti.

Le due entità sono associabili a due tipologie di agente: l'agente utente (*user agent*) svolge il ruolo dell'attore *utente*, mentre l'agente di sistema (system agent) svolge il ruolo dell'attore *sistema*. L'associazione dei concetti di utente e agente è stata spiegata nel Capitolo 1; invece il concetto di agente di sistema nasce nel momento in cui si comprende che, per ragioni di sicurezza, l'agente utente non deve poter accedere direttamente ai dati. Inoltre, l'agente di sistema esiste per supportare estensioni future dell'architettura.

I casi d'uso relazionati agli attori sono implementati dagli agenti sotto forma di uno o più comportamenti (*behaviour*). Pertanto l'agente utente deve eseguire, su richiesta dell'utente, le azioni *create event*, *promote event*, *check preferences*, etc.; l'agente di sistema, invece, riceve una richiesta, in base alle azioni dell'agente utente, di eseguire un'azione che va a leggere o scrivere informazioni nella banca dati, e può scegliere se soddisfare o meno la richiesta. Indicativamente, un agente di sistema:

- rifiuta le richieste che riceve da parte di utenti che non hanno eseguito la procedura di autenticazione;
- rifiuta le richieste non pertinenti o non comprensibili;
- accetta tutte le altre richieste.

Lo *user agent* riceve input dall'utente stesso, che fa uso di un dispositivo cellulare, anche se talvolta può eseguire azioni per il system agent, come l'aggiornamento dei dati locali

(disponibili cioè sul dispositivo mobile); i dati sono conservati in una banca dati centrale e non distribuiti per permettere all'utente di fare accesso al sistema, pianificare e gestire eventi con qualunque dispositivo disponibile: ciò soddisfa il requisito di mobilità.

In altre parole, l'agente utente riceve l'input dall'utente attraverso l'interfaccia dell'applicativo installato sul dispositivo mobile, e comunica con un altro agente utente esclusivamente attraverso un agente di sistema; l'agente di sistema implementa lo strato di validazione e memorizzazione dei dati e lo strato di sicurezza della piattaforma, in quanto mediatore di ogni tipo di conversazione.

L'architettura logica si presenta pertanto come in Figura 13. Architettura logica di MOEVENT

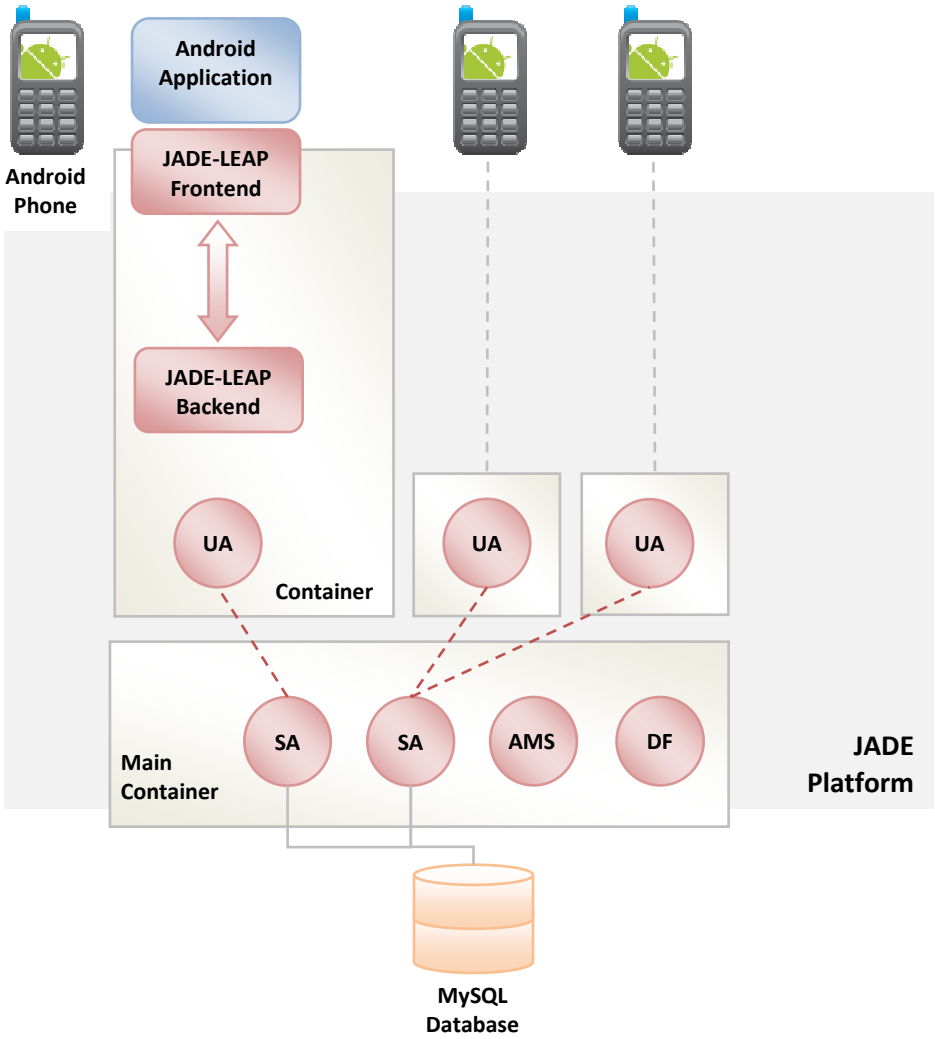

Figura 13. Architettura logica di MOEVENT

Gli user agent e i system agent fanno parte di un sistema multiagente realizzato con la piattaforma JADE: gli user agent sono indicati in figura con l'acronimo UA, mentre i system agent con SA. L'architettura del sistema segue il modello *peer-to-peer* non puro:

alcuni *nodi* (ossia alcuni agenti della piattaforma) coprono un ruolo più importante degli altri. Infatti, una piattaforma JADE ospita agenti speciali dedicati al mantenimento della piattaforma stessa (AMS) e alla registrazione e ricerca dei servizi (DF); in più abbiamo creato degli agenti di sistema per garantire sicurezza e controllo al sistema e per la gestione delle informazioni relative agli utenti e agli eventi. La scelta tra le varie piattaforme ad agenti è ricaduta su JADE perché JADE è open source, scritto in Java e supporta l'esecuzione di agenti da parte dei dispositivi cellulari (JADE-LEAP).

A lato utente troviamo i dispositivi mobili con sistema operativo Google Android. Ogni dispositivo è interfacciato con la piattaforma ad agenti JADE, attraverso il plug-in dedicato JADE-LEAP for Android. Grazie a questa interfaccia, i dispositivi possono eseguire il proprio agente personale (UA) sulla piattaforma. L'esecuzione di JADE-LEAP avviene in modalità *split*, ovvero il container che esegue l'agente personale è dislocato sia sul dispositivo mobile (che viste le capacità limitate esegue soltanto il *Frontend* leggero) sia su una macchina fissa (detta *mediator*, che ospita il *Backend*). Per maggiori informazioni sull'esecuzione di JADE-LEAP si rimanda al Capitolo 2.

Ogni agente personale si trova in un proprio container, mentre tutti gli agenti di sistema si trovano all'interno del container principale (Main Container in Figura 13), insieme agli agenti speciali AMS e DF. I container appartenenti a una stessa piattaforma possono essere eseguiti su una o più macchine.

Per supportare altri dispositivi cellulari, dovremmo creare un applicativo client dedicato a ciascuno di essi; perciò abbiamo deciso di implementare soltanto un'applicazione per Android, perché Android è open source e innovativo, è scritto in Java ed è interfacciato con JADE, e la sua diffusione sul mercato è sempre più evidente.

La banca dati che conserva le informazioni sullo stato del sistema è stata realizzata con il Database Management System MySQL.

3.3 Progettazione di dettaglio

La piattaforma ad agenti e la connessione tra gli agenti e i dispositivi mobili sono state implementate dagli autori di JADE e dello add-on JADE-LEAP for Android. Per i dettagli inerenti queste tecnologie, rimandiamo al Capitolo 2.

La negoziazione tra gli individui, per ciò che concerne la pianificazione di un evento, è stata implementata con un sistema di sondaggio sulle date, perché ciascun partecipante esprima la propria preferenza sulla data (o sulle date) in cui desidera che si svolga l'evento. Quando un utente decide di organizzare un evento può selezionare una o più date in cui l'evento avrà luogo, secondo la propria disponibilità; nel caso di eventi semplici, che si svolgono in una data precisa, l'organizzatore seleziona una data finale tra quelle proposte, mentre nel caso di eventi composti può selezionare diverse date finali per indicare quali sessioni avranno effettivamente luogo. In entrambi i casi, gli utenti invitati possono esprimere la loro preferenza a proposito di ciascuna data proposta, indicando la propria presenza, assenza o incertezza.

Per evitare di influenzare l'andamento del sondaggio abbiamo scelto di non mostrare ad alcuno degli utenti invitati gli altri invitati allo stesso evento: un utente deve rispondere unicamente in base alle informazioni che il promotore dell'evento gli fornisce, e in base alla propria disponibilità. La scelta della data finale tra quelle proposte per un evento semplice è lasciata all'utente, anche se si può facilmente implementare questa funzione con un comportamento specifico di user agent.

Al fine di rendere meglio l'idea di come sono rappresentate le entità del contesto nel progetto, riportiamo di seguito la struttura del database, che definisce lo stato del sistema, ossia un fotografia del sistema in un preciso istante che descrive: gli utenti che hanno accesso al sistema; gli eventi che sono stati pianificati; le relazioni che sussistono tra gli utenti (ovvero le associazioni tra ciascun utente e i suoi contatti personali); le relazioni tra gli utenti e gli eventi (ovvero quali utenti partecipano ad una specifica data tra quelle proposte per un evento), chiamate partecipazioni. In particolare, le tabelle rappresentano le entità del sistema, mentre i campi interni delle tabelle rappresentano le caratteristiche o le proprietà delle entità. L'elenco riporta le tabelle e i relativi campi come voci principali e secondarie:

- **users**: mantiene informazioni sugli utenti del sistema;
  - **id**: identificatore univoco dell'utente;
  - **name**: nome utente;
  - **phone**: numero di telefono dell'utente considerato univoco tanto quanto l'identificatore;
- **contacts**: mantiene informazioni sulle relazioni che intercorrono tra gli utenti del sistema;
  - **id_user**: identificatore dell'utente soggetto della relazione;
  - **id_contact**: identificatore dell'utente oggetto della relazione;
- **dates**: mantiene informazioni delle date implicate nella creazione degli eventi; è più performante avere un numero intero che identifica la data piuttosto che non riportarne il valore ove ce ne sia bisogno;
  - **id**: identificatore univoco della data;
  - **id_event**: identificatore dell'evento cui si riferisce la data (può essere una data proposta o finale);
  - **time**: valore in millisecondi rappresentante la data in questione, calcolato come il tempo passato a partire dal giorno 1 Gennaio 1970 alle ore 00:00;
- **events**: mantiene informazioni sugli eventi creati;
  - **id**: identificatore univoco dell'evento;
  - **id_user**: identificatore dell'utente promotore dell'evento;
  - **name**: titolo dell'evento;
  - **description**: descrizione testuale dell'evento;
  - **time_created**: valore in millisecondi che esprime la data in cui l'evento è stato creato;

- **id_date**: l'identificatore della data proposta per l'evento e selezionata come data finale; nel caso che nessuna data sia stata selezionata come finale, il campo è nullo;

- **participations**: mantiene informazioni riguardo le relazioni tra gli utenti e gli eventi; ogni *entry* della tabella è associata ad una tripla (evento, utente, data);
  - **id**: identificatore univoco dell'associazione;
  - **id_event**: identificatore dell'evento;
  - **id_user**: identificatore dell'utente associato all'evento;
  - **id_date**: identificatore della data proposta associata all'evento;
  - **status**: stato della partecipazione: positiva, in dubbio o negativa.

Seguono i sequence diagram che illustrano i messaggi scambiati in corso di esecuzione tra le principali componenti architetturali.

La procedura di connessione non ha bisogno di presentazioni, in quanto già descritta nella sezione JADE-LEAP for Android del Capitolo 2.

La procedura di comunicazione segue lo schema in Figura 14.

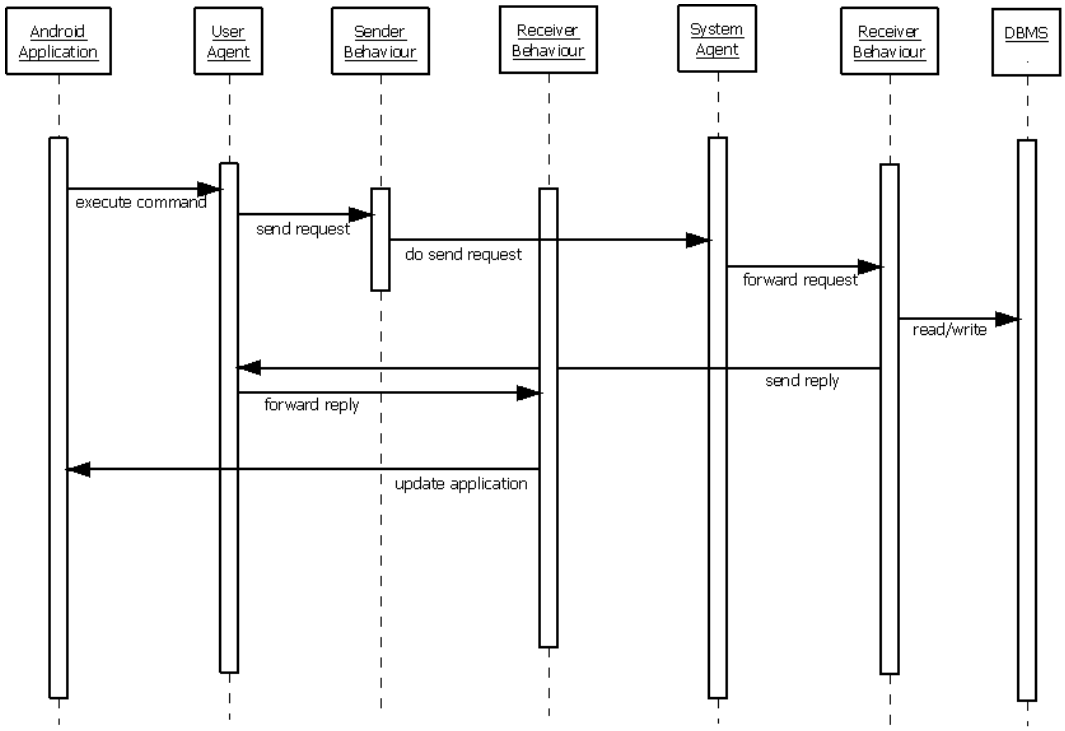

Figura 14. Sequence diagram sulla procedura di comunicazione

Ogni operazione descritta nei casi d'uso segue gli stessi meccanismi, perciò il diagramma riportato in Figura 14 copre tutti i casi possibili, a parte la procedura di connessione. Ogni azione parte da un input dell'utente, registrato dall'interfaccia dell'applicazione Android, e prosegue verso il sistema, a cui si presenta come una richiesta; in base a determinate condizioni, la richiesta può essere accettata ed interpretata dal sistema, che applica quanto richiesto attuando un processo di lettura e scrittura sulla banca dati;

di fatto, ogni azione comporta il cambiamento dello stato del sistema, e la modifica delle informazioni presenti nella banca dati rappresenta proprio questo cambiamento.

Alcune operazioni richiedono di modificare lo stato del sistema, altre invece richiedono informazioni circa lo stato attuale del sistema, altre entrambe le cose: le prime si concludono quando il sistema apporta una modifica alle informazioni conservate nella banca dati; le altre prevedono che il sistema invii le informazioni estratte dalla banca dati all'applicazione Android, la quale provvede ad aggiornarsi.

Tutte le operazioni comportano uno scambio di messaggi tra l'utente e il sistema, cioè tra user agent e system agent. Lo user agent deve eseguire due behaviour, per inviare e per ricevere messaggi; mentre il system agent esegue un behaviour per la ricezione dei messaggi, e lo stesso behaviour sporadicamente invia i messaggi di risposta: definiamo il behaviour atto a spedire messaggi "behaviour mittente", e il behaviour che rimane in attesa (bloccante, al fine di consumare meno risorse) di messaggi entranti "behaviour ricevente".

Analizziamo il sequence diagram: l'input dell'utente viene inoltrato come un comando dall'interfaccia utente dell'applicazione Android allo user agent connesso; al comando l'agente risponde inizializzando un behaviour mittente per spedire la richiesta ad un system agent scelto casualmente attraverso il servizio di messaggistica MTP. Il servizio inserisce il messaggio nella coda dei messaggi del system agent, che lo preleva con il suo behaviour ricevente e lo analizza, quindi esegue una o più operazioni di lettura e scrittura sul database. Una volta terminata questa serie di operazioni, se necessario, il behaviour ricevente chiede al system agent di inviare un messaggio di risposta allo user agent: naturalemente, il messaggio contiene i dati che l'utente, o l'applicazione Android, desidera ricevere. In base ai dati ricevuti, l'applicazione aggiorna l'interfaccia grafica.

Nella prossima sezione descriviamo il protocollo di comunicazione utilizzato, analizzando le tipologie di messaggi di richiesta e le relative tipologie di messaggi di risposta.

3.4 Protocollo di interazione

La messaggistica è basata sul modello FIPA: l'atto comunicativo dei messaggi inviati dagli agenti personali agli agenti di sistema è una richiesta (REQUEST), mentre quello dei messaggi trasmessi dagli agenti di sistema agli agenti personali è una risposta che contiene informazioni (INFORM).

Di seguito analizziamo l'intento che specifica nel dettaglio l'atto comunicativo di un messaggio e le relazioni tra i messaggi di richiesta e quelli di risposta, ove sussistono.

- *sign-in*
  - intento: richiesta di registrazione o accesso;
  - mittente: agente personale;
  - destinatario: agente di sistema;

    - risposta: l'agente di sistema invia dei messaggi contenenti informazioni sugli eventi creati (get-promotions), i contatti personali (get-contacts) e gli inviti ricevuti (get-invitations);
- *promote*
    - intento: creazione di un nuovo evento;
    - mittente: agente personale;
    - destinatario: agente di sistema;
    - risposta: get-promotions;
- *invite*
    - intento: invito a un evento creato recentemente, a seguito della ricezione di un messaggio promote;
    - mittente: agente di sistema;
    - destinatario: agente personale;
- *accept-refuse*
    - intento: accettazione o rifiuto di partecipare a un evento;
    - mittente: agente personale (partecipante) o agente di sistema;
    - destinatario: agente di sistema o agente personale (promotore);
- *get-all-users*
    - intento: ricavare o comunicare informazioni sugli utenti del sistema;
    - mittente: agente personale (richiesta) o agente di sistema (risposta);
    - destinatario: agente di sistema (richiesta) o agente personale (risposta);
    - risposta: get-all-users, solo se inviato da un agente personale;
- *add-contact*
    - intento: aggiunta di un utente del sistema come contatto personale;
    - mittente: agente personale;
    - destinatario: agente di sistema;
    - risposta: get-contacts;
- *get-contacts*
    - intento: ricavare o comunicare informazioni sui contatti personali;
    - mittente: agente personale (richiesta) o agente di sistema (risposta);
    - destinatario: agente di sistema (richiesta) o agente personale (risposta);
    - risposta: get-contacts, trasmesso da un agente di sistema a seguito di un messaggio get-contacts o add-contact;
- *get-promotions*
    - intento: ricavare o comunicare informazioni sugli eventi creati;
    - mittente: agente personale (richiesta) o agente di sistema (risposta);
    - destinatario: agente di sistema (richiesta) o agente personale (risposta);
    - risposta: get-promotions, trasmesso da un agente di sistema a seguito di un messaggio get-promotions o promote;
- *get-invitations*
    - intento: ricavare o comunicare informazioni sugli inviti ricevuti;
    - mittente: agente personale (richiesta) o agente di sistema (risposta);
    - destinatario: agente di sistema (richiesta) o agente personale (risposta);

        - risposta: get-invitations, trasmesso da un agente di sistema a seguito di un messaggio get-invitations;
- *set-final-date*
    - intento: impostare la data di un evento creato come *finale*;
    - mittente: agente personale;
    - destinatario: agente di sistema.

Abbiamo concluso la fase di progettazione del prototipo, pertanto nel prossimo capitolo analizzeremo l'implementazione della specifica funzione della pianificazione di eventi.

Capitolo 4
Implementazione di MOEVENT

L'implementazione di MOEVENT è suddivisa in tre progetti che contengono il codice dell'applicativo Android (sezione 4.2), della piattaforma (sezione 4.3) e la libreria comune ai precedenti (sezione 4.1). Nella sezione 4.4 vengono illustrati i casi d'uso in fase operativa.

4.1 Il progetto MOEVENTLib

Il progetto MOEVENTLib costituisce la libreria comune agli altri due progetti, perciò contiene tutte le classi condivise.

I diagrammi delle classi mostrano le classi cui appartengono gli oggetti scritti nel codice, eventualmente i metodi e variabili globali delle classi stesse, comprensivi della loro visibilità, ma soprattutto le relazioni che intercorrono fra le classi, come ad esempio generalizzazioni e associazioni.

Presentiamo di seguito i diagrammi delle classi più importanti che appartengono ai package del progetto MOEVENTLib e le relative descrizioni.

Il package core

Il package `core` è costituito dalle classi `Event`, `User` e `Participation`, associate agli omonimi concetti chiave del sistema, descritti nella sezione ; le tre classi identificano anche le rispettive entità nel database relazionale.

La Figura 15 mostra il diagramma delle tre classi; per ogni classe vengono mostrati gli attributi, ma non i metodi, essendo costruttori con e senza parametri, *getter* e *setter* gli unici presenti.

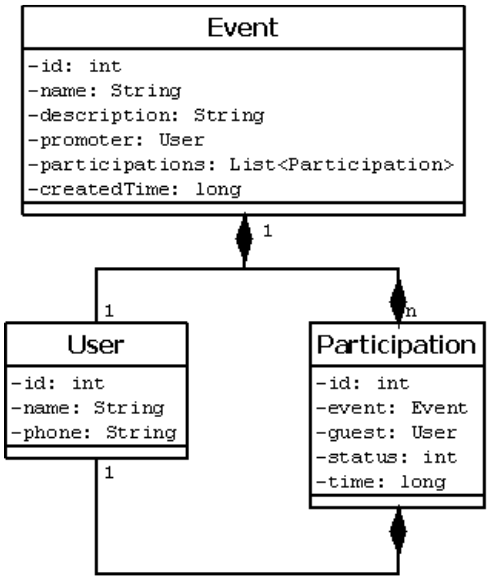

Figura 15. it.unitn.MOEVENT.core

`Event` rappresenta un evento, perciò al suo interno contiene:

- `name`, una stringa contenente il titolo dell'evento;
- `description`, una stringa che contiene la descrizione dell'evento;
- `promoter`, un'istanza della classe User che contiene informazioni sull'utente cha ha creato l'evento;
- `participations`, una lista di oggetti Participation per memorizzare la preferenza di un utente per una particolare data proposta per l'evento stesso;
- `createdTime`, che esprime in millisecondi la data di creazione dell'evento, un dato utile unicamente a livello di sistema;
- `finalTime`, che rappresenta la data decisiva (final) per l'evento, espressa in millisecondi;
- `id`, un numero intero che identifica l'evento in maniera univoca, e corrisponde al valore della chiave primaria nella tabella degli eventi del database.

Ricapitolando, un evento ha un solo promotore, uno o più invitati ad ognuno dei quali è associata una preferenza per ciascuna data proposta perché l'evento abbia luogo.

Un'istanza della classe `User` serve a rappresentare un utente, e contiene i seguenti attributi:

- `id`, un identificativo numerico univoco dell'utente; esso viene estrapolato dalla tabella degli utenti nel database, e non è noto a priori all'utente; il motivo per cui non viene utilizzato direttamente il numero di telefono come identificativo è legato al fatto che il confronto tra numeri interi è più rapido rispetto al confronto tra stringhe;
- `name`, una stringa che ha per valore il nome dell'utente;

- `phone`, una stringa contenente il numero di telefono dell'utente, che lo identifica univocamente; d'altronde, i numeri di telefono sono unici, perciò esso è stato utilizzato per garantire all'utente la sicurezza dell'accesso al sistema: l'applicativo a lato utente fornisce il numero di telefono attivo sul dispositivo stesso al sistema per la verifica della sicurezza.

La classe `Participation` contiene i seguenti attributi:

- `id`, ancora una volta contenente l'identificativo numerico della entry della tabella delle partecipazioni corrispondente all'oggetto corrente;
- `event`, un'istanza della classe `Event`, che contiene l'evento associato con la preferenza; nel caso in cui l'istanza corrente faccia parte di una lista di preferenze appartenente ad un'istanza di `Event`, l'oggetto `Event` interno non contiene informazioni se non `id`, onde evitare la replicazione dei dati;
- `guest`, un'istanza della classe User, che contiene i dati dell'utente associato con la preferenza; da un punto di vista concettuale, `guest` rappresenta sempre un utente invitato;
- `status`, un valore intero rappresentante lo stato attuale della preferenza; i possibili valori associati a status sono `STATUS_SUSPENDED` (0), `STATUS_CONFIRMED` (1), `STATUS_CANCELLED` (2), costanti contenute nella classe stessa che rappresentano rispettivamente: che l'utente non ha ancora deciso se accettare o rifiutare l'invito; che l'utente ha accettato l'invito; che l'utente ha rifiutato di partecipare all'evento;
- `time`, un valore numerico che esprime in millisecondi la data associata alla preferenza.

Il package msg

Il package `msg` contiene le classi utilizzate per lo scambio di messaggi tra gli agenti. Un messaggio costituisce un atto comunicativo, e in quanto tale deve contenere delle espressioni interpretabili in maniera univoca attraverso il linguaggio costruito …

I messaggi scambiati dagli agenti in una piattaforma JADE contengono diversi campi al loro interno, tra cui il mittente del messaggio, i riceventi ed il contenuto testuale del messaggio. Questo contenuto testuale è utilizzato per aggiungere informazioni sull'ontologia del sistema; l'ontologia è scritta in formato XML ed è rappresentata dalla classe `EMessage`.

Nel diagramma in Figura 16 sono riportate anche alcune classi (`Event`, `User`) esterne al package in esame perché comunque relazionate ad esso.

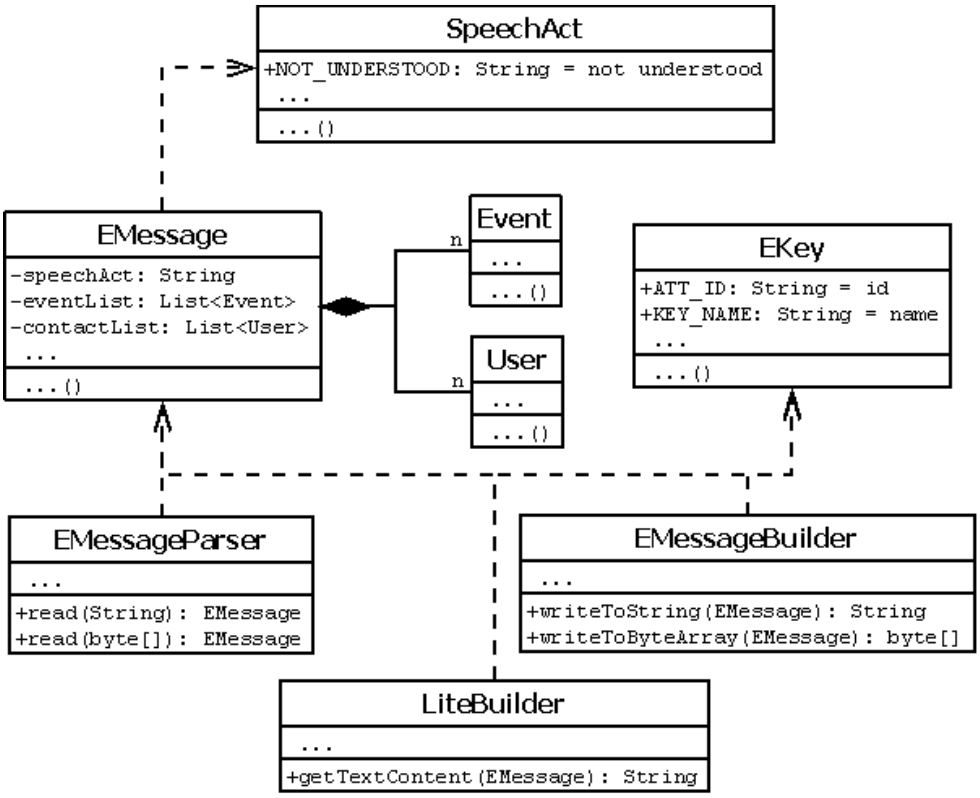

Figura 16. it.unitn.MOEVENT.msg

Tra le classi del package non esistono relazioni dirette, come associazioni, aggregamenti o generalizzazioni, ma piuttosto delle dipendenze generiche. Ad esempio, le classi utilizzate per creare il contenuto testuale dei messaggi (`EMessageBuilder`) o per leggerlo (`EMessageParser`) sono legate all'implementazione della classe che rappresenta il contenuto testuale stesso (`EMessage`).

La classe `EMessage` estende il messaggio ACL, nel senso che implementa il campo del contenuto del messaggio stesso. Perciò la classe contiene gli attributi:

- `speechAct`, una stringa che rappresenta l'atto comunicativo del messaggio;
- `eventList`, una lista di oggetti di tipo `Event`, utilizzata per le operazioni sugli eventi;
- `contactList`, una lista di oggetti di tipo `User`, utilizzata per le operazioni sui contatti.

Gli atti comunicativi sono elencati come attributi della classe `SpeechAct`, o meglio come i valori degli attributi, che sono delle stringhe univoche. Nel diagramma delle classi è stato riportato un solo attributo per necessità di spazio. Di seguito, ne riportiamo l'elenco completo con relativa descrizione del caso di utilizzo:

- `SIGN_IN`, indica che si sta effettuando un accesso al sistema;
- `PROMOTE`, indica che si sta creando un nuovo evento;
- `INVITE`, indica che si sta consegnando l'invito a un evento creato recentemente;

- ACCEPT_REFUSE, dichiara la propria preferenza in merito ad una data proposta per un particolare evento;
- GET_PROMOTIONS, indica che si desidera conoscere tutti gli eventi creati in precedenza;
- GET_INVITATIONS, qualora si desideri ottenere la lista degli inviti ricevuti;
- ADD_CONTACT, identifica la richiesta di aggiungere un nuovo contatto;
- GET_CONTACTS, identifica la richiesta di conoscere tutti i propri contatti;
- GET_ALL_USERS, identifica la richiesta di avere la lista di tutti gli utenti;
- SET_FINAL_DATE, identifica la richiesta di impostare una data proposta per l'evento come decisiva;
- NOT_UNDERSTOOD, identifica la risposta di un agente che non ha compreso l'atto comunicativo di un messaggio, perché il suo formato non corrisponde a quello atteso.

Secondo l'atto comunicativo, l'oggetto `EMessage` decide di istanziare ed aggiungere un oggetto `Event` o `User` alla lista.

Ad esempio, se l'utente (o meglio, il suo agente) necessita di comunicare la creazione di un nuovo evento, l'oggetto che rappresenta l'evento viene aggiunto alla lista di eventi dell'istanza di `EMessage` prima che questa sia tradotta in linguaggio XML e passata come contenuto testuale di un messaggio ACL ad un agente di sistema. In questo caso, l'oggetto `Event` in questione contiene la lista delle date proposte per l'evento sottoforma di una lista di date, di cui viene fatto il prodotto cartesiano con l'insieme degli utenti per ottenere la lista delle preferenze.

Di seguito elenchiamo quali attributi sono specificati per un oggetto `EMessage` secondo l'atto comunicativo del messaggio:

- PROMOTE: la lista contiene una sola istanza di `Event`, che rappresenta l'evento che si intende creare e promuovere. L'istanza contiene una lista di date proposte per l'evento e una lista di preferenze che memorizza gli invitati, senza specificare una data né uno stato per ciascuna di esse;
- SIGN_IN, GET_PROMOTIONS, GET_INVITATIONS, GET_CONTACTS, GET_ALL_USERS, NOT_UNDERSTOOD: le liste degli eventi e degli utenti è vuota nel caso in cui il messaggio venga inviato dall'agente utente all'agente di sistema per ottenere informazioni; la risposta dell'agente di sistema invece utilizza le liste per passare all'utente le informazioni richieste; precisamente:
    - SIGN_IN: l'agente di sistema inserisce un'istanza di `User` nella lista dei contatti; tale istanza contiene come dato significativo l'identificativo dell'utente (`id`);
    - GET_PROMOTIONS: il messaggio contiene la lista degli eventi creati, comprensivi delle preferenze espresse da ciascun utente per ciascuna delle date proposte per l'evento;

        - GET_INVITATIONS: il messaggio contiene la lista degli eventi cui si è stati invitati; per ogni evento è indicato anche il promotore;
        - GET_CONTACTS: il messaggio contiene la lista degli oggetti User che corrispondono ai contatti personali dell'utente, che l'agente di sistema inserisce nel messaggio dopo averli estratti dal database;
        - GET_ALL_USERS: il messaggio contiene una lista di tutti gli utenti del sistema, inseriti come istanze di User nella lista dei contatti dell'istanza di EMessage;
    - INVITE: la lista degli eventi contiene un oggetto Event rappresentante l'evento a cui l'utente viene invitato;
    - ACCEPT_REFUSE: l'istanza di EMessage contiene un evento nella lista degli eventi, quello per cui si esprime la/e preferenza/e;
    - ADD_CONTACT: l'istanza di EMessage contiene a propria volta una o più istanze di User nella lista degli utenti, ad indicare quali siano i nuovi contatti;
    - SET_FINAL_DATE: l'istanza di EMessage contiene un'istanza di Event che rappresenta l'evento per cui si desidera impostare una data proposta come decisiva.

Prima di essere inviato da un agente, il contenuto di un'istanza di EMessage viene trasformato in una stringa in formato XML e successivamente impostato come valore del campo *contenuto* del messaggio ACL che lo racchiude. Viceversa, quando il messaggio giunge a destinazione, il contenuto testuale viene analizzato sintatticamente e viene creato un oggetto di tipo EMessage corrispondente.

Le classi EMessageBuilder e LiteBuilder sono utilizzate per la costruzione del contenuto testuale del messaggio ACL, formattato in XML, a partire da un'istanza della classe EMessage, che contiene già tutte le informazioni necessarie.

La traduzione del contenuto di un'istanza di EMessage in XML è svolto dal metodo writeToString(EMessage) della classe EMessageBuilder. Questo metodo prevede l'impiego di librerie che appartengono al package javax.xml.stream, escluso dalle librerie standard di Android: per questo motivo è stata creata la classe LiteBuilder, che con il metodo getTextContent(EMessage) svolge la stessa funzionalità concatenando semplicemente i contenuti testuali in uno StringBuilder.

La classe EMessageParser svolge l'analisi sintattica del testo contenuto nel campo *content* del messaggio ACL ricevuto dall'agente destinatario, e successivamente crea un oggetto EMessage scrivendo i valori corretti nei rispettivi campi. La funzionalità è svolta dal metodo read che a sua volta richiama il metodo ricorsivo visit, che effettua una visita in profondità dell'albero XML relativo al campo di testo, ottenuto sfruttando le potenzialità di DOM.

I tag e gli attributi utilizzati nel linguaggio XML sono conservati nella classe EKey.

Di seguito presentiamo un esempio di un'istanza di `EMessage` e del testo corrispondente. Supposto che l'utente promotore decida di festeggiare il prossimo compleanno, che si svolge il giorno 7 Novembre 2009, proponendo come date possibili quelle comprese nel weekend del 7 Novembre, ossia il giorno stesso (sabato) e il giorno seguente (domenica). L'utente invita al proprio compleanno gli utenti User 1, User 2 e User 3.

Seguono una tabella rappresentante l'oggetto `EMessage` nella memoria e il testo corrispondente in formato XML.

...	
speechAct	promote
eventList(0).id	0
eventList(0).name	Birthday party
eventList(0).description	Hi guys, you're all invited to my birthday party in my house! Since it is on 7th November, I chose two dates for you: vote your preference!
eventList(0).timeCreated	1250507978000
eventList(0).timeList(0)	1257606000000
eventList(0).timeList(1)	1257692400000
eventList(0).promoter.id	1
eventList(0).promoter.name	Promoter 0
eventList(0).promoter.phone	
eventList(0).participations(0).id	0
eventList(0).participations(0).user.id	3
eventList(0).participations(0).user.name	User 1
eventList(0).participations(0).user.phone	00391111111111
eventList(0).participations(0).status	0
eventList(0).participations(1).id	0
eventList(0).participations(1).user.id	4
eventList(0).participations(1).user.name	User 2
eventList(0).participations(1).user.phone	00392222222222
eventList(0).participations(1).status	0
eventList(0).participations(2).user.id	5
eventList(0).participations(2).user.name	User 3
eventList(0).participations(2).user.phone	00393333333333
eventList(0).participations(2).status	0
contactList	
...	

Figura 17. Un oggetto EMessage in memoria

```
<?xml version="1.0" encoding="UTF-8"?>
<emsg xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
       <metadata version="1.0"></metadata>
       <comm-act type="promote" />
       <event id="0">
              <name>Birthday party</name>
              <description>Hi guys,
                     you're all invited to my birthday party in my house!
Since it is on 7th November, I chose two dates for you: vote your
preference!</description>
              <dateCreated>2009-08-17T15:19:38</dateCreated>
```

```
            <proposedTime>2009-11-07T15:18:00</proposedTime>
            <proposedTime>2009-11-08T15:18:00</proposedTime>
            <participation id="0" status="0">
                  <proposedTime>1970-01-01T01:00:00</proposedTime>
                  <guest id="2">
                        <name>User 1</name>
                        <phone>00391111111111</phone>
                  </guest>
            </participation>
            <participation id="0" status="0">
                  <proposedTime>1970-01-01T01:00:00</proposedTime>
                  <guest id="4">
                        <name>User 2</name>
                        <phone>00392222222222</phone>
                  </guest>
            </participation>
            <participation id="0" status="0">
                  <proposedTime>1970-01-01T01:00:00</proposedTime>
                  <guest id="5">
                        <name>User 3</name>
                        <phone>00393333333333</phone>
                  </guest>
            </participation>
      </event>
</emsg>
```

Figura 18. Un oggetto EMessage in XML

4.2 Il progetto MOEVENTAndroid

Il progetto MOEVENTAndroid contiene l'applicativo utente, designato per essere eseguito su piattaforma Android; all'interno del progetto si trovano anche le classi che implementano le funzionalità dello user agent.

Il progetto è costituito di diversi package, dedicati allo sviluppo dell'interfaccia utente, e dell'agente utente e dei suoi comportamenti: rispettivamente il package `core` e il package `jade`.

I package android.core, android.core.adapter

I package `android.core`, `android.core.adapter` contengono le classi che implementano l'interfaccia utente e sono chiamate *activity*.

Le activity estendono la classe `Activity` di Android, oppure una estensione della classe, come ad esempio `TabActivity`. Le activity non hanno dipendenze strette come aggregazioni, associazioni o generalizzazioni, perciò per ottenere una maggiore leggibilità dei diagrammi delle classi, riportiamo ciascuna di esse in un diagramma separato.

MainActivity

La activity principale è implementata dalla classe `MainActivity`, di cui riportiamo il diagramma a classi in Figura 19.

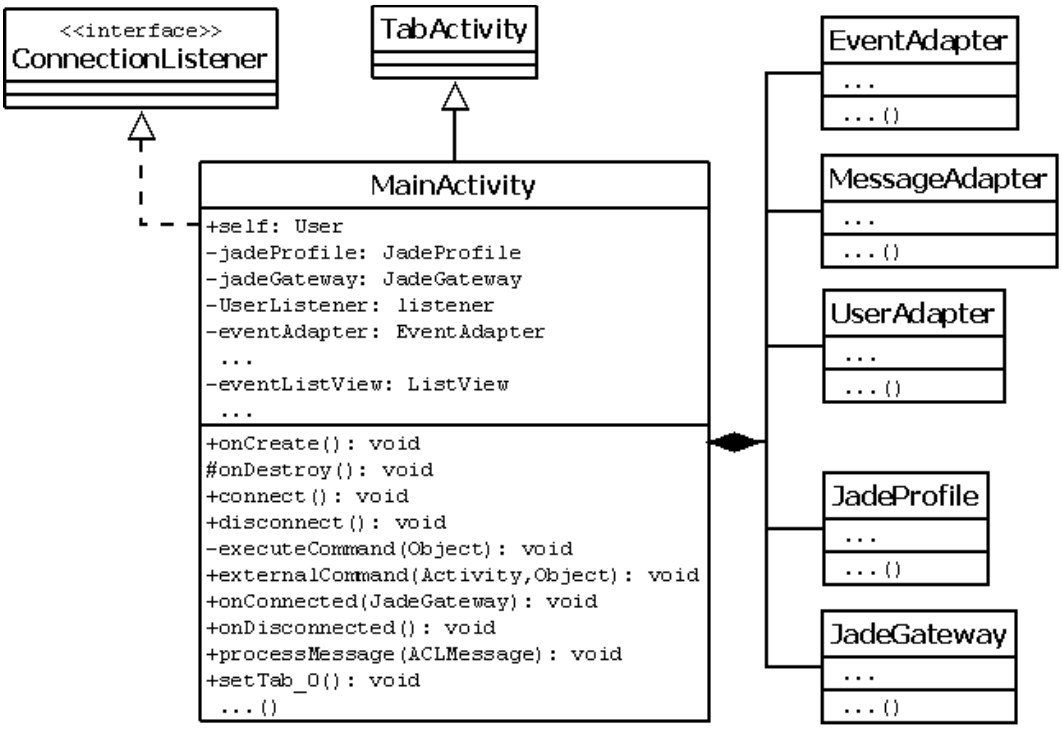

Figura 19. it.unitn.MOEVENT.android.core

L'istanza attiva di `MainActivity` svolge alcuni compiti importanti per l'applicazione:

- conserva le tabelle aggiornate degli eventi creati, dei messaggi di invito ricevuti e dei contatti personali;
- effettua e mantiene la connessione alla piattaforma ad agenti, permettendo anche ad altre attività di inviare comandi all'agente attraverso di essa;
- analizza e processa i messaggi entranti.

I metodi `onCreate` e `onDestroy` vengono ereditati dalla classe `Activity` e sono chiamati rispettivamente ogni volta che la activity inizia il proprio ciclo di vita e che lo termina (come descritto nel paragrafo Ciclo di vita dei componenti nella sezione Tecnologia del Capitolo 3).

Il metodo `onCreate(Bundle)` associa all'applicazione un profilo JADE, ossia un'istanza della classe `JadeProfile`, che racchiude le informazioni necessarie per effettuare la connessione alla piattaforma, come l'indirizzo IP della macchina che ospita la piattaforma JADE, la porta del relativo servizio e il nome dell'agente che si desidera eseguire, ed altre informazioni generiche; successivamente, il metodo tenta di aprire una connessione verso la piattaforma, utilizzando la chiamata di `JadeGateway.connect(String, String[], Properties, Context, ConnectionListener)`, a cui vengono passati come parametri nell'ordine il nome della classe dell'agente da eseguire, gli argomenti da passare all'agente, le proprietà per effettuare la connessione (contenute nell'oggetto `JadeProfile`), un riferimento alla activity corrente e al gestore delle callback di connessione. Il gestore delle callback è la classe stessa, in quanto implementa l'interfaccia `ConnectionListener`.

Quando la connessione con la piattaforma è stabilita, viene eseguita la callback `onConnected(JadeGateway)`. Il metodo inizializza l'attributo della activity `jadeGateway` con il proprio unico parametro, e successivamente invia un oggetto di tipo `UserListener` all'agente, il quale lo utilizzerà per inoltrare alla activity i messaggi ricevuti. L'istanza di `UserListener` e gli oggetti inviati all'agente successivamente come comandi vengono passati come argomento alla funzione `execute(Object)` di `JadeGateway`. L'istanza di `UserListener` dispone l'agente utente a gestire i messaggi ACL ricevuti e ad inoltrarli alla activity principale. Gli oggetti inviati all'agente utente durante il corso di esecuzione dell'applicazione sono *behaviour* che si intende fare eseguire all'agente; nella maggior parte dei casi, si tratta di behaviour atti a trasmettere una richiesta di informazioni ad un agente di sistema. Inizialmente, l'agente utente richiede informazioni riguardo gli eventi proposti, gli inviti ricevuti ed i contatti personali, e successivamente la activity principale visualizza in tre schede di liste dedicate i dati ottenuti chiamando i metodi `setTab_0()`, `setTab_1()` e `setTab_2()`.

Una volta che la activity principale viene rimossa dallo stack - esattamente come quanto avviene per ogni activity - viene chiamata la funzione `onDestroy()`, che chiude la connessione con la piattaforma ad agenti, invocando i metodi `shutdownJADE()` e `disconnect(Context)` di `JadeGateway`.

Il metodo `processMessage(ACLMessage)` pone sotto analisi ogni messaggio ACL in ingresso e intraprende azioni diverse a seconda del suo contenuto:

- `SIGN_IN`: viene impostato l'identificativo numerico univoco dell'utente corrente (le cui informazioni sono contenute nell'attributo `self` di tipo `User` dell'istanza della classe `MainActivity`);
- `GET_PROMOTIONS`, `GET_INVITATIONS`, `GET_CONTACTS`: la lista degli eventi creati, degli eventi a cui l'utente corrente è stato invitato e dei suoi contatti personali, contenute nel messaggio vengono visualizzate nell'ordine esposto nella lista della scheda di `MainActivity`;
- `GET_ALL_USERS`: la lista dei contatti disponibili (per essere aggiunti come contatti personali) viene passata attraverso l'uso di variabili statiche alla activity `AddContactActivity`;
- `INVITE`: il nuovo evento a cui si è invitati viene aggiunto alla lista degli eventi della seconda scheda di `MainActivity`;
- `ACCEPT_REFUSE`: la lista di preferenze per l'evento indicato nel messaggio aggiorna la lista di preferenze attuali dell'utente, e viene passata alla activity `MyEventDetailActivity` nel caso in cui l'evento considerato sia visualizzato proprio in quel medesimo istante.

Il metodo `externalCommand(Activity, Object)` di `MainActivity` è *statico* ed utilizzato da altre activity per inoltrare dei comandi alla piattaforma attraverso il `JadeGateway` della activity principale; ad esso vengono passati come parametro la

activity corrente che esegue l'operazione ed il comando che deve essere eseguito dall'agente utente (generalmente un behaviour).

Flusso delle attività

Le applicazioni Android sono costituite di un certo numero di finestre, dette activity, che estendono la classe `Activity` e rappresentano di fatto le attività che l'utente sta svolgendo. In Figura 20 presentiamo un diagramma di flusso che mostra le attività in esecuzione durante il ciclo di vita dell'applicazione.

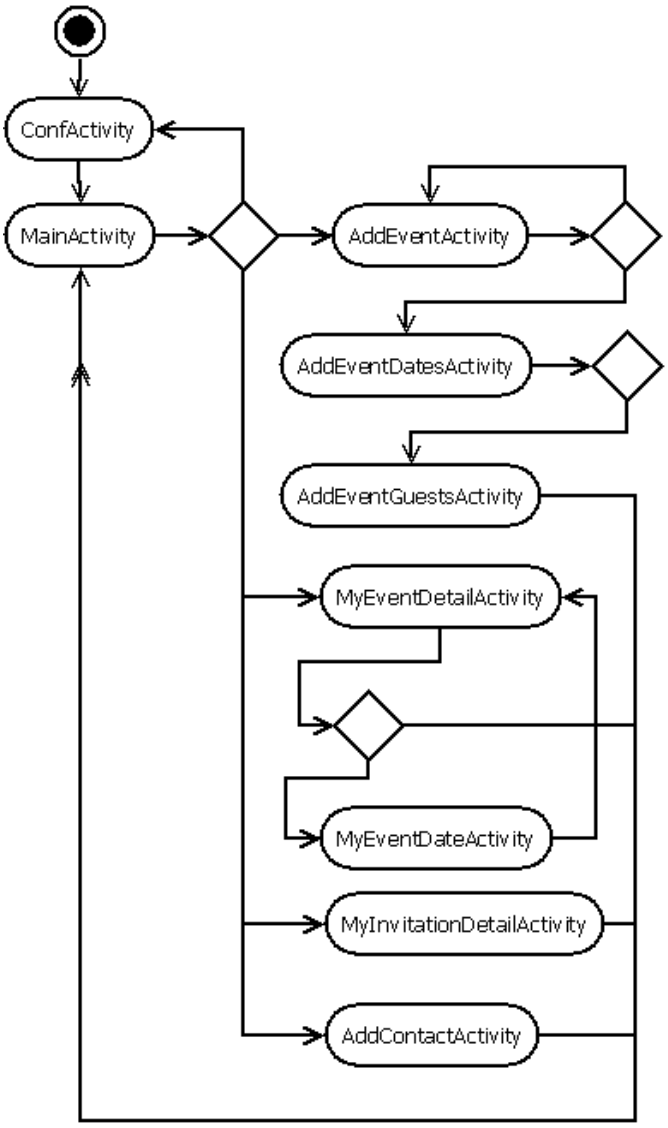

Figura 20. Activity flowchart

Lo stato iniziale e finale del diagramma rappresentano la activity principale di Android, chiamata *Home*. Da Home può essere avviato l'applicativo MOEVENT.

La prima finestra che si presenta all'utente, implementata dalla classe `ConfActivity`, contiene i parametri di configurazione; tali parametri, in seguito al primo utilizzo, vengono memorizzati nel database del dispositivo, in modo che non sia necessario

inserirne i valori ad ogni esecuzione successiva. Durante il passaggio alla activity successiva viene inizializzata la connessione con la piattaforma ad agenti.

La activity principale dell'applicazione è implementata da `MainActivity`. Questa classe è responsabile della comunicazione con la piattaforma ad agenti, in entrambe le direzioni. Soltanto nel momento in cui la connessione è stata effettuata con successo vengono recuperati i dati relativi all'utente corrente; ciò avviene attraverso la sottoposizione delle richieste dei dati all'agente utente e trasmesse da questo ad un agente di sistema disponibile nella piattaforma, ed attraverso l'interpretazione del messaggio di risposta che contiene le informazioni desiderate. Inoltre, la activity principale visualizza in tre schede diverse: la lista degli eventi creati; la lista degli inviti ricevuti; la lista dei contatti personali.

La procedura di creazione di un evento avviene in tre passi attraverso le classi: `AddEventActivity`, `AddEventDatesActivity` e `AddEventGuestsActivity`. La prima permette di inserire il titolo dell'evento ed eventualmente una sua descrizione; la seconda di aggiungere almeno una data all'evento - possono essere aggiunte più date, per l'argomento vedere il paragrafo dedicato; la terza di inserire almeno un contatto tra gli invitati all'evento e di procedere. Ad ogni passo è possibile arretrare al precedente oppure avanzare fino a concludere la procedura, che riporta alla activity principale.

La activity `MyEventDetailActivity` permette di visualizzare le preferenze espresse dagli invitati per ciascuna data proposta per un evento creato dall'utente corrente. Si accede a questa activity selezionando nella activity principale un evento dalla lista degli eventi promossi. Da `MyEventDetailActivity` si può tornare alla activity principale premendo il tasto `BACK` del telefono, oppure visualizzare ulteriori dettagli per ciascuna data proposta, semplicemente selezionandola: in questo modo viene visualizzata la activity `MyEventDateActivity`, che espone per una specifica data: gli invitati che hanno confermato la propria partecipazione ed un tasto per impostare la data in questione come *decisiva* (o *final*). Una data contrassegnata come *decisiva* viene esposta con un'icona (triangolo giallo) a lato.

La activity `MyInvitationDetailActivity` permette di visualizzare le informazioni riguardanti un evento cui si è stati invitati, e le preferenze espresse dall'utente corrente per ciascuna data proposta per l'evento stesso. Si accede a questa activity selezionando uno degli inviti tra quelli nella lista degli inviti ricevuti nella activity principale.

Infine, la activity `AddContactActivity` viene visualizzata a partire dalla activity principale, e permette di aggiungere nuovi contatti alla lista dei contatti personali.

Le activity possiedono una variabile globale pubblica `RESCODE`, che contiene un identificatore della activity. Qualora una activity avvii l'esecuzione di un'altra activity attraverso la chiamata del metodo `startForActivityResult`, riceve il valore di ritorno di questa attraverso la chiamata della callback. La callback che riceve i valori di ritorno

della activity terminata riconosce di quale activity si tratta proprio grazie al codice `RESCODE`.

Gli adapter

Un *adapter* è una classe Android utilizzata da una activity per popolare una *viewgroup* con i dati estratti da una sorgente esterna (come ad esempio il database SQLite). Nel progetto, gli adapter sono tutti utilizzati per visualizzare come elementi di una `ListView` oggetti come istanze della classe `Event`, `User`, etc.

Segue il diagramma delle classi di un adapter specifico, `EventAdapter`. La struttura degli altri adapter è del tutto analoga, perciò rimandiamo alla descrizione del diagramma che segue in Figura 21.

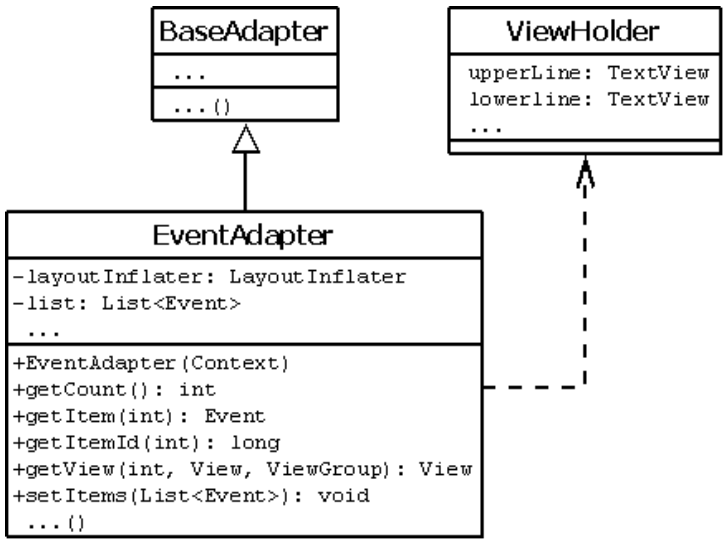

Figura 21. Esempio da it.unitn.MOEVENT.android.core.adapter

Gli adapter implementati ereditano dalla classe `BaseAdapter` delle librerie di Android.

Gli adapter possiedono una lista degli oggetti che devono visualizzare: nel caso di `EventAdapter` si tratta di oggetti di tipo `Event`.

L'attributo di tipo `LayoutInflater` è comune a tutti gli adapter, e serve ad impostare la view di ogni singolo elemento della lista a partire da un file XML. Il costruttore di `LayoutInflater` richiede come parametro un oggetto di tipo `Context`, motivo per cui il costruttore di ciascun adapter possiede a propria volta un parametro dello stesso tipo. La visualizzazione degli elementi - le righe della lista - attraverso l'uso di file XML consente una maggiore personalizzazione dell'esposizione.

I metodi `getCount`, `getItem` e `getItemId` sono comuni agli adapter, poiché sono ereditati dalla classe `BaseAdapter`. Il primo metodo restituisce un valore intero pari al numero degli elementi della lista, il secondo restituisce l'oggetto della lista alla posizione indicata dal parametro - nel caso specifico, si tratta di un oggetto di tipo `Event` - infine il

terzo restituisce un identificativo dell'oggetto, in questo caso l'indice stesso dell'oggetto nella lista.

Il metodo `getView` viene invocato da Android ogni volta che avviene un aggiornamento della view. Il secondo parametro `convertView` viene ritornato al sistema di rendering per essere visualizzato, perciò nel caso sia vuoto deve essere creato nuovamente; tuttavia, la continua creazione di oggetti di questo tipo può risultare poco performante, perciò gli oggetti visualizzati vengono salvati dentro un'istanza della classe interna statica `ViewHolder` per poter essere riutilizzati.

La classe statica `ViewHolder` contiene gli stessi oggetti grafici descritti nel relativo file di layout XML. Nel caso specifico, poiché ogni elemento della lista di eventi visualizza su una riga il titolo dell'evento e sull'altra la data di creazione dello stesso, la `ViewHolder` contiene due `TextView` rappresentanti le due linee di testo, chiamate per semplicità `upperLine` e `lowerLine`.

Ogni adapter contiene uno o più metodi per aggiungere elementi alla lista locale. Nel caso di `EventAdapter`, è stata creata una funzione `setItems(List<Event>)` che consente di impostare una lista sorgente di elementi come lista locale dello adapter.

Gli attributi e le operazioni descritte sono comuni agli adapter, tuttavia ogni implementazione contiene attributi e metodi specifici. Ad esempio `UserCheckAdapter`, utilizzato da `AddContactActivity` per visualizzare gli utenti del sistema che possono essere aggiunti come contatti personali, contiene per ogni elemento una *checkbox* ed un *listener* associato ad essa, ed un *array* di *booleani* utilizzato per associare ad ogni elemento della lista un valore *true* o *false* a seconda che la checkbox sia spuntata o meno.

Il package android.jade

android.jade contiene le classi utilizzate per l'implementazione dell'agente utente e dei relativi behaviour.

In Figura 22 è illustrato il diagramma delle classi del package `android.jade`. Il diagramma non riporta i metodi e gli attributi delle classi appartenenti a librerie di terze parti (Java, Android o JADE), mentre riporta in modo abbreviato con tre puntini attributi e operazioni di classi che appartengono ad altri package di MOEVENT.

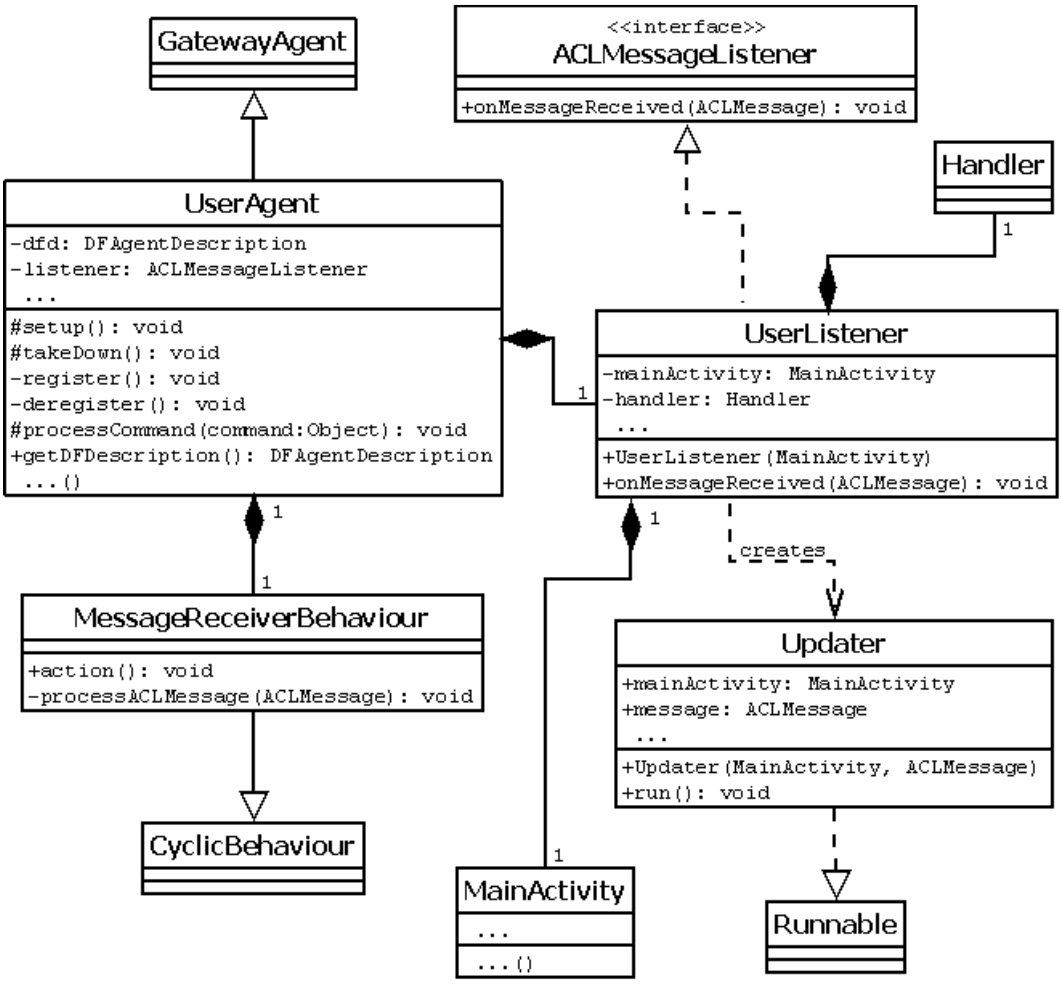

Figura 22. it.unitn.MOEVENT.android.jade

La classe `UserAgent` implementa le funzionalità di un agente utente. L'ambiente richiede che un agente eseguito in modalità split implementi la classi `GatewayAgent`: questa richiede l'implementazione del metodo `processCommand`, quanto al resto è del tutto simile ad un comune agente. Il ciclo di vita di un agente inizia con la chiamata del metodo `setup` e si conclude con quella del metodo `takeDown`; durante l'inizializzazione l'agente effettua la registrazione al Directory Facilitator, attraverso l'uso del metodo interno `register`, mentre prima di venire cancellato invoca `deregister` per eseguire la deregistrazione ed evitare che siano presenti riferimenti obsoleti ad esso all'interno delle pagine del Directory Facilitator. Il metodo `register` esegue la chiamata al metodo statico `DFService.register`, passando come argomenti un riferimento all'agente stesso e la descrizione dei suoi servizi; il metodo `deregister` chiama a propria volta la funzione `DFService.deregister`, passando come argomento il riferimento all'agente. La classe che descrive i servizi offerti dall'agente è `DFAgentDescription`, di cui è mantenuta un'istanza come variabile globale della classe.

Il metodo `processCommand`, invocato qualora una activity abbia chiamato il metodo `JadeGateway.executeCommand`, è utilizzato per passare all'agente degli oggetti generici:

qualora l'oggetto fosse vuoto o di tipo inaspettato, esso verrà ignorato. Le tipologie di oggetto accettate appartengono alle seguenti classi (o alle loro implementazioni):

- `ACLMessageListener`;
- `Behaviour`.

Inizialmente l'agente riceve un oggetto di tipo `ACLMessageListener` in modo che sia predisposto ad inoltrare i messaggi ricevuti alla activity principale; in seguito, l'agente riceve degli oggetti di tipo `Behaviour` che devono essere eseguiti. Gli oggetti di questo secondo tipo vengono banalmente aggiunti ai behaviour attivi dell'agente tramite il metodo `Agent.addBehaviour` ereditato dalla classe `jade.core.Agent`; in seguito l'agente esegue a turno il metodo `action` di tutti i behaviour attivi. L'oggetto passato come parametro al metodo `processCommand` deve essere rilasciato con l'utilizzo del metodo `releaseCommand`; nel caso che l'oggetto passato sia un behaviour, esso deve appunto essere aggiunto alla lista dei behaviour attivi prima di essere rilasciato, ma poiché `Agent.addBehaviour` non esegue l'azione del behaviour nello stesso momento in cui lo aggiunge è necessario creare un behaviour che esegue in sequenza le due operazioni richieste.

Durante l'inizializzazione, l'agente aggiunge ai behaviour attivi anche un'istanza della classe interna `MessageReceiverBehaviour`, che estende `CyclicBehaviour` poiché esegue ciclicamente l'operazione di richiedere al buffer dei messaggi ACL di cui è destinatario. L'operazione di ascolto dei messaggi è bloccante.

La classe `UserListener` implementa l'interfaccia `ACLMessageListener`, perciò una istanza della classe può essere passata come argomento alla funzione `processCommand` di `UserAgent`. La activity principale `MainActivity` invoca il metodo `executeCommand` passando come argomento uno `UserListener` che contiene un riferimento ad essa: tale riferimento è passato come argomento del costruttore della classe. L'istanza di `UserListener` così passata a UserAgent viene salvata nella sua variabile globale `listener`. La callback `onMessageReceived` viene ereditata dall'interfaccia `ACLMessageListener`; poiché il behaviour `MessageReceiverBehaviour` può accedere all'attributo `listener` dell'agente, ogni qual volta rileva un messaggio in ingresso, ne invoca la callback `onMessageReceived`.

Il metodo `onMessageReceived` crea un oggetto eseguibile di tipo `Updater` e lo esegue su un *thread* parallelo alla activity principale. L'oggetto di tipo `Updater` è eseguibile in quanto implementa l'interfaccia `Runnable`. L'esecuzione su un thread parallelo avviene assegnando un oggetto di tipo `Runnable` da eseguire ad un *handler*, che Android implementa appunto con la classe `Handler`. Perciò viene eseguito il codice scritto nel metodo `run` di `Updater`.

La classe `Updater` ha un costruttore parametrico a cui vengono passati come argomenti un'istanza della activity principale `MainActivity` ed un'istanza di `ACLMessage`, che rappresenta il nuovo messaggio entrante: in questo modo `Updater` può accedere al

metodo `processMessage(ACLMessage)` di `MainActivity`, ed il messaggio può venire analizzato e processato.

Il meccanismo descritto in questo paragrafo completa la comunicazione fornendo alle activity la possibilità di ricevere messaggi ACL in qualunque momento (la comunicazione ACL è infatti asincrona).

I behaviour dell'agente utente

L'agente utente esegue i behaviour che gli vengono inviati dalle activity attraverso i metodi `executeCommand` e `externalCommand` di `MainActivity`. Fatta eccezione per il behaviour dedicato all'ascolto dei messaggi entranti, `MessageReceiverBehaviour`, gli altri behaviour dell'agente eseguono una singola azione di invio di un messaggio particolare.

Le classi che implementano i behaviour estendono `SenderBehaviour`, che a propria volta estende `OneShotBehaviour`. La classe `SenderBehaviour` fornisce alle sotto-classi alcuni metodi di utilità: `findAgents(String)`, `findRandomAgent(String)` e `send(AID, EMessage)`; i primi due metodi sono utilizzati per effettuare una ricerca presso il Directory Facilitator di agenti che forniscono il servizio il cui nome è indicato nel parametro, e restituiscono (se la ricerca ha successo) oggetti di tipo `DFAgentDescription` che identificano gli agenti trovati; l'ultimo metodo riceve come parametri un identificatore di un agente destinatario, ed un oggetto `EMessage` da incapsulare come testo all'interno del campo contenuto del messaggio ACL che viene infine inviato. Naturalmente, `LiteBuilder` si occupa della trasformazione di `EMessage` in testo in formato XML.

La classe JadeProfile

La classe `JadeProfile` è utilizzata dall'applicativo Android per memorizzare i dati di accesso alla piattaforma JADE. Essa estende la classe `Properties`, perciò una sua istanza può essere passata come argomento al metodo `connect` di `JadeGateway`.

Oltre a contenere una mappa in memoria delle impostazioni di connessione alla piattaforma JADE, `JadeProfile` conserva tali dati nel database locale SQLite del dispositivo Android. Per questo motivo `JadeProfile` contiene al proprio interno una classe statica JadeProfileAdapter utilizzata per la comunicazione con il database, per creare nuove entry nella tabella specifica o per estrarle tutte e caricarle in memoria.

4.3 Il progetto MOEVENTServer

Il progetto MOEVENTServer contiene l'applicativo a lato server, ossia l'implementazione del system agent, dei suoi comportamenti e delle classi necessarie a gestire la banca dati.

Esso è costituito di due package, `server.mas` e `server.sql`, che contengono rispettivamente le classi che implementano l'agente di sistema e i suoi behaviour, e lo strato di comunicazione con il database MySQL.

L'agente di sistema

In Figura 23 mostriamo il diagramma delle classi del package `server.mas`.

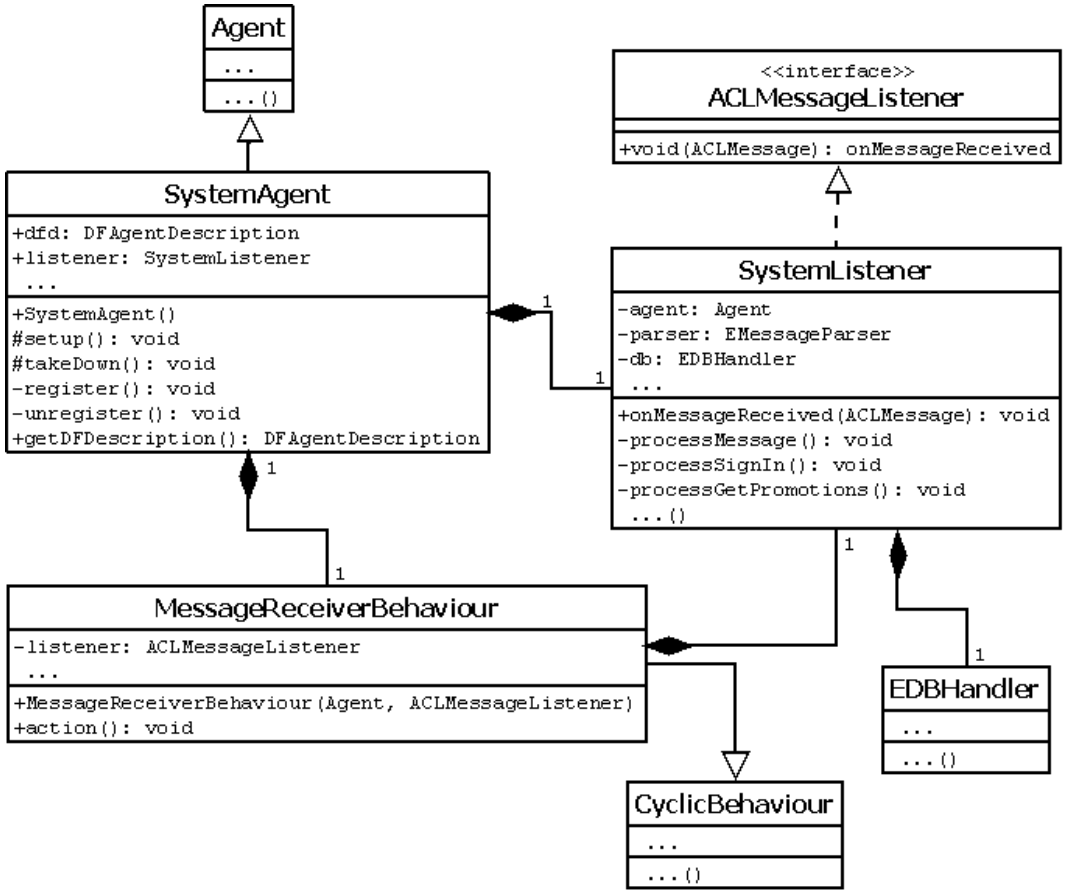

Figura 23. it.unitn.MOEVENT.server

Il package contiene le classi:

- `SystemAgent` che implementa l'agente di sistema;
- `MessageReceiverBehaviour` che implementa l'unico behaviour dell'agente di sistema; il behaviour si pone ciclicamente in ricezione di nuovi messaggi entranti;
- `SystemListener` che processa i messaggi ricevuti.

La classe `SystemAgent` estende la classe `jade.core.Agent`, ossia la classe rappresentante un agente della piattaforma JADE. I suoi metodi più significativi sono:

- il costruttore `SystemAgent()` che inizializza l'attributo di tipo `SystemListener` presente nella classe;
- il metodo `setup()` ereditato da `jade.core.Agent` che inizializza l'agente; nel corso dell'inizializzazione l'agente compie la registrazione presso il Directory Facilitator attraverso la chiamata del metodo della classe `register`; viene aggiunto ed eseguito un behaviour di tipo `MessageReceiverBehaviour`; la

registrazione dell'agente avviene attraverso la chiamata del metodo statico `DFService.register`;

- il metodo `takeDown()` ereditato anch'esso da `jade.core.Agent` ed invocato quando l'agente termina la propria esecuzione; nel corso della procedura di terminazione viene effettuata la deregistrazione dell'agente presso il Directory Facilitator attraverso l'invocazione del metodo della classe `unregister`; la deregistrazione avviene attraverso la chiamata del metodo statico `DFService.deregister`;
- il getter `getDFDescription` che ritorna l'istanza di `jade.domain.FIPAAgentManagement.DFAgentDescription` che contiene informazioni riguardo i servizi offerti dall'agente.

Durante l'inizializzazione, `SystemAgent` attiva il behaviour `MessageReceiverBehaviour`. Il behaviour estende `CyclicBehaviour`, e si pone ciclicamente in ascolto di nuovi messaggi. L'operazione di ascolto è bloccante. Il costruttore della classe, `MessageReceiverBehaviour(Agent, ACLMessageListener)` accetta come parametri un riferimento all'agente che esegue il behaviour e ad un oggetto che implementa l'interfaccia `ACLMessageListener`, ossia l'attributo `listener` di `SystemAgent`. Quando un nuovo messaggio ACL viene ricevuto, esso viene passato come argomento al metodo `onMessageReceived` che appartiene alla classe `ACLMessageListener`, o meglio alla sua implementazione `SystemListener`, di cui il behaviour ha riferimento nell'attributo `listener` passatogli da `SystemAgent`.

La classe `SystemListener` processa i messaggi entranti destinati all'agente di sistema. La classe possiede come attributi un riferimento all'agente di sistema, un'istanza di `EMessageParser` per estrarre il contenuto di un messaggio ACL in forma di un oggetto di tipo `EMessage`, ed un'istanza di `EDBHandler` per comunicare con il database MySQL.

Possiede inoltre i seguenti metodi:

- `onMessageReceived(ACLMessage)`: analizza il messaggio ACL ed ne estrae il contenuto di testo nella forma di un oggetto di tipo `EMessage`; l'estrazione dell'oggetto `EMessage` avviene attraverso la chiamata del metodo `read` di `EMessageParser`; se il contenuto del messaggio ACL è vuoto, ignora la richiesta, altrimenti la inoltra al metodo successivo;
- `processMessage()`: analizza la richiesta ed inoltra l'oggetto `EMessage` relativo ad essa ad uno dei metodi seguenti, secondo l'atto comunicativo incluso nell'oggetto `EMessage` stesso; quindi, estrae dal database le informazioni relative all'utente che ha eseguito la richiesta, estrapolando il nome dell'agente mittente del messaggio, ossia il numero di telefono del dispositivo che lo identifica in maniera univoca nel sistema: se il numero di telefono non è associato ad alcun utente nel sistema e l'atto comunicativo del messaggio è SIGN_IN, l'utente viene registrato utilizzando il metodo `processRegisterUser()`, altrimenti la richiesta viene ignorata;

- `processSignIn()`: le informazioni sull'utente estratte dal database vengono inoltrate all'utente stesso;
- `processGetPromotions()`: gli eventi di cui l'utente figura come promotore vengono estrapolati dal databse ed inseriti nella lista di eventi dell'istanza di `EMessage` che viene associata al messaggio ACL di risposta;
- `processGetInvitations()`: analogamente al precedente, questo metodo è utilizzato per estrarre dal database le informazioni inerenti gli eventi a cui l'utente è stato invitato;
- `processGetContacts()`: estrae dal database i contatti personali di un utente e le informazioni relative ad essi per poi restituirne all'utente stesso la lista;
- `processGetAllUsers()`: estrae dal database le informazioni relative a tutti gli utenti del sistema in modo che l'utente possa selezionare nuovi utenti da aggiungere ai propri contatti personali;
- `processAddContact()`: estrapola dalla richiesta il contatto che l'utente desidera inserire tra i propri contatti personali, e ritorna successivamente all'utente la lista aggiornata degli stessi;
- `processPromote()`: estrae dal messaggio in ingresso l'evento (o gli eventi) che l'utente desidera promuovere e dopo averne inserito le relative informazioni spedisce i messaggi di invito che devono essere recapitati agli ospiti;
- `processAccept()`: aggiorna lo stato della partecipazione dell'utente corrente ad un evento specifico per le date proposte, spedisce un aggiornamento delle preferenze all'utente promotore (se collegato alla piattaforma), ed infine invoca il metodo `processGetInvitations` per l'utente corrente;
- `processRegisterUser()`: estrae da `EMessage` l'unico oggetto di contactList che contiene i dati relativi all'utente che si vuole registrare; non è previsto alcun controllo, poiché conoscere il numero di telefono di un utente fornisce già una certa garanzia di sicurezza; al termine della procedura viene effettuato l'accesso al sistema da parte dell'utente corrente, ora registrato, e gli viene restituito l'id assegnato con l'utilizzo del metodo `processSignIn`;
- `processSetFinalDate()`: imposta una delle date proposte per un evento come data decisiva; tale azione può essere eseguita più volte per uno stesso evento, il che comporta che l'ultima data selezionata come decisiva sostituisce la precedente;
- `processNotUnderstood()`: viene usato come atto comunicativo quando l'atto comunicativo del messaggio non è compreso.

L'interfaccia con il database

Il package `it.unitn.MOEVENT.server.sql` fornisce dei metodi di comunicazione con il database MySQL. In Figura 24 mostriamo il diagramma delle classi relativo al package, contenente le classi più importanti.

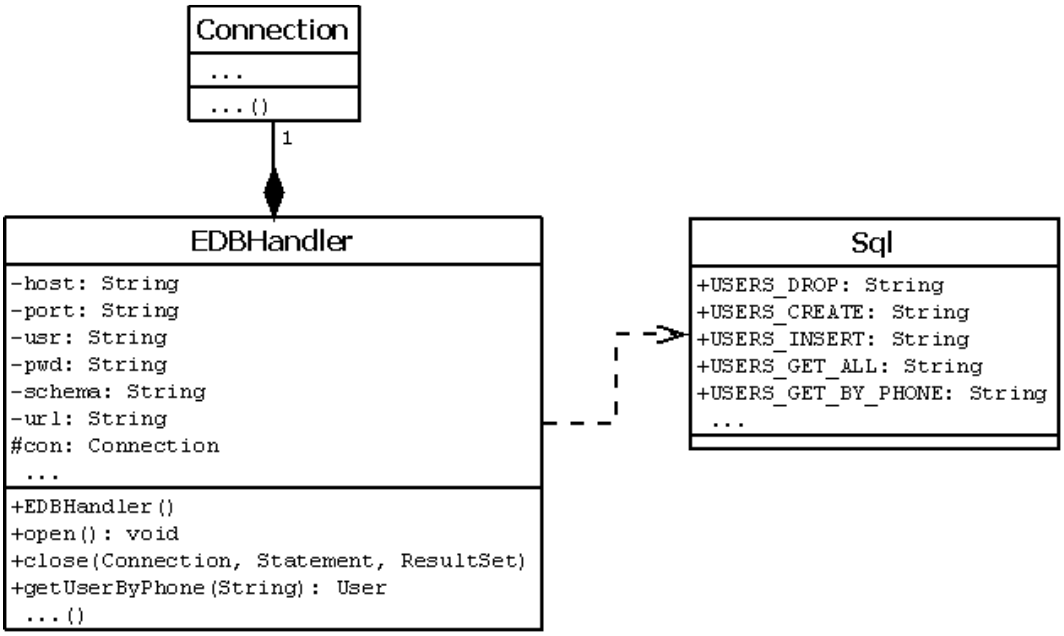

Figura 24. it.unitn.MOEVENT.server.sql

In breve, la classe `EDBHandler` fornisce delle API per effettuare una connessione al database, per terminare la connessione, per eliminare e creare le tabelle utilizzate dal sistema, per estrarre e scrivere informazioni. Le query in linguaggio SQL sono memorizzate negli attributi statici della classe `Sql`.

In particolare, in `EDBHandler` sono memorizzati sotto forma di attributi i parametri necessari ad effettuare la connessione al database: `host`, `port`, `usr`, `pwd` e `schema`, che sono stringhe rappresentanti rispettivamente l'indirizzo o il nome della macchina che ospita il database, la porta del servizio, il nome utente e la password per accedere al database e alle informazioni in esso contenute e il nome del database utilizzato per il sistema MOEVENT. La creazione della stringa di connessione (`url`) viene creata dal costruttore `EDBHandler`.

Ogni metodo per scrivere o leggere a database apre una nuova connessione utilizzando la funzione `open`, mentre gli oggetti utilizzati vengono rilasciati quando sono passati come parametri (tutti o parte di essi) alla funzione `close`. Tutte le API iniziano con la chiamata di `open` e terminano con (oppure ritornano subito dopo) la chiamata di `close`.

A titolo di esempio, la funzione `getUserByPhone`, come suggerisce il nome, estrae tutte le informazioni relative ad un utente partendo dal suo numero di telefono; il numero di telefono, univoco, è una stringa passata come argomento ad uno `PreparedStatement` che contiene la query `Sql.USERS_GET_BY_PHONE`. L'esecuzione dello statement ritorna un oggetto di tipo `java.sql.ResultSet` che contiene le entry selezionate dalla query; nel caso specifico, essendo il numero di telefono univoco, la query restituirà una entry singola. In ogni caso, l'oggetto di tipo `ResultSet` viene esaminato e le informazioni estrapolate dalla entry vengono inserite in un oggetto di tipo `User`. In conclusione, prima che l'oggetto sia ritornato e incapsulato in un oggetto `EMessage` e quindi inserito in

forma testuale nel contenuto di un messaggio, gli oggetti utilizzati nel corso dell'operazione di tipo `Connection`, `PreparedStatement` e `ResultSet`, vengono rilasciati con l'invocazione del metodo `close`.

Gli altri metodi svolgono operazioni analoghe a quella descritta: essi sono richiamati dall'agente di sistema attraverso `SystemListener` in seguito alla ricezione di un messaggio etichettato da un particolare atto comunicativo. Ne riportiamo un elenco completo:

- `int insertUser(User)`: utilizzato nel corso della registrazione di un nuovo utente, restituisce l'identificativo numerico univoco associato all'utente; la query associata è `Sql.USERS_INSERT`;
- `List<Event> getEventsByUser(User)`: tale metodo viene invocato in seguito alla ricezione di un messaggio di tipo `GET_PROMOTIONS`, ed associato alla query `Sql.EVENTS_GET_BY_USER`;
- `List<Event> getParticipationsByUser(User)`: invocato in seguito alla ricezione di un messaggio di tipo `GET_INVITATIONS`, ed associato alla query `Sql.PARTICIPATIONS_GET_BY_USER`;
- `List<User> getContactsByUser(User)`: invocato in seguito alla ricezione di un messaggio di tipo `GET_CONTACTS`, ed associato alla query `Sql.CONTACTS_GET_BY_USER`;
- `List<User> getAllUsers()`: invocato in seguito alla ricezione di un messaggio di tipo `GET_ALL_USERS`, ed associato alla query `Sql.USERS_GET_ALL`;
- `void insertContact(User, User)`: invocato in seguito alla ricezione di un messaggio di tipo `ADD_CONTACT`, ed associato alla query `Sql.CONTACTS_INSERT`;
- `Event insertEvent(Event)`: invocato in seguito alla ricezione di un messaggio di tipo `PROMOTE`, ed associato alla query `Sql.EVENTS_INSERT`; esegue il prodotto cartesiano delle date proposte per gli utenti invitati per ottenere gli oggetti di tipo `Participation` associati all'evento che vanno inseriti nel database; inoltre, le date proposte vengono scritte nella tabella dedicata a database;
- `int insertParticipation(Participation)`: invocato in seguito alla ricezione di un messaggio di tipo `PROMOTE`, ed associato alla query `Sql.PARTICIPATIONS_INSERT`; viene invocata come descritto nel metodo precedente;
- `void updateParticipation(Participation)`: invocato in seguito alla ricezione di un messaggio di tipo `ACCEPT`, ed associato alla query `Sql.PARTICIPATIONS_SET_STATUS`;
- `void setFinalDate(Event, DateItem)`: invocato in seguito alla ricezione di un messaggio di tipo `SET_FINAL_DATE`; la query `Sql.EVENTS_SET_FINAL_DATE` è associata a questo metodo.

4.4 Implementazione dei casi d'uso

Mostriamo di seguito l'implementazione dei casi d'uso, raccogliendo quanto detto nei diversi paragrafi, e attraversando l'architettura del sistema per ciascuno di essi.

Le immagini di riferimento utilizzate in questa sezione sono tratte dall'emulatore.

Effettuare l'accesso alla piattaforma

Al fine di effettuare l'accesso alla piattaforma, è necessario inserire i seguenti dati:

- l'indirizzo della piattaforma, ossia l'indirizzo IP della macchina server che contiene l'installazione JADE cui si desidera collegarsi;
- la porta della piattaforma, ossia la porta che identifica il servizio JADE sulla macchina server;
- il nome utente: la prima volta che si effettua l'accesso, esso viene memorizzato dal sistema;
- il numero di telefono: esso viene estrapolato automaticamente grazie alle API fornite dalle librerie di Android, e non può essere modificato; inoltre, il numero di telefono identifica un utente in modo univoco.

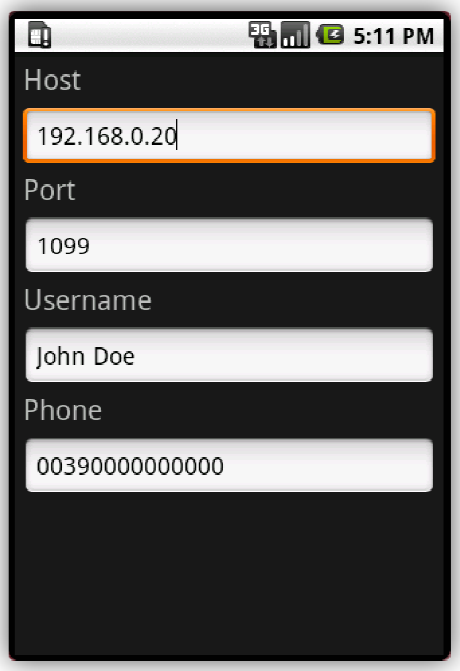

Figura 25. Schermata di connessione.

I parametri di connessione vengono inseriti manualmente dall'utente al primo utilizzo dell'applicativo, in seguito non è necessario reinserirli. I parametri di connessione vengono memorizzati nel database SQLite a disposizione del sistema operativo Android.

La figura a lato rappresenta la schermata iniziale dell'applicativo in un utilizzo successivo al primo: per avviare la procedura di connessione, è necessario premere il tasto `MENU` sul dispositivo e cliccare sulla voce di menu `CONNECT`.

La procedura di connessione crea sulla piattaforma un back end relativo all'agente in esecuzione sul dispositivo mobile, ed in seguito l'utente riceve la lista degli eventi creati, dei messaggi di invito ricevuti, ed infine dei propri contatti; questi dati vengono visualizzati rispettivamente in tre diverse schede che appaiono nella finestra principale con il titolo `EVENTS`, `INVITATIONS`, `CONTACTS`.

Si suppone che la piattaforma JADE sia operativa, e che almeno un agente di sistema sia attivo quando ha inizio la procedura di accesso da parte di un utente. L'applicativo client mostra una activity di tipo `ConfActivity`, che si presenta come nella precedente figura. Nel momento in cui si preme il pulsante per proseguire ed effettuare l'accesso, i dati inseriti vengono memorizzati nel profilo, e viene inviato un *intent* al sistema perché sia inizializzata un'istanza di `MainActivity`.

In particolare, i parametri di connessione alla piattaforma vengono salvati come coppie (chiave, valore) all'interno della mappa contenuta nella classe `JadeProfile`. Chiamando il metodo `JadeProfile.save` i valori contenuti nella mappa vengono scritti nel database locale del dispositivo Android.

Prima che sia inizializzata la classe `MainActivity`, un oggetto statico di tipo `User` viene passato ad essa; esso contiene le informazioni note riguardo all'utente corrente, ossia il nome utente ed il numero di telefono.

Un oggetto di tipo `MainActivity` viene creato, posto in cima allo stack delle activity ed inizializzato. Viene richiamato il metodo `onCreate`, che effettua una connessione con la piattaforma JADE richiamando a propria volta il metodo interno `connect` e quindi `JadeGateway.connect`. Il profilo JADE che viene passato come parametro di connessione al metodo `JadeGateway.connect` è un oggetto di tipo `JadeProfile` i cui valori sono estratti dal database SQLite del dispositivo.

Una volta che la connessione con il backend viene stabilita, il servizio di connessione invoca la callback `MainActivity.onConnected` poiché la classe implementa l'interfaccia `ConnectionListener`. L'inizializzazione dell'agente utente è terminata: è possibile effettuare la procedura di registrazione se il numero di telefono non è stato utilizzato in precedenza per connettersi alla piattaforma oppure effettuare la procedura di accesso, e successivamente inviare dei messaggi per conoscere gli eventi promossi dall'utente, gli inviti ricevuti ed i contatti personali dello stesso.

L'agente utente è implementato dalla classe `UserAgent`. In particolare, viene invocato il metodo `setup` di `UserAgent`, in modo che l'agente si prepari a ricevere messaggi attraverso il behaviour `MessageReceiverBehaviour`, e ciò avviene soltanto dopo che l'oggetto `ACLMessageListener` gli è stato passato dalla activity `MainActivity` con l'utilizzo del metodo `executeCommand` che invoca `JadeGateway.processCommand`. Come detto in precedenza, l'agente utente invia delle richieste all'agente di sistema: ciò avviene facendo eseguire all'agente nell'ordine i behaviour rappresentati dalle classi `SignInBehaviour`, `GetPromotionsBehaviour`, `GetInvitationsBehaviour` e `GetMyContactsBehaviour`. La activity `MainActivity` sceglie il momento adatto per eseguire i behaviour passando all'agente utente gli oggetti dei tipi citati come argomenti del metodo interno `executeCommand`. I behaviour ereditano tutti da `SenderBehaviour` e compiono la singola azione di inviare un messaggio ad un agente di sistema che ha reso disponibile il proprio servizio registrandosi al Directory Facilitator della piattaforma.

Inizialmente `SignInBehaviour` spedisce all'agente di sistema un messaggio di tipo `SIGN_IN`, che contiene un solo utente nella lista dei contatti, rappresentante l'utente stesso che opera la richiesta. L'istanza di `User` contiene il nome utente e il numero di telefono associati con l'utente stesso. Il SystemListener associato con l'agente di sistema può interpretare la richiesta dell'utente in due diversi modi: come una registrazione, nel caso in cui il numero di telefono non sia associato ad alcun utente presente nel database; oppure come una richiesta di accesso, nel caso in cui il numero di telefono è

presente nel database associato a qualche utente. Perciò l'oggetto `SystemListener` può richiamare `processRegister` o `processSignIn`, secondo il caso; il gestore del database (un oggetto di tipo `EDBHandler`) agisce di conseguenza: in entrambi i casi esso ricava l'identificativo numerico intero univoco associato all'utente e lo restituisce ad esso, inviando un messaggio ACL di tipo `SIGN_IN`.

Ad esempio, il messaggio di richiesta di accesso per l'utente John Doe è il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<emsg xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
	<comm-act type="sign-in" />
	<contact id="0">
		<name>John Doe</name>
		<phone>00391231234567</phone>
	</contact>
</emsg>
```

mentre il messaggio di risposta è il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<emsg xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
	<comm-act type="sign-in" />
	<contact id="2">
		<name>John Doe</name>
		<phone>00391231234567</phone>
	</contact>
</emsg>
```

L'agente utente invia al sistema una richiesta che serve a ottenere le informazioni riguardanti gli eventi creati dall'utente; il behaviour `GetPromotionsBehaviour` invia un messaggio ACL che ha per contenuto un oggetto `EMessage` con atto comunicativo `GET_PROMOTIONS`. L'agente di sistema analizza il messaggio ricevuto e, se l'utente che fa la richiesta è registrato al sistema, il `SystemListener` associato all'agente invoca il metodo interno `processGetPromotions`; tale metodo seleziona dal database tutte le entry della tabella degli eventi, chiamata `events`, per cui l'identificativo numerico del promotore eguaglia l'identificativo numerico dell'utente, in altre parole seleziona gli eventi di cui l'utente è promotore. Naturalmente, per ottenere le informazioni necessarie bisogna eseguire delle query incrociate (`JOIN`) sulle tabelle degli eventi (`events`), degli utenti (`users`), delle preferenze (`participations`) e delle date (`dates`), poiché ad ogni evento sono associati una lista di utenti invitati, ciascuno dei quali esprime la propria preferenza (ossia accetta o rifiuta di parteciparvi) per ciascuna data proposta dal promotore. Dalle informazioni ricavate dal database, viene creata una lista di oggetti di tipo `Event` che viene associata con un oggetto `EMessage` inserito nel campo contenuto del messaggio ACL (un oggetto che appartiene alla classe `ACLMessage`) di risposta.

Segue un esempio del contenuto testuale dei messaggi di richiesta e di risposta per ricevere informazioni sugli eventi promossi. Si suppone che l'utente promotore sia John Doe, gli eventi creati Event 1 e Event 2, gli utenti invitati Contact 1 e Contact 2.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<emsg xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <comm-act type="get-promotions" />
</emsg>

<?xml version="1.0" encoding="UTF-8"?>
<emsg xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <comm-act type="get-promotions" />
      <event id="12">
            <name>Event 1</name>
            <description>Description of Event 1</description>
            <dateCreated>2009-08-17T15:19:38</dateCreated>
            <participation id="34" status="0">
                  <proposedTime>2009-10-05T20:00:00</proposedTime>
                  <guest id="4">
                        <name>Contact 1</name>
                        <phone>00391111111111</phone>
                  </guest>
            </participation>
            <participation id="35" status="1">
                  <proposedTime>2009-10-05T20:00:00</proposedTime>
                  <guest id="6">
                        <name>Contact 2</name>
                        <phone>00392222222222</phone>
                  </guest>
            </participation>
      </event>
      <event id="15">
            <name>Event 2</name>
            <description>Description of Event 2</description>
            <dateCreated>2009-08-17T15:25:32</dateCreated>
            <participation id="41" status="1">
                  <proposedTime>2009-11-07T18:00:00</proposedTime>
                  <guest id="4">
                        <name>Contact 1</name>
                        <phone>00391111111111</phone>
                  </guest>
            </participation>
            <participation id="42" status="0">
                  <proposedTime>2009-11-08T18:00:00</proposedTime>
                  <guest id="4">
                        <name>Contact 1</name>
                        <phone>00391111111111</phone>
                  </guest>
            </participation>
      </event>
</emsg>
```

Il messaggio precedente spiega che gli eventi promossi dall'utente corrente sono Event 1 ed Event 2: al primo sono stati invitati Contact 1 e Contact 2, con una sola data proposta, il 5 Ottobre, per cui essi hanno previsto rispettivamente di non partecipare e di partecipare; al secondo è stato invitato l'utente Contact 1 che parteciperà soltanto l'8 Novembre, non il 7.

I messaggi scambiati e le operazioni eseguite dagli agenti e dall'applicativo client per ricavare la lista degli eventi per cui si è ricevuto un invito e per la lista dei contatti personali sono del tutto analoghi al caso illustrato.

A mano a mano che l'applicativo client ottiene le informazioni necessarie sugli eventi creati, sugli inviti ricevuti e sui contatti personali, la activity `MainActivity` può riempire le liste degli eventi, degli inviti e dei contatti nelle rispettive schede.

Aggiungere contatti

Per promuovere nuovi eventi è necessario invitare uno o più tra i propri contatti; perciò la prima cosa da fare è aggiungere uno o più utenti del sistema come contatti personali.

Per svolgere questa operazione è sufficiente aprire la terza scheda della finestra principale, e premere il tasto `MENU` sul dispositivo e selezionare la voce di menu `ADD`; la finestra aperta si presenta come in Figura 26 (a). La lista dei possibili contatti comprende tutti gli utenti della piattaforma che non siano già presenti tra i contatti dell'utente in questione, e per ciascuno di essi viene mostrato il nome utente ed il numero di telefono che li identifica. Per aggiungere uno o più contatti è sufficiente selezionarli uno ad uno, spuntando la relativa casella con un clic, quindi premere il tasto `MENU` sul dispositivo e successivamente la voce del menu `GO`: al termine dell'operazione, l'applicazione ritorna alla schermata principale, con la scheda `CONTACTS` aperta che mostra la lista dei contatti comprensiva dei nuovi contatti aggiunti.

Supponendo di avere inserito `Contact 1` quale nuovo contatto dell'utente `John Doe`, la schermata appare come in Figura 26 (b).

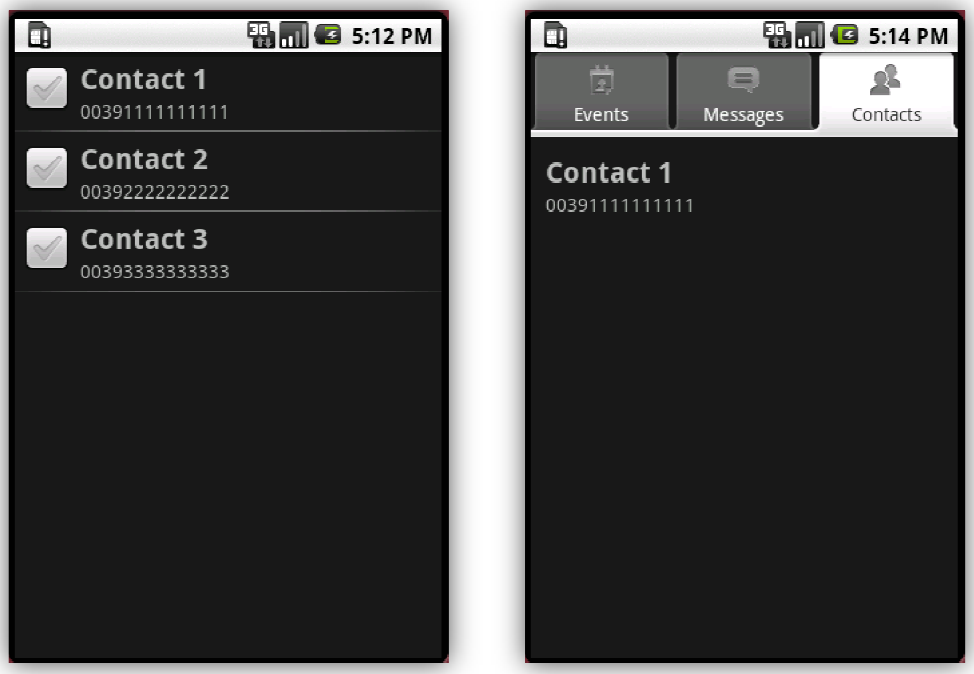

Figura 26. (a) Lista degli utenti disponibili; (b) lista dei contatti personali.

La activity `MainActivity` necessita per prima cosa di conoscere quali siano gli utenti del sistema che possono essere aggiunti come contatti; per semplicità, tutti gli utenti che non siano già contatti personali dell'utente. La richiesta di queste informazioni viene inviata al sistema (o meglio ad un agente di sistema attivo) con il metodo consueto, descritto nel paragrafo precedente, attraverso l'esecuzione del behaviour `GetAllPossibleContactsBehaviour` da parte dell'agente utente. Il contenuto del messaggio ACL di richiesta si presenta nel seguente modo:

```
<?xml version="1.0" encoding="UTF-8"?>
<emsg xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <comm-act type="get-all-users" />
</emsg>
```

L'agente di sistema riceve il messaggio e lo consegna ad una istanza di `SystemListener` perché sia processato: in questa fase, un oggetto `EDBHandler` estrapola dalle basi di dati le informazioni riguardanti tutti gli utenti del sistema; le informazioni vengono convertite in una lista di oggetti di tipo `User`, assegnata ad un oggetto di tipo `EMessage`, che viene trasformato in testo per essere inviato come contenuto di un messaggio. Segue un messaggio di risposta per un sistema a cui sono regitrati gli utenti Contact 1, Contact 2, Contact 3; nessuno di essi è contatto personale dell'utente John Doe.

```
<?xml version="1.0" encoding="UTF-8"?>
<emsg xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <comm-act type="get-all-users" />
      <guest id="2">
            <name>John Doe</name>
            <phone>00390000000000</phone>
      </guest>
      <guest id="5">
            <name>Contact 1</name>
            <phone>00391111111111</phone>
      </guest>
      <guest id="6">
            <name>Contact 2</name>
            <phone>00392222222222</phone>
      </guest>
      <guest id="7">
            <name>Contact 3</name>
            <phone>00393333333333</phone>
      </guest>
</emsg>
```

Quando l'agente utente riceve il messaggio in risposta dal sistema lo inoltra ad un oggetto di tipo `UserListener`, che provvede ad eseguire il parsing del suo contenuto testuale per estrarre le informazioni desiderate, in questo caso la lista degli utenti che possono essere aggiunti come contatti personali. Dalla lista degli utenti così ottenuta, un oggetto di tipo `List<User>`, vengono eliminate le istanze di `User` che rappresentano l'utente che ha fatto la richiesta ed eventuali utenti già presenti tra i suoi contatti personali. Il risultato è presentato nella visualizzazione di `AddContactActivity`.

L'utente può selezionare dall'interfaccia grafica gli utenti che è interessato ad aggiungere alla lista dei propri contatti; questi utenti, rappresentati con delle istanze della classe `User` vengono inseriti nella lista dei contatti dell'oggetto `EMessage` che viene tradotto in formato testuale ed inserito come contenuto del messaggio ACL che viene spedito dall'agente durante l'esecuzione del behaviour dedicato `AddContactBehaviour`. L'agente di sistema che riceve il messaggio lo inoltra al proprio `SystemListener` che analizza il messaggio, riconosce l'atto comunicativo `ADD_CONTACT` e ricrea la lista dei nuovi contatti dell'utente a partire dal contenuto testuale del messaggio ACL; per ogni utente della lista, un oggetto `EDBHandler` inserisce nella tabella `contacts` del database un riferimento tra l'utente richiedente ed il suo nuovo contatto, in pratica una nuova entry che contiene i valori numerici associati ai due utenti. Dopo che l'operazione di scrittura a database è completata, l'agente di sistema invia all'agente utente la lista di tutti i suoi contatti, in modo che l'applicativo client possa visualizzare i dati aggiornati.

Pertanto il messaggio di richiesta sarà del tipo:

```
<?xml version="1.0" encoding="UTF-8"?>
<emsg xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <comm-act type="get-all-users" />
</emsg>
```

mentre la risposta sarà del tutto analoga a quella ricevuta al termine dell'operazione di accesso al sistema.

Promuovere eventi

La promozione di eventi avviene in tre passi:

- inserimento di titolo e descrizione dell'evento che si desidera creare;
- inserimento di una o più date in cui può svolgersi l'evento;
- creazione di una lista di invitati.

I tre passi sono presentati in Figura 27.

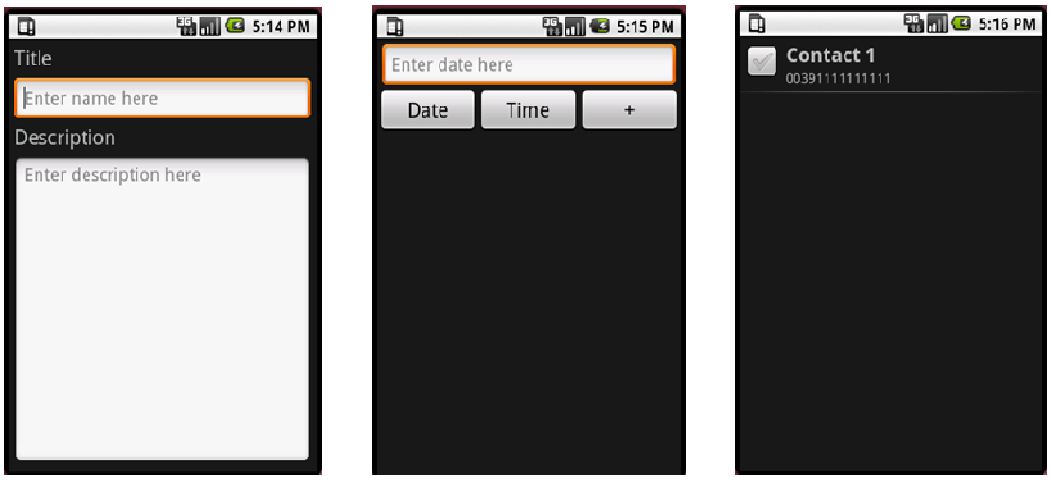

Figura 27. Procedura di creazione di un evento.

L'operazione di creazione di un evento richiede che siano rispettati i seguenti vincoli per poter procedere:

- il titolo dell'evento non può essere lasciato vuoto (la descrizione può essere vuota);
- deve essere presente almeno una data nella lista delle date proposte, e tale data non può essere inserita più volte, né essere passata;
- deve essere presente almeno un contatto selezionato nella lista degli invitati.

Ad ogni passo è necessario premere sul dispositivo il tasto MENU e selezionare l'unica voce di menu NEXT. La data può essere scritta a mano nel formato previsto, oppure impostata tramite l'ausilio dei *dialog* per la selezione della data e dell'ora del giorno, aperti premendo sui pulsanti a schermo `DATE` e `TIME`; la data viene aggiunta alla lista premendo sul pulsante a schermo `+`. La finestra successiva permette di selezionare i contatti che si desidera invitare all'evento (spuntando il relativo *checkbox* sulla sinistra); quando tutti i contatti desiderati sono stati selezionati, si procede premendo il solito tasto `MENU` sul dispositivo e successivamente la voce di menu `GO`; così appare un dialog che chiede la conferma dell'operazione, per concludere la quale bisogna premere il tasto `YES`. La finestra principale è di nuovo in primo piano, e nella lista visualizzata sulla scheda `EVENTS` è presente - se l'operazione è andata a buon fine - il nuovo evento. Supponendo di avere creato l'evento `My event`, il risultato appare come nella figura a lato. Premendo il tasto `NO` la procedura rimane sospesa sulla selezione dei contatti.

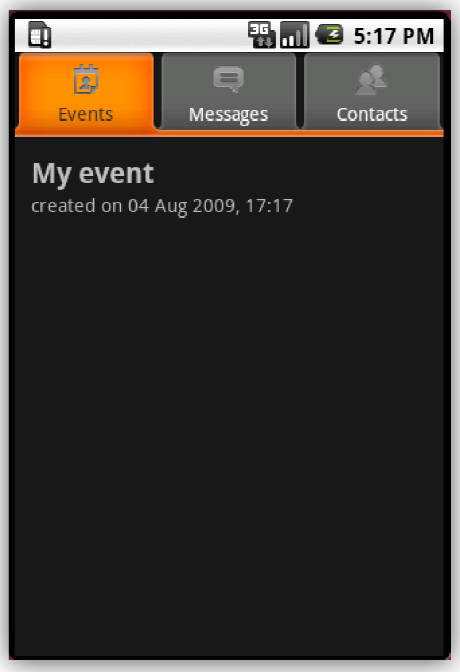

Figura 28. Lista degli eventi creati.

La procedura di creazione di un evento prevede uno scambio di messaggi analogo ai casi precedenti: l'utente effettua la richiesta attraverso l'interfaccia offerta all'applicativo client da `MainActivity`, e la activity richiede di eseguire il behaviour `PromoteBehaviour`; la classe `PromoteBehaviour` ha un attributo statico di tipo `Event` che serve a memorizzare i dati relativi all'evento che si desidera creare, e viene inserito nella lista di eventi presente nella classe `EMessage` che rappresenta il contenuto del messaggio ACL. Come per i casi precedenti, la classe `EMessage` viene trasformata in testo in formato XML dalla classe `LiteBuilder`, ed aggiunto in questa forma al contenuto del messaggio ACL di richiesta che deve essere inviato al sistema (ad uno dei suoi agenti). Il

messaggio ACL ricevuto dall'agente di sistema viene inoltrato a `SystemListener`, che dopo aver verificato che il messaggio esprima l'atto comunicativo `PROMOTE` e che contenga di fatto i dati relativi ad un evento, effettua le query necessarie sul database per inserire nelle tabelle l'evento creato.

L'oggetto `Event`, che rappresenta il nuovo evento che l'utente desidera creare e promuovere, contiene una lista di oggetti di tipo `java.util.Date` e una lista di oggetti di tipo `Participation`. La prima lista rappresenta l'insieme delle date proposte perché l'evento abbia luogo, la seconda serve soltanto a ricordare quali utenti siano stati invitati all'evento. Perciò le informazioni relative all'evento, come titolo e descrizione, vengono inserite nella tabella `events`, mentre le entry della tabella `participations` vengono create facendo il prodotto cartesiano tra gli insiemi delle date e degli utenti invitati; ad esempio, se l'insieme delle date proposte è il seguente:

```
{ date 1, date 2, date 3},
```

e l'insieme degli utenti invitati è il seguente:

```
{ user 1, user 2 },
```

le possibili combinazioni delle partecipazioni risultano essere:

```
{ (date 1, user 1), (date 1, user 2), (date 2, user 1), (date 2, user 2),
(date 3, user 1), (date 3, user 2) },
```

poiché ogni utente invitato può esprimere la propria preferenza (partecipazione o assenza) per ciascuna data proposta per l'evento. Ciò vale sia nel caso in cui l'evento si svolga in una data precisa, scelta dal promotore tra quelle proposte, sia nel caso in cui l'evento necessiti di svolgersi in un determinato numero di sessioni, dove ogni data rappresenta un possibile incontro. Il prodotto cartesiano di date ed invitati viene svolto a lato server per evitare il passaggio di un insieme di dati troppo ampio.

Il messaggio inviato in risposta all'agente utente contiene la lista aggiornata degli eventi creati dall'utente, pertanto è analogo a quello inviato durante la procedura di accesso.

Visualizzazione degli inviti

I messaggi di invito sono visualizzati nella seconda scheda della finestra principale (`MESSAGES`). Il solo tipo di messaggi ricevuto - e di conseguenza visualizzato - nella versione corrente è l'invito a partecipare ad un evento.

Supponendo di impersonare l'utente `Contact 1` dei casi precedenti, che ha ricevuto un invito per l'evento `My Event` dall'utente `John Doe`, la lista dei messaggi di `Contact 1` si presenta come nella seguente figura a sinistra; cliccando sull'oggetto della lista si visualizzano i dettagli relativi all'evento stesso, trovandosi davanti una situazione come quella in Figura 29 (b).

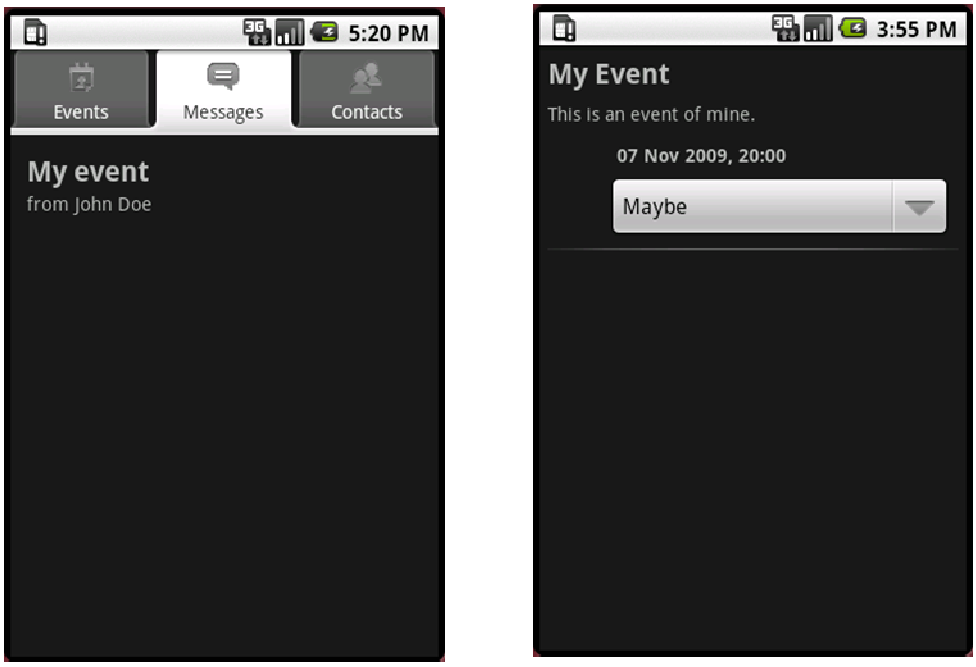

Figura 29. (a) Lista degli inviti; (b) contenuto di un invito.

Gli eventi cui l'utente corrente è invitato divengono noti durante la fase di accesso. L'agente utente esegue un behaviour di tipo `GetInvitationBehaviour`, che invia all'agente di sistema un messaggio ACL di richiesta contenente l'atto comunicativo `GET_INVITATIONS`; il messaggio ricevuto dall'agente di sistema viene inoltrato al suo `SystemListener`, che provvede a ricavare le informazioni dal database, creare degli oggetti di tipo `Event` da inserire nella lista di `EMessage` successivamente tradotti in formato XML ed incapsulati nel messaggio di risposta. Il messaggio ricevuto dall'agente utente ed inoltrato al suo `UserListener` vede nuovamente la trasformazione del suo contenuto testuale in oggetti di tipo `Event` che vanno ad aggiornare la lista degli eventi a cui l'utente è stato invitato, visualizzabile nella seconda scheda della activity principale, `MainActivity`, presentati a schermo come in Figura 29 (a).

Impostare e visualizzare le preferenze

Un invito ricevuto contiene informazioni riguardo l'evento in questione, titolo, descrizione e soprattutto una serie di date proposte perché l'evento abbia luogo. Per ogni data è indicata la propria preferenza personale allo stato corrente, modificabile espandendo il menu a tendina e selezionando un nuovo stato. Parallelamente, dal lato dell'utente promotore, per ogni data è indicato graficamente il numero di preferenze ricevute. Ogni presenza all'evento che viene confermata da un partecipante incrementa il conteggio delle preferenze ricevute per una specifica data. L'invitato non può visualizzare le preferenze esposte dagli altri invitati, in modo che la sua preferenza sia fatta in modo del tutto obiettivo.

Supponiamo che l'utente `Contact 1` decida di confermare la propria presenza per l'unica data proposta per l'evento My Event: questi modificherà il proprio stato come in Figura 30 (a). Scegliendo dal menu a tendina di partecipare all'evento e sottomettendo al sistema il cambio di stato (premendo in sequenza il tasto `MENU` sul dispositivo e selezionando la voce `GO` dal menu), viene aggiornato istantaneamente anche il numero delle preferenze per quella data per l'utente John Doe, promotore dell'evento, che vedrà la barra di progresso delle preferenze per la data in questione interamente colorata, come nella schermata ritratta in Figura 30 (b).

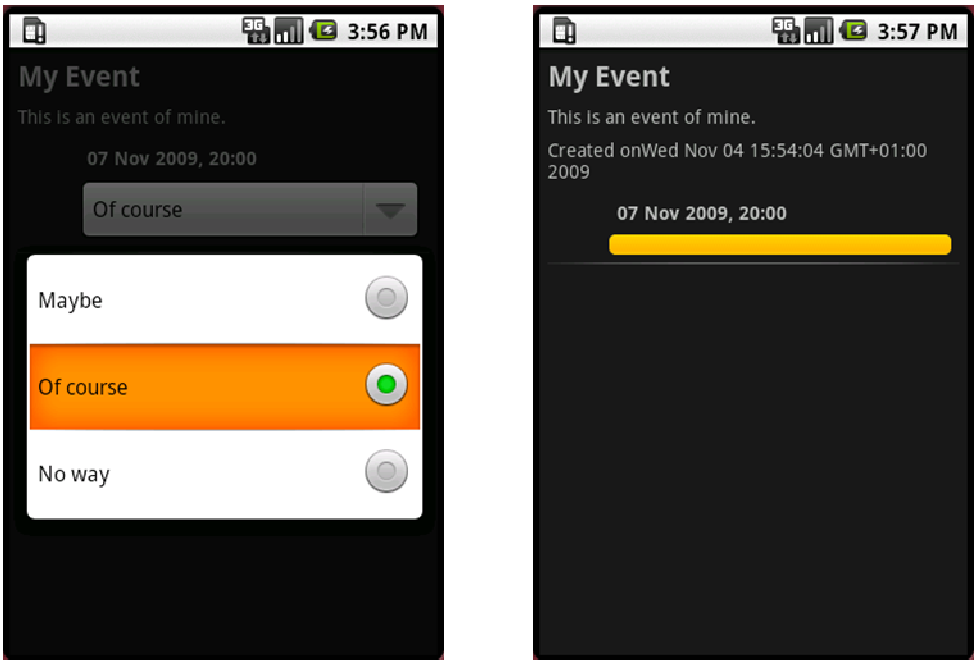

Figura 30. (a) Opzioni di partecipazione; (b) contatore dei partecipanti.

L'impostazione delle preferenze per un evento avviene mediante uno scambio di messaggi, analogo ai casi precedenti. I messaggi scambiati (di richiesta e risposta) presentano entrambi l'atto comunicativo `ACCEPT_REFUSE`.

Il messaggio di richiesta viene spedito dall'agente utente a quello di sistema come unica azione del behaviour `AcceptRefuseBehaviour`; l'oggetto di tipo `EMessage` incapsulato nel messaggio di richiesta in forma di testo contiene soltanto l'evento per cui l'utente invitato ha espresso la propria preferenza: l'oggetto `Event` relativo contiene soltanto gli oggetti `Participation` relativi all'utente, con lo stato della preferenza (partecipazione, assenza o dubbio) espresso per ogni data tra quelle possibili.

Il messaggio di risposta contiene invece tutti gli eventi per cui l'utente corrente ha ricevuto un invito, in pratica lo stesso messaggio ricevuto in seguito ad una richiesta di

tipo `GET_INVITATIONS`. La lista degli eventi serve ad aggiornare l'applicativo client con le nuove preferenze espresse dall'utente.

Lo scenario descritto si presenta solamente nel caso in cui promotore ed invitato siano entrambi connessi alla piattaforma quando l'invitato esprime le proprie preferenze. Nel caso in cui l'utente promotore non fosse connesso al sistema riceverà le preferenze dei suoi invitati non appena effettuerà il prossimo accesso.

Come già detto, esistono due chiavi di lettura per ciò che riguarda l'insieme delle date proposte per un evento.

Nel primo caso, l'utente che desidera promuovere l'evento, desidera raccogliere il maggior numero possibile di consensi (e quindi di partecipanti), perciò rende disponibili un insieme di date su cui ciascun utente invitato stabilisce delle preferenze; una possibilità è che l'utente promotore scelga come data definitiva quella per cui il conteggio delle preferenze ha il valore più alto.

In una chiave di lettura alternativa, l'utente desidera conoscere il numero di consensi per un evento che si ripete nel tempo, periodicamente o sporadicamente. Il medesimo meccanismo precedentemente descritto permette all'utente promotore di conoscere in quali date l'evento può essere tenuto o meno. Ad esempio, un utente promotore desidera organizzare una serie di partite di calcio a cinque per un'intera stagione di allenamenti; per questo richiede la disponibilità della propria squadra per due sere in settimana. Ogni utente dichiara la propria disponibilità per gli allenamenti, e soltanto per le serate in cui sia raggiunto il minimo numero di partecipanti per giocare una partita di allenamento, il promotore conferma che l'evento ha luogo.

Scelta della data finale

Un utente promotore sceglie, secondo una propria logica, quale debba essere, tra le date proposte perché l'evento abbia luogo, quella *decisiva* o *finale*. La logica più semplice potrebbe essere designare come data finale dell'evento quella per cui il maggior numero di partecipanti ha espresso la propria preferenza. Una volta scelta quella che deve essere la data finale, il promotore la seleziona nell'elenco delle date proposte per l'evento creato; per essa sono mostrati: il numero degli invitati, e la lista dei nomi dei partecipanti che hanno espresso la loro preferenza per la data in questione, come si può vedere in Figura 31.

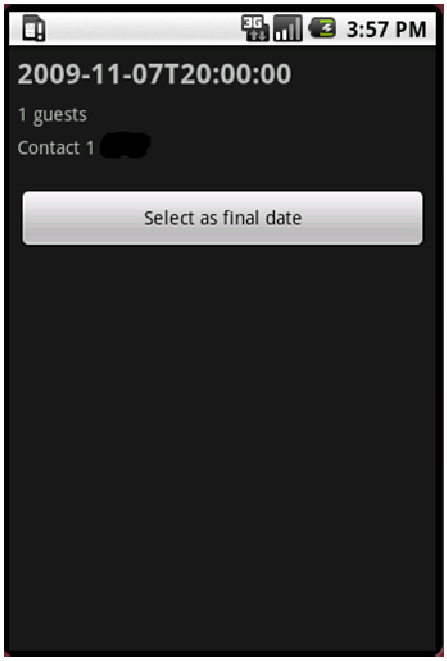

Figura 31. Impostazione di una data come *finale*.

Quando viene premuto il pulsante a schermo `Select as final date`, l'agente utente esegue un behaviour `SetFinalDateBehaviour`. Il behaviour invia a un agente di sistema selezionato casualmente un messaggio di richiesta di tipo `set-final-date`, per cui l'agente di sistema esegue un'azione precisa: impostare come data finale quella suggerita, ossia modificare nella tabella `events` del database relazionale il campo `id_date`, impostando come valore del campo l'identificatore della data selezionata come finale. Nel caso in cui l'operazione venga effettuata più volte il valore obsoleto viene sostituito: in altre parole, un evento non può avere più di una data finale.

Nel momento in cui un evento con data finale è richiesto, ad esempio perché un utente desidera conoscere i dati inerenti agli eventi che sta organizzando, oppure perché desidera conoscere i dati inerenti agli eventi cui è stato invitato, l'istanza corrispondente di `Event` contiene anche un valore maggiore di zero nel campo `finalTime`. Le date finali sono contrassegnate con un'icona a lato nelle liste delle date proposte, sia per gli eventi creati sia per gli inviti ricevuti, come si può vedere in Figura 32.

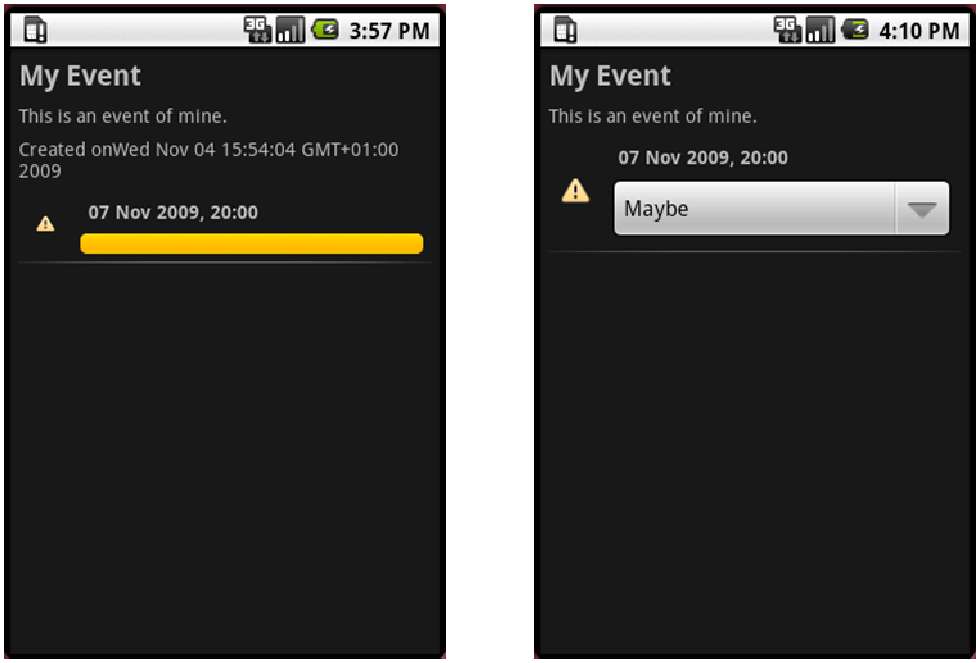

Figura 32. Visualizzazione di una data finale per il promotore (a) e per il partecipante (b).

Abbiamo supposto che l'utente `John Doe` abbia segnalato come data finale la sola presente per l'evento `My Event`: il 7 Novembre 2009. L'icona che contrassegna la data finale appare a `John Doe` nella lista delle date per l'evento `My Event` e appare a `Contact 1` nella lista delle date per l'invito ricevuto all'evento `My Event`.

Capitolo 5
Testing e considerazioni generali

Vengono svolte alcune sperimentazioni sull'adempimento dei requisiti funzionali, del requisito di scalabilità e di bassa latenza.

5.1 Ambiente di test e test sulle funzionalità

Lo scenario di test è rappresentato nella figura seguente.

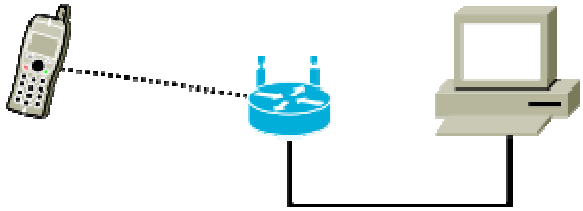

Figura 33. Scenario del test

Segue una lista dello hardware impiegato nel corso del test:

- PC con sistema operativo Linux Ubuntu 9.04 e Android SDK 1.5;
- telefono cellulare Android G1 (HTC Dream) con installato sistema operativo Android aggiornato alla release 3;
- Router wireless Netgear 802.11g.

Il computer è connesso alla rete locale via cavo, mentre il telefono con connessione Wi-Fi. Sulla macchina desktop è in esecuzione la piattaforma JADE versione LEAP SE; nella piattaforma è in esecuzione il container principale della piattaforma, che a propria volta contiene un agente di sistema in attesa di servire eventuali richieste.

Il lato client di MOEVENT è stato installato sul telefono come un'applicazione che può essere avviata dal menu delle applicazioni. Lo smartphone *HTC G1*, il primo Google phone conosciuto anche con il nome di *HTC Dream*, dispone di una connessione wireless che supporta lo standard 802.11g.

Abbiamo svolto innanzitutto una valutazione delle funzionalità e successivamente una valutazione delle performance, misurando i tempi di latenza che un applicativo agente impiega per inviare e ricevere un messaggio ad un altro agente della piattaforma.

I test sulle funzionalità hanno dato dei risultati positivi: l'applicazione permette di aggiungere contatti; di creare eventi e promuoverli presso un insieme di propri contatti; di ricevere inviti ed esprimere le proprie preferenze riguardo ciascuna data proposta per l'evento (presenza, dubbio, assenza); per ogni evento creato e per ogni sua data proposta è possibile monitorare in tempo reale l'andamento del sondaggio.

5.2 Valutazione delle performance

Le performance di un'applicazione si valutano rispetto a delle variabili: le variabili comunemente utilizzate sono l'occupazione della memoria ed il tempo di calcolo. Nell'ambiente mobile possiamo considerare altre variabili, ad esempio la quantità di traffico impiegato per svolgere un'operazione o il consumo della batteria; le motivazioni per introdurre nuove variabili sono dovute alle caratteristiche tipiche dell'ambiente mobile: gli utenti dei telefoni cellulari e degli altri dispositivi mobili generalmente pagano agli operatori una tariffa in relazione alla quantità di traffico generato; la batteria fornisce un'autonomia di poche ore, perciò un'applicazione sviluppata per dispositivi mobili deve evitare di eseguire operazioni non necessarie per minimizzarne il consumo.

Esaminiamo di seguito le caratteristiche delle piattaforme online e le differenze con il progetto che abbiamo implementato.

I dispositivi cellulari possono connettersi ad Internet utilizzando dei servizi offerti loro dagli operatori. Generalmente, un cliente paga una tariffa fissa per disporre di un quantitativo massimo di dati trasmessi su una connessione Internet. Un applicativo deve perciò cercare di minimizzare il traffico inviato e ricevuto.

Supponiamo di creare un evento con Doodle e poi con il nostro sistema.

Collegandosi a Doodle con un dispositivo mobile si scarica la pagina in versione *Mobile Beta*: questa pagina è molto più leggera di quella che viene scaricata da un browser per PC, come Opera o Mozilla Firefox. Eseguendo la procedura di creazione dell'evento vengono inviate richieste HTTP e scaricate pagine web ad ogni passo: la scelta di creazione di un evento, l'impostazione del titolo dell'evento, la scelta delle date, degli orari ed infine la visualizzazione del link ipertestuale univoco che porta alla pagina di gestione dell'evento. Nella figura seguente viene visualizzata la prima pagina web in versione Mobile Beta scaricata durante il processo.

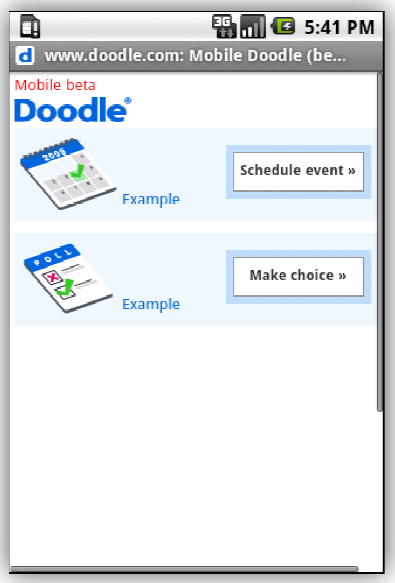

Figura 34. Doodle versione Mobile

Di seguito mostriamo invece la pagina originale: rispetto ad essa, la pagina in versione Mobile è stata ristretta in termini di contenuti e di dimensioni.

Figura 35. Doodle

Inviare richieste http ad un http server e scaricare da esso pagine web (anche se di contenuti ridotti) significa scambiare diversi Kilobyte di dati. I dati scambiati in MOEVENT attraverso i messaggi ACL hanno dimensioni inferiori. Ad esempio, abbiamo calcolato che un oggetto di tipo `EMessage`, utilizzato per rappresentare il contenuto dei messaggi ACL scambiati tra gli agenti della nostra piattaforma e contenente un evento con una data proposta e una lista di ben cento utenti invitati, occupa circa 2 Kilobyte di memoria.

5.3 Overhead JICP

I protocolli utilizzati introducono uno *overhead*, ma esso si può considerare trascurabile. Siccome l'agente utente viene eseguito in modalità *split*, il front end, che giace sul dispositivo mobile, passa un comando serializzato al back end, che si trova sulla piattaforma. Il comando trasmesso al back end è costituito di 3 byte più il messaggio ACL serializzato (con un metodo più efficiente della serializzazione Java). Il comando viene trasportato dentro un pacchetto JICP, che ha un'intestazione di 3 byte, mentre la lunghezza del payload è di 4 byte: lo overhead ammonta a 10 byte.

Per ogni messaggio JICP trasmesso viene ricevuto un ACK che ha una intestazione di 3 byte, un payload di 4 byte ed il payload della dimensione fissa di 2 byte; un ACK aggiunge un overhead di 9 byte.

Al netto del livello TCP lo overhead introdotto da JADE per la consegna del messaggio ACL è di 19 byte. Per ogni pacchetto JICP trasmesso vengono scambiati un pacchetto TCP che trasporta il messaggio stesso e il relativo TCP ACK. In totale, per ogni pacchetto JICP vengono scambiati quattro pacchetti: il mittente invia al ricevente il pacchetto JICP incapsulato in uno o più pacchetti TCP, e per ciascuno di essi riceve un TCP ACK come da protocollo; il ricevente invia un JICP ACK che viene a propria volta incapsulato in un pacchetto TCP per il quale il mittente risponde con un TCP ACK. In conclusione, lo overhead introdotto è trascurabile.

5.4 Misurazioni

Al fine di misurare il tempo impiegato da un agente per l'invio e la ricezione di un messaggio, abbiamo creato un applicativo di test che risponde ad un messaggio ACL rinviandolo al mittente; il mittente è un agente della piattaforma, il destinatario è un agente in esecuzione sul dispositivo mobile. Gli agenti utilizzati per queste misurazioni sono ridotti ad inviare e a ricevere messaggi.

I messaggi trasmessi nel corso del test hanno un contenuto di diverse dimensioni (non conosciamo a prescindere la dimensione del contenuto messaggio, in quanto possono essere trasmessi dati su uno o più eventi ciascuno dei quali contengono liste di utenti e di date di numero variabile) rispettivamente di 1024 byte, 2048 byte, 10 Kilobyte e 20 Kilobyte. Poiché il messaggio ACL viene ritrasmesso dal ricevente verso il mittente, la latenza registrata è stata dimezzata.

In questo modo si sono ottenuti i risultati riportati nel grafico seguente:

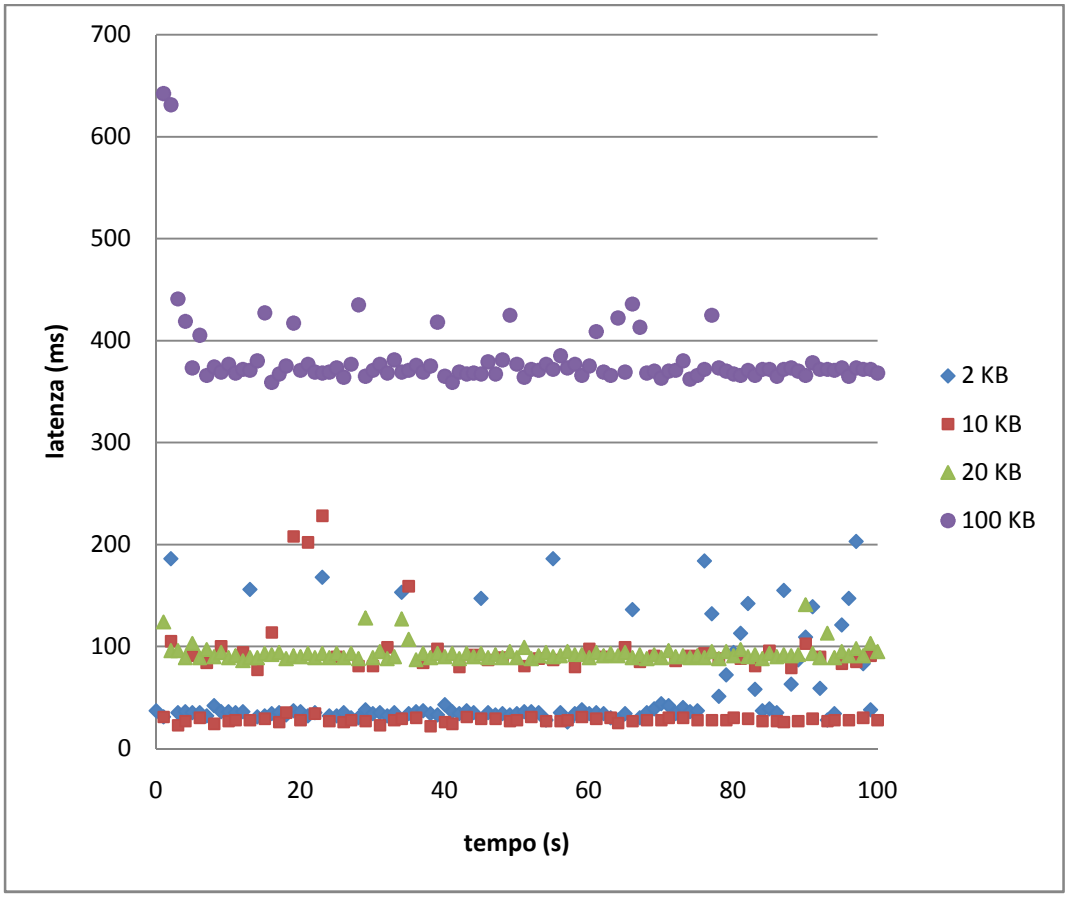

Figura 36. Test sulla latenza

I valori medi riportati per questo test sono i seguenti:

2 KB	28.51 ms
10 KB	59.81 ms
20 KB	93.56 ms
100 KB	382.82 ms

Un secondo esperimento è stato svolto per verificare la scalabilità della piattaforma ad agenti. Allo scopo abbiamo creato degli agenti che scambiano messaggi a degli intervalli prefissati estratti con probabilità uniforme tra 100 e 5000 millisecondi; questi agenti sono detti *rumour agent* perché vociferano per creare del traffico, e il loro numero varia nelle diverse misurazioni da nessuno a dieci, cento e infine mille agenti. Oltre ad essi la piattaforma esegue naturalmente l'agente mittente e ricevente del primo esperimento. La dimensione del pacchetto scambiato corrisponde a 128 byte, per evitare di riempire il buffer dei messaggi.

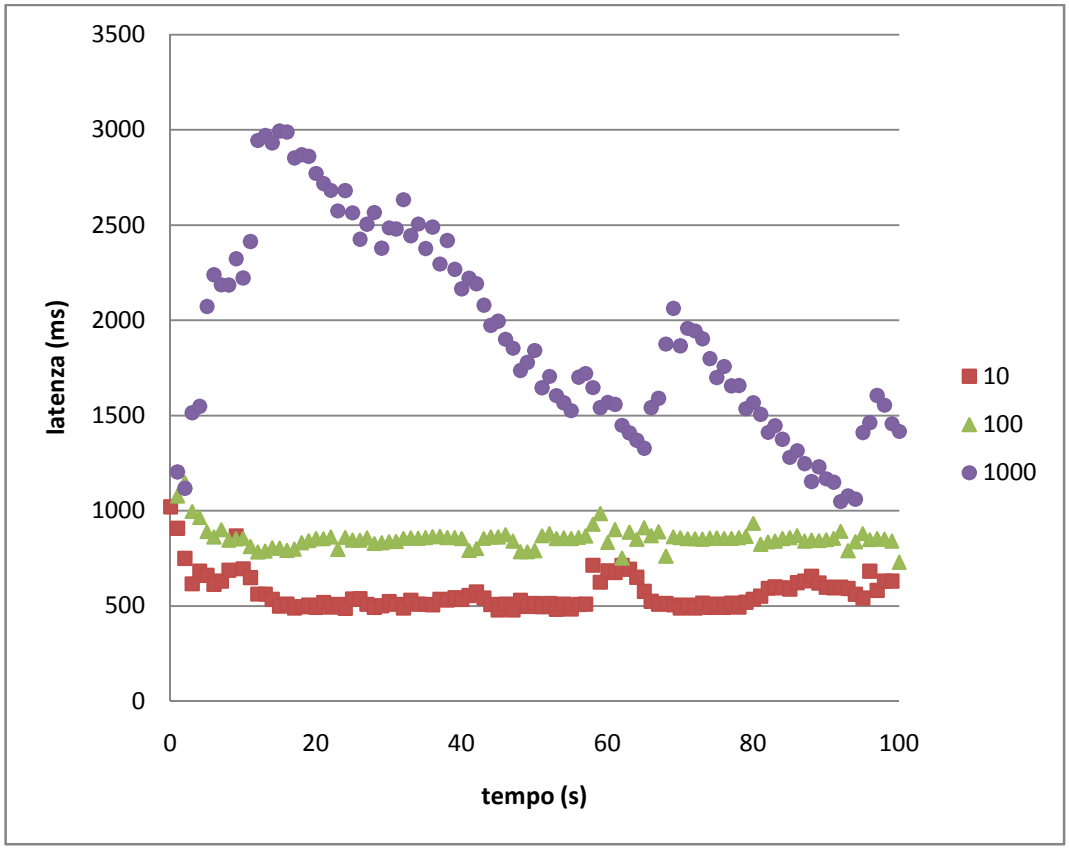

Figura 37. Test sulla scalabilità

Anche in questo secondo esperimento, le latenze misurate sono state dimezzate per le stesse ragioni del caso precedente. I valori medi ottenuti dalle misurazioni effettuate sono i seguenti:

10 agenti	567.58 ms
100 agenti	854.43 ms
1000 agenti	1925.26 ms

Gli agenti aumentano esponenzialmente per potenze di dieci, mentre le latenze aumentano circa per potenze di due. Possiamo concludere che il sistema è scalabile.

Conclusione

Abbiamo ideato il prototipo di un sistema sociale ad agenti, ed implementato la funzionalità di pianificazione di eventi attraverso l'uso della piattaforma JADE estesa ai dispositivi mobili (Android), tramite l'impiego di un plug-in dedicato (JADE LEAP for Android). I punti di forza del sistema sono:

- riduzione dei tempi di comunicazione e una migliore gestione degli eventi creati;
- riduzione dei costi di comunicazione, attraverso una riduzione dei dati scambiati al minimo necessario per effettuare una qualsiasi operazione di gestione;
- memorizzazione dei dati utilizzati dal sistema;
- supporto dell'ambiente mobile, in particolare di Google Android, la piattaforma open che rappresenta l'attuale maggiore innovazione tecnologica nel campo.

Rimangono aperte diverse questioni.

Innanzitutto, MOEVENT è stato un esperimento per verificare se la realizzazione di un social network ad agenti fosse attuabile; poiché i risultati sono stati positivi, MOEVENT può essere esteso con le funzionalità comuni ai social network per offrire un servizio completo. A questo scopo sarebbe importante dotare il sistema di un'interfaccia web.

In secondo luogo, MOEVENT può essere portato verso altre tecnologie cellulari. Ad esempio, i dispositivi cellulari che non appartengono all'ultima generazione e che supportino Java, possono essere facilmente integrati con la piattaforma attuale tramite l'uso di JADE-LEAP e Java Micro Edition (J2ME).

Alcune operazioni nel contesto della pianificazione di eventi possono essere automatizzate, che è il vero scopo dell'uso di un sistema multiagente. Ad esempio, la manifestazione delle preferenze in merito alle date proposte per un evento, operazione svolta manualmente dall'utente, può essere lasciata al suo agente personale, che può decidere di rifiutare automaticamente l'invito a una partita di calcio quando nello stesso orario si svolge un'importante riunione aziendale, oppure accetta automaticamente tutti gli inviti a seminari sul clima. L'agente personale può verificare gli impegni fissati dall'utente consultando la sua agenda sul telefono oppure utilizzando il servizio Google Calendar. La data e l'ora non sono le uniche variabili di interesse per l'agente che si trova a dover decidere se accettare o rifiutare un invito: l'agente può rifiutare automaticamente di partecipare a un evento che ha un costo pro capite troppo alto, o che si svolge troppo lontano.

Un utente potrebbe desiderare di negoziare alcuni parametri che caratterizzano un evento; ad esempio, proponendo di giocare la prossima partita di calcio in un campo con un costo di affitto più basso, o che si trova più vicino, oppure proponendo delle date alternative a quelle stabilite dall'organizzatore.

Infine, la capacità di ragionamento e di cooperazione degli agenti può essere sfruttata per portare a termine procedure complesse. Abbiamo più volte citato l'esempio dell'evento che si svolgerà soltanto se il requisito di raggiungere una soglia minima di partecipanti viene soddisfatto: gli agenti possono cooperare per uno scopo comune, la riuscita dell'evento. In conclusione, l'agente personale di un utente può compiere delle ricerche di quali eventi di interesse per l'utente siano in corso, ed informarlo; la ricerca sfrutta la comunicazione con gli altri agenti.

Bibliografia

[1] C. Lunt, J. Abrams and S. Sanchez, *System and method for managing connections in an online social network*, in United States Patent, Mar, 2007.

[2] Facebook Statistics, http://www.facebook.com/press/info.php?statistics.

[3] Second Life, http://secondlife.com/.

[4] Facebook, http://www.facebook.com/.

[5] MySpace, http://www.myspace.com/.

[6] M. Wooldridge, *An Introduction to Multi-agent Systems*, John Wiley & Sons, 2002.

[7] J. Sabater and C. Sierra, *Reputation and Social Network Analysis in Multi-Agent Systems*, ACM, in International Conference on Autonomous Agents, Bologna (Italy), 2002.

[8] C. Castelfranchi, R. Falcone and F. Marzo, *Being Trust in a Social Network: Trust and Relational Capital*, in Trust Management, Springer, pagg. 19-32, 2006.

[9] J. Lospinoso, I. McCulloh and K.M. Carley, *Utility Seeking in Complex Social Systems: An Applied Longitudinal Network Study on Command and Control*. Social Networks and Multiagent Systems Symposium (SNAMAS), Edinburgh (Scotland), Apr, 2009.

[10] D. Donnetto and F. Cecconi, *The emergence of shared social reprentations in complex networks*. Social Networks and Multiagent Systems Symposium (SNAMAS), Edinburgh (Scotland), Apr, 2009.

[11] T. Grant, *Modelling Network-Enabled C2 using Multiple Agents and Social Network*. Social Networks and Multiagent Systems Symposium (SNAMAS), Edinburgh (Scotland), Apr, 2009.

[12] G. Boella, L. Van Der Torre and S. Villata, *Four Ways to Change Coalitions: Agents, Dependencies, Norms and Internal Dynamics*. Multi-Agent Logics, Languages, and Organisations Federated Workshops (MALLOW) Workshop on Coordination, Organization, Institutions and Norms in agent systems in on-line communities, Torino (Italy), Sep, 2009.

[13] Twitter, http://twitter.com/.

[14] Anyvite, http://anyvite.com/home/.

[15] Evite, http://www.evite.com/.

[16] Pingg, http://www.pingg.com/.

[17] Meeting Wizard, http://www.meetingwizard.com/.

[18] MyPunchbowl, http://www.mypunchbowl.com/.

[19] Doodle, http://www.doodle.com/.

[20] Diarised, http://www.diarised.com/.

[21] Zoji, http://www.zoji.com/.

[22] F. Bellifemine, G. Caire and D. Greenwood, *Developing multi-agent systems with JADE*, John Wiley & Sons, 2007.

[23] Android, http://developer.android.com/guide/index.html.

[24] Open Handset Alliance, http://www.openhandsetalliance.com/.

[25] AndroidLib, http://www.androlib.com/.

[26] G. Caire and F. Pieri, *LEAP User Guide*, TILAB, 2008.

[27] C. Cucè and M. Ughetti, *Analisi e risoluzione delle problematiche di integrazione tra JADE e Android per lo sviluppo di applicazioni distribuite di tipo location-based*, Università degli Studi Mediterranea di Reggio Calabria, Dec, 2008.

[28] D. Gotta, T. Trucco, M. Ughetti, S. Semeria, C. Cucè and A.M. Porcino, (2008), *Jade Android Add-on Guide*, TILAB, 2008.