Progetto e realizzazione di estensioni multipiattaforma per un sistema di amministrazione remota di server basato su messaggistica istantanea

Parole chiave: **Amministrazione di sistemi**

Messaggistica istantanea

Sicurezza

Framework OSGi

Sommario

Introduzione

Al giorno d'oggi l'amministratore di sistema ha a disposizione parecchi strumenti che lo rendono capace di vigilare sulla propria rete attraverso terminali remoti. I servizi offerti dal demone Secure Shell su Linux e dal Desktop Remoto di Windows sono indispensabili nel caso in cui si voglia interagire su un sistema che non è fisicamente presente. Può capitare, però, che un guasto si verifichi in un momento in cui l'amministratore non ha a disposizione un client di amministrazione remota e quindi non è in grado di intervenire prontamente.

I servizi di messaggistica istantanea sono diventati uno strumento di comunicazione sempre più diffuso negli ultimi anni. I proprietari dei principali protocolli hanno cercato di garantire agli utenti un uso sempre più continuativo dei propri servizi portando i client su diversi sistemi operativi e su diverse piattaforme. È sempre più facile incontrare servizi di questo tipo anche su telefoni cellulari e dispositivi palmari, ma anche sui normali computer presenti negli internet point in giro per il mondo.

Grazie al progetto RoboAdmin un amministratore può sfruttare la diffusione dei client di messaggistica istantanea per amministrare un sistema in remoto anche in casi di urgenza. RoboAdmin si presenterà all'amministratore come un normale utente di un servizio di chat. Con semplici messaggi istantanei sarà possibile interagire con la macchina remota come se ci si trovasse di fronte ad una shell.

La presente tesi riprende il progetto RoboAdmin già funzionante con il servizio di comunicazione IRC e lo estende dal punto di vista della molteplicità dei servizi di comunicazione che possono essere utilizzati per interagire con il sistema. In questo modo viene fornita all'amministratore una varietà di canali di comunicazione con caratteristiche differenti, ognuno dei quali può essere abilitato o disabilitato in qualsiasi momento se ha subito attacchi da parte di utenti non autorizzati, se è richiesta una particolare velocità di risposta, oppure se è necessario un particolare livello di sicurezza. Meccanismi di autorizzazione dell'utente amministratore sono previsti per fare in modo che RoboAdmin non renda vulnerabile il sistema se contattato da utenti non affidabili. Un amministratore può permettere ad altri utenti l'accesso ai servizi di RoboAdmin e controllare attraverso i log se si sono verificati tentativi di attacco al sistema.

Il **Capitolo 1** descrive più nel dettaglio gli scopi dell'applicazione, mostrando la struttura di RoboAdmin e il contenuto del data base. Verranno illustrate brevemente anche le procedure per compilare i sorgenti e per configurare e avviare l'applicazione.

Il **Capitolo 2** illustra i principali obiettivi che si è cercato di raggiungere per ogni servizio di comunicazione e, per ciascun servizio, specifica le caratteristiche delle librerie utilizzate, le funzionalità effettivamente implementate e le modalità di installazione.

Il **Capitolo 3** mette a confronto le caratteristiche dei bundle di comunicazione creati per RoboAdmin, in modo che all'amministratore che ne vuole fare uso siano chiari i pro e i contro di ciascuno di essi. Per aiutare nella scelta dei servizi di comunicazione più adatti agli scopi dell'amministratore, sono stati riportati i risultati di alcuni test di funzionamento.

Il **Capitolo 4**, infine, riassume i risultati ottenuti e ipotizza alcuni dei possibili sviluppi futuri del progetto.

Capitolo 1. Progetto RoboAdmin

Il progetto RoboAdmin [01] è nato con lo scopo di mettere un amministratore in contatto con il proprio sistema attraverso un normale canale di chat. Già dalla sua precedente versione RoboAdmin si presenta come un intermediario tra una shell sul sistema locale e un utente amministratore collegato in remoto via chat. Nella versione precedente erano già presenti un servizio di interpretazione dei comandi, un gestore della sicurezza, un servizio Data Base, un servizio di logging e una intelligenza artificiale che interviene nel caso in cui i comandi inseriti dall'utente remoto non siano stati riconosciuti. L'unico servizio di comunicazione presente nella precedente versione è quello relativo al protocollo IRC.

L'intero progetto è stato pensato per essere assemblato basandosi sul framework modulare Apache Felix. Questa struttura ha agevolato l'aggiunta successiva di ulteriori servizi di comunicazione.

1.1 Obiettivi del progetto

Il progetto RoboAdmin deve permettere ad un amministratore di sistema di comunicare con la shell locale attraverso uno qualsiasi dei servizi di chat supportati.

Principale obiettivo di questo progetto è la protezione della porta di amministrazione del sistema: tale porta può essere raggiunta da un client di amministrazione remota per permettere ad un amministratore di impartire comandi per la shell da un terminale remoto. RoboAdmin è nato per diventare un client di amministrazione sostitutivo: grazie alle sue funzionalità si può tranquillamente disabilitare la classica porta di amministrazione, ormai non più necessaria. Questo permette di rendere inefficaci i più classici tentativi di attacco al sistema, come ad esempio quelli di forza bruta. Grazie a RoboAdmin l'attaccante non è in grado di prevedere l'infrastruttura di rete sottostante e si può trovare in difficoltà nel tentativo di riconoscere il server di suo interesse in un servizio popolato da moltissimi utenti contemporaneamente.

Altro importante obiettivo è quello di rendere il sistema accessibile dall'esterno sfruttando la maggior parte dei servizi di messaggistica istantanea disponibili. In questo modo si aumenta considerevolmente la disponibilità del sistema da parte dell'amministratore: anche in caso di malfunzionamenti su un canale di comunicazione, il sistema potrà comunque rimanere accessibile attraverso altri canali, e in caso di attacco l'amministratore potrà comunque interagire per prendere le necessarie contromisure.

Per permettere una maggiore omogeneità e una migliore facilità d'uso di RoboAdmin, le differenze tra un canale di comunicazione e l'altro devono essere ridotte al minimo, anche se le infrastrutture di rete sottostanti sono radicalmente diverse.

1.2 Politiche di sicurezza

Nel campo dell'amministrazione dei sistemi in remoto sono stati fatti enormi progressi per quanto riguarda la sicurezza della comunicazione con il server. Questo si è reso necessario per contrastare l'aumento in numero e complessità dei casi di attacco da parte di malintenzionati.

Il trasferimento di dati tra client e server può contenere informazioni che è necessario proteggere da eventuali intercettazioni passive. I principali strumenti di amministrazione remota, primo tra tutti il servizio Secure Shell, riescono con ottima approssimazione a garantire la sicurezza della comunicazione grazie alla crittografia. In particolare un buon client di amministrazione remota deve assicurare riservatezza e integrità dei dati trasferiti: in questo modo si cerca di evitare l'intercettazione (attacco di tipo passivo) e l'alterazione delle informazioni (attacco di tipo attivo). È anche importante garantire la paternità delle informazioni ricevute dal server (chi invia un dato deve essere un utente noto e autorizzato) e il client necessita di conoscere l'identità del server (per evitare di comunicare con un attaccante invece che con il legittimo destinatario).

In RoboAdmin la sicurezza della comunicazione dipende strettamente dal servizio di comunicazione che l'amministratore intende adottare per comunicare col server. Le politiche di sicurezza di un servizio di messaggistica istantanea possono offrire molte meno garanzie rispetto ad un servizio di amministrazione remota, ed è quindi importante per l'amministratore conoscere le differenze tra un protocollo di comunicazione ed un altro, nel caso intenda trasferire password o dati sensibili utilizzando RoboAdmin.

In generale si è cercato di nascondere RoboAdmin ad utenti non autorizzati, ma talvolta alcuni servizi di comunicazione rendono visibile un utente a tutta la comunità che utilizza quei servizi: nel caso un amministratore intenda comunque utilizzare canali di questo tipo sono state previste ulteriori barriere di protezione, che comunque non pregiudicano l'usabilità e la naturalezza della comunicazione. L'amministratore avrà quindi a disposizione un comando di login senza il quale RoboAdmin si comporterà come un normale utente del servizio di chat, grazie ad un sistema di intelligenza artificiale. Nel caso in cui l'amministratore voglia tenere sotto controllo gli accessi al sistema è stato reso disponibile un servizio di logging.

1.3 Struttura dell'applicazione

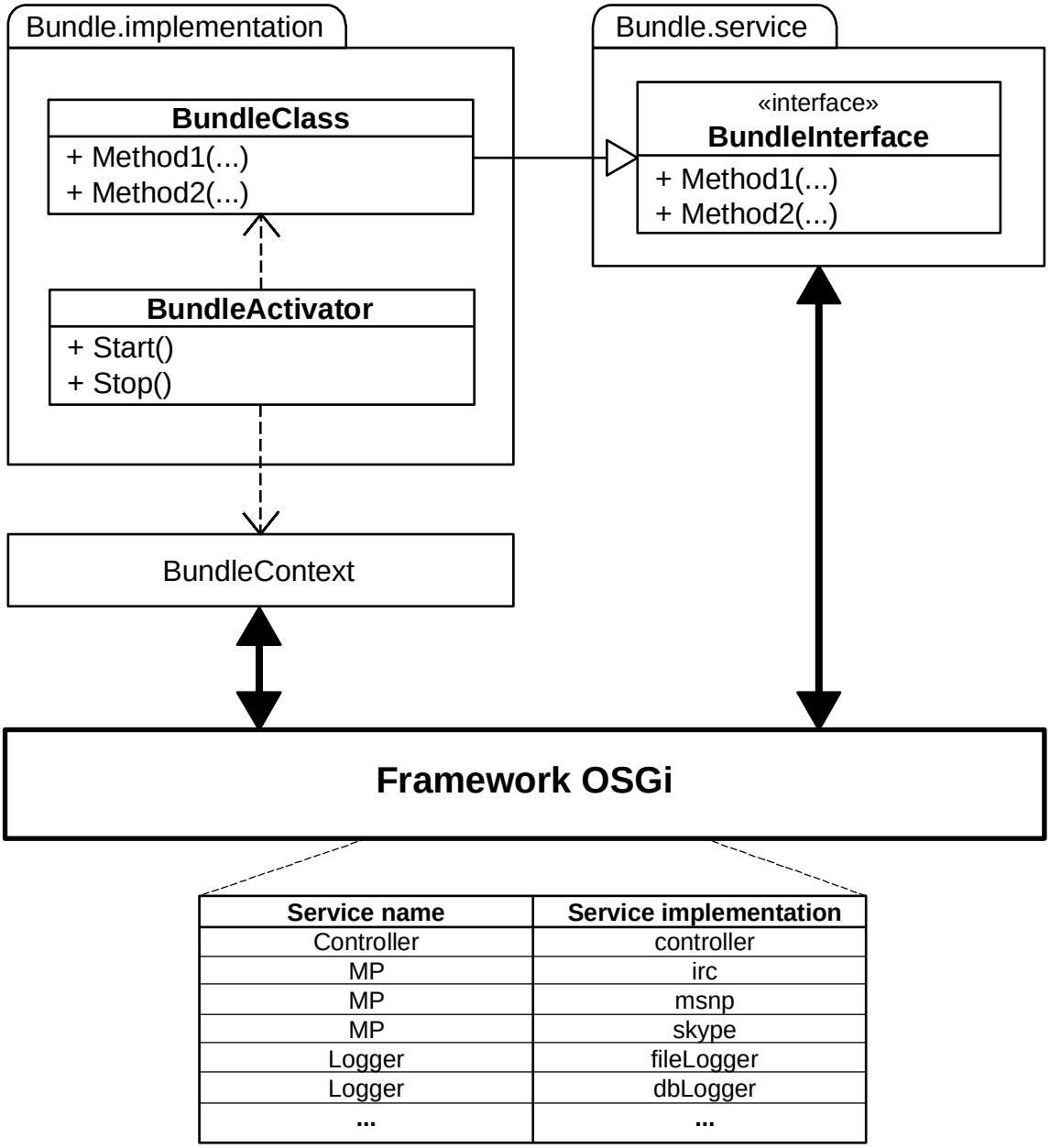
Il servizio RoboAdmin è basato interamente sul framework modulare Apache Felix. In questo modo i servizi di comunicazione possono essere avviati o disabilitati anche durante l'esecuzione dell'applicazione, e nuovi plug-in possono essere aggiunti anche in momenti futuri.

1.3.1 Framework Apache Felix

Data la natura modulare del progetto, è stato scelto il framework Apache Felix [02] come base su cui installare i componenti costitutivi di RoboAdmin. Il framework è un'implementazione Java-based delle specifiche dettate dalla Open Services Gateway Initiative (OSGi) [03], nate nel campo della *Home automation* ma ormai molto diffuse anche in ambito informatico, per applicazioni sia desktop che server.

Il framework gestisce automaticamente l'installazione e l'attivazione dei componenti all'avvio dell'applicazione. È comunque possibile installare dinamicamente nuovi componenti a run time e gestire il loro funzionamento tramite interfaccia testuale o via codice.

Ciascun componente, detto bundle, è costituito da un unico pacchetto JAR, contenente le classi necessarie al raggiungimento di uno specifico obiettivo dell'applicazione. Una volta attivati, i bundle possono comunicare tra loro: le dipendenze tra bundle vengono specificate dal programmatore e salvate nel JAR grazie ad un file di testo detto manifest. Se il bundle che si intende avviare dipende da altri bundle già installati in Felix, il framework cercherà di attivare anche gli altri automaticamente.



Service name	Service implementation
Controller	controller
MP	irc
MP	msnp
MP	skype
Logger	fileLogger
Logger	dbLogger
...	...

Figura 1: interfacciamento di un bundle con il framework OSGi

1.3.1.1 Configurazione di Apache Felix

Il framework è estremamente versatile e può essere usato in parecchi contesti diversi. Per permettere al programmatore che ne fa uso di personalizzarlo in base alle proprie necessità è stato predisposto il file di configurazione *config.properties*, presente nella directory *conf*, che contiene tutte le proprietà commentate dettagliatamente e con formato *chiave=valore*.

Le proprietà utili a RoboAdmin sono essenzialmente tre:

- `org.osgi.framework.system.packages.extra` permette di specificare quali package devono essere inclusi nel classpath di Felix. Attraverso questa

proprietà sono stati specificati tutti i package contenenti le classi delle librerie richieste per il funzionamento dei bundle.

- `org.osgi.framework.storage.clean` gestisce la politica di svuotamento della cache di Felix. Dal momento che Apache Felix è in grado di scaricare automaticamente dei bundle presenti su host remoti, la cache conserva una copia di questi bundle per rendere più veloci i riavvii del framework. Sono comunque presenti le copie dei bundle locali, con alcune informazioni aggiuntive tra le quali l'ordine di caricamento del bundle e il suo stato di attivazione, utili per ripristinare il contesto di esecuzione di Felix nei successivi riavvii di RoboAdmin. In fase di sviluppo dell'applicazione la cache può essere d'intralcio al programmatore, che deve ogni volta aggiornare i bundle che ha modificato usando i comandi della shell di Felix. Grazie alla proprietà `storage.clean` è possibile impostare lo svuotamento automatico della cache all'avvio di Felix (impostando il valore a `onFirstInit`) oppure permetterne il funzionamento normale (impostando il valore `none`, usato come default).
- `felix.auto.start.1` è la proprietà che maggiormente caratterizza il funzionamento di RoboAdmin: qui sono indicati i bundle che devono essere installati e attivati all'avvio dell'applicazione. Per ogni bundle viene indicato il protocollo e il percorso locale o remoto da cui prelevare il bundle. Nel nostro caso tutti i bundle sono locali e il protocollo usato è sempre *file:*. I bundle di comunicazione non sono stati specificati in questo elenco perché il loro avvio viene gestito dinamicamente dal bundle Interprete di RoboAdmin.

1.3.1.2 Struttura di un bundle

I bundle, in base alle specifiche OSGi, sono normali archivi JAR contenenti alcune informazioni aggiuntive usate da Felix per collocare le classi all'interno dell'applicazione.

Le classi appartenenti allo stesso bundle vengono raccolte all'interno dello stesso package. Le informazioni aggiuntive vengono date attraverso un file di manifest costituito da coppie *chiave:valore*. Un esempio di file manifest è riportato di seguito:

```
Bundle-Name: Servizio RoboAdmin
Bundle-Description: Un bundle che rende disponibile le
funzionalità di RoboAdmin
Bundle-Vendor: Apache Felix
Bundle-Version: 1.0.0
```

```
Bundle-Activator: RA.V1.RoboAdminActivatorV1
Export-Package: RA
Import-Package: org.osgi.framework, org.osgi.util.tracker,
RA.service, configurator.service, log.service,
communicationLayer.service, db.service, intelligence.service,
security.service, javax.mail, javax.mail.internet
```

La proprietà Bundle-Activator del manifest si riferisce alla classe all'interno del bundle che ha funzione di attivatore: grazie ad essa il bundle potrà essere avviato o fermato da Felix in base alle istruzioni dell'amministratore. Tale classe deve implementare l'interfaccia BundleActivator, che contiene i seguenti metodi invocabili da Felix:

```
public void start(BundleContext context) throws Exception
public void stop(BundleContext context) throws Exception
```

Tramite il parametro di tipo BundleContext l'attivatore potrà accedere agli altri bundle installati in Felix e recuperare le classi necessarie al proprio funzionamento.

Per scelta progettuale, l'astrazione di un servizio e la sua implementazione sono stati suddivisi su bundle differenti: in questo modo, se alcune funzioni di un bundle richiedono servizi di altri componenti si potrà fare riferimento al bundle astratto, distaccandosi dalla effettiva implementazione del servizio. I bundle di servizio contengono soltanto interfacce e non richiedono alcun tipo di attivazione da parte di Felix.

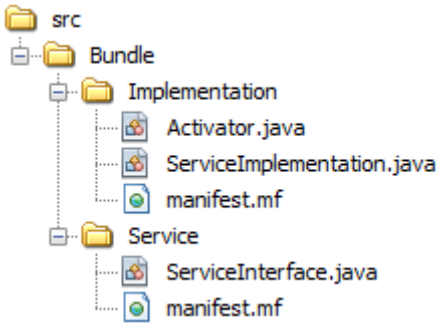


Figura 2: struttura generale di un bundle

1.3.2 Schema generale

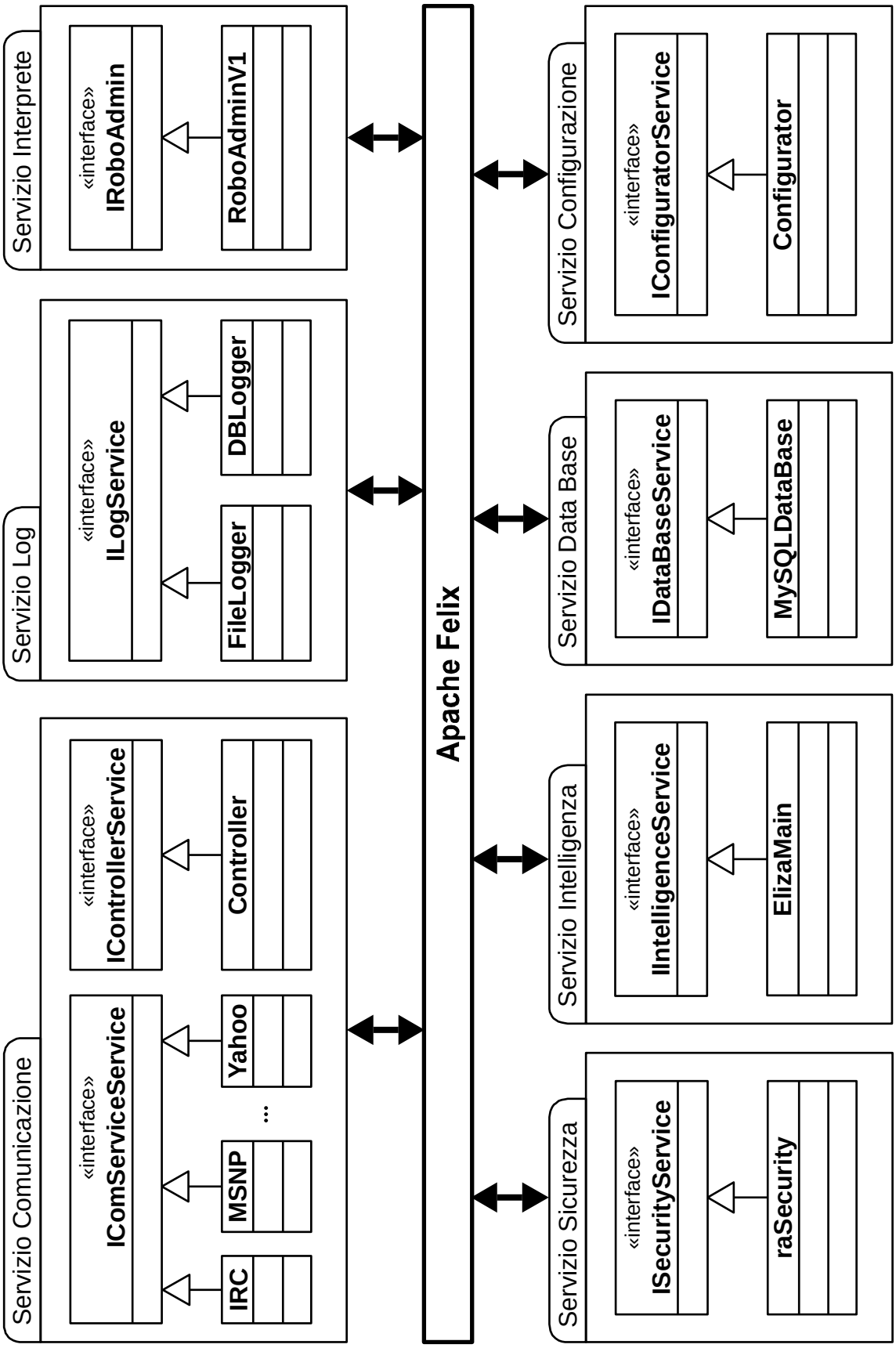


Figura 3: schema sintetico dei bundle di RoboAdmin

1.3.3 Bundle Interprete

La principale responsabilità del bundle interprete è ricevere i messaggi in ingresso, elaborarli e restituire uno o più messaggi di risposta al servizio di comunicazione.

I comandi che possono essere interpretati sono specificati nel file *RoboAdmin.properties* e sono tutti ridefinibili. Il comando *!LOGIN! <username> <password>* permette a RoboAdmin di individuare l'amministratore e consente ad esso di eseguire qualsiasi altro comando.

Il comando sicuramente più utile tra tutti quelli disponibili è *!EXECUTE! <comando>*, che permette all'amministratore in remoto di eseguire comandi di shell attraverso RoboAdmin: ciascun comando viene eseguito su un thread secondario, per evitare lo stallo dell'intera applicazione.

Altri comandi utili all'amministratore e configurabili in *RoboAdmin.properties* sono riportati di seguito:

- *!LOGOUT!* Libera RoboAdmin e lo rende disponibile al login da parte di altri amministratori (o dello stesso amministratore su altri canali di chat).
- *!ID!* Restituisce la versione di RoboAdmin
- *!HI!* Risponde con una stringa di saluto
- *!DIE!* Chiude il canale di chat attualmente in uso
- *!REPEAT! <stringa>* Rimanda al mittente la stringa passata come parametro
- *!DESKTOP!* Attiva il servizio di desktop remoto
- *!AUTORE!* Mostra informazioni sull'autore di RoboAdmin

I comandi non riconosciuti dall'interprete e i messaggi che non contengono alcun comando vengono inoltrati al bundle di intelligenza artificiale, che risponderà opportunamente alle domande che gli vengono poste. In questo modo un utente malevolo non riconoscerà RoboAdmin, nascosto tra parecchi contatti reali di un servizio di chat.

La scelta di quali bundle di comunicazione attivare all'avvio dell'applicazione è responsabilità dell'amministratore: i bundle scelti possono essere specificati all'interno della tabella *MP* sul Data Base *RoboAdminDB.* Al bundle interprete è stata data la capacità di avviare i bundle di comunicazione in base alle scelte imposte dall'amministratore. Grazie a questa funzionalità, in futuro si potrà comandare anche l'avvio e la terminazione dei bundle in remoto inviando opportuni comandi per l'interprete.

1.3.4 Bundle Sicurezza

Il bundle Security è indispensabile per gestire l'autenticazione dell'amministratore: alla ricezione di qualsiasi comando, il bundle security viene richiamato per controllare se l'amministratore si è autenticato. I parametri passati al login sono lo username e la password, verificati all'interno del data base. Se il login o altri tentativi di comunicazione sono considerati sospetti, il possibile aggressore verrà etichettato come "fake administrator" e verranno utilizzati gli strumenti offerti dal servizio di comunicazione per bloccarlo.

Il bundle Security ha anche la responsabilità di evitare che più amministratori tentino di amministrare il sistema contemporaneamente. Dopo ogni tentativo di login riuscito, il sistema diventerà occupato e non verranno accettati altri login. È responsabilità dell'amministratore inviare il comando di logout quando non ha più bisogno di RoboAdmin.

Può capitare che un amministratore abbandoni il servizio senza eseguire il logout. In questo caso, dopo che è trascorso un certo tempo, RoboAdmin risulterà automaticamente libero per nuovi login.

1.3.5 Bundle Intelligenza Artificiale

L'intelligenza artificiale entra in azione se l'interprete riceve dei messaggi che non contengono comandi: in questo caso RoboAdmin deve camuffarsi dagli attaccanti inviando risposte coerenti alle eventuali domande poste dall'utente remoto.

Per implementare questo servizio si è scelta la libreria Eliza, scritta in linguaggio Java. Sono state fatte alcune modifiche per rendere casuale il tempo di risposta all'utente remoto, ed è stata aggiunta la classe Activator per rendere il componente compatibile con il framework.

1.3.6 Bundle Data Base, Log e Configurazione

RoboAdmin può interagire con il data base MySQL [04] attraverso i metodi del bundle *db.mySQL*. Il bundle permette di verificare l'identità dell'amministratore al login, leggere i nomi dei bundle di comunicazione che devono essere attivati all'avvio e verificare se un utente incontrato in chat è autorizzato a comunicare con RoboAdmin oppure no.

Per il momento il data base MySQL è l'unico supportato ma, data la natura modulare di RoboAdmin, non sarà difficile creare in futuro nuovi bundle per interfacciarsi con altri tipi di data base.

Il servizio di logging è usato per mantenere traccia degli accessi e dei messaggi scambiati con l'amministratore. Sono disponibili due bundle che svolgono questa funzionalità: uno salva il log su data base appoggiandosi ai metodi del bundle *db.mySQL* menzionato in precedenza, mente l'altro salva i messaggi di log su file.
Il bundle di configurazione è importante per ottenere informazioni sulla posizione dei file di configurazione, spesso usati per variare il comportamento di RoboAdmin in base alle scelte dell'amministratore.

1.3.7 Bundle di Comunicazione e Controller

I bundle di comunicazione si basano sui più comuni protocolli di messaggistica istantanea per connettere l'interprete di RoboAdmin ai vari servizi di chat.
Problemi di dipendenza ciclica tra package hanno reso necessaria l'introduzione di un bundle chiamato controller, che fa da tramite tra i bundle di comunicazione e l'interprete, nella fase di ricezione dei messaggi. In questo modo, invece che inviare i messaggi ricevuti direttamente all'interprete, il bundle di comunicazione invocherà le funzioni del controller alla ricezione di un messaggio privato.
Il funzionamento dei bundle di comunicazione e le relazioni con il controller saranno oggetto del prossimo capitolo.

1.4 Avvio e utilizzo di RoboAdmin

Di seguito vengono riportate le istruzioni basilari per cominciare ad interagire con RoboAdmin, e le istruzioni su come gestire il funzionamento dei bundle di comunicazione attraverso l'interfaccia utente testuale (TUI) di Apache Felix.

1.4.1 Compilazione del progetto

L'applicazione è stata sviluppata all'interno dell'ambiente di sviluppo NetBeans [05]. Per poter compilare l'applicazione è necessario importare il progetto nell'IDE mantenendo la struttura delle directory, e configurare gli account di comunicazione all'interno della directory *properties*. Per ciascun account è necessario specificare le credenziali di autenticazione e talvolta altri parametri dipendenti dal servizio che si vuole utilizzare (verranno dettagliati nel capitolo successivo).
All'interno della directory *src* si trovano i sorgenti, suddivisi in base al loro bundle di appartenenza.
All'atto della compilazione NetBeans salva automaticamente i file class nella directory *build/classes*, ma non è in grado di aggiornare anche i JAR dei bundle. A questo scopo

sono stati creati due script di nome *bundlebuild*, distribuiti assieme al progetto. Il file con estensione *sh* funziona su piattaforma UNIX, mentre il file con estensione *bat* funziona per sistemi Windows. Una volta aggiornato un bundle potrebbe essere necessario svuotare la cache di Felix, altrimenti verrà caricata comunque la vecchia versione.

1.4.2 Configurazione del Data Base

Per conservare in maniera ordinata alcune informazioni di configurazione e i messaggi di log si è scelto di utilizzare un data base MySQL.

Il data base di interesse per l'applicazione è stato chiamato *RoboAdminDB*: al suo interno si trovano le tabelle *Accept*, *Log*, *MP* e *Users*.

Le tabelle *Log* e *Users* contengono rispettivamente i record di log di RoboAdmin e le coppie username e password usate dal bundle Security per riconoscere l'amministratore.

La tabella *MP* contiene i nomi dei servizi di comunicazione presenti in RoboAdmin. Da qui l'amministratore può abilitare o disabilitare i bundle che verranno installati all'avvio di RoboAdmin. Il nome del servizio indicato nella colonna *nome* deve corrispondere al nome del bundle con estensione JAR presente nella directory *bundle* del progetto in modo che, consultando il data base, il bundle CORE possa avviare automaticamente il servizio specificato.

La tabella *Accept* è nata per permettere a ciascun bundle di comunicazione di riconoscere gli utenti autorizzati a vedere lo stato e ad inviare comandi a RoboAdmin. Per ogni tupla si deve specificare il nome utente e il servizio di comunicazione che l'utente usa per comunicare.

In fase di sviluppo è stato utilizzato il data base MySQL inserito nel pacchetto XAMPP [06], che offre anche il servizio di configurazione PHPMyAdmin [07] e il web server Apache. Attraverso le pagine di PHPMyAdmin può essere importato il file (distribuito assieme al progetto) contenente tutte le tabelle di *RoboAdminDB*. Le credenziali per l'accesso al DB vanno specificate nel file *MySqlDB.properties* contenuto nella directory *properties* del progetto.

1.4.3 Avvio di RoboAdmin

Se RoboAdmin è stato importato come progetto in NetBeans è possibile avviare l'applicazione direttamente all'interno dell'ambiente di sviluppo. In caso contrario si potrà avviare dalla shell con il comando "`java -jar bin/felix.jar`".

Felix mette a disposizione dell'utente una ristretta gamma di comandi, utili a gestire il funzionamento dei bundle. Il comando *help* mostra l'elenco completo dei comandi di Felix. È possibile vedere lo stato di attivazione dei bundle con il comando *ps*, ed esistono comandi *start* e *stop* per avviare o fermare i bundle installati. Grazie ai comandi *install*, *uninstall* e *update* è possibile inserire nuovi componenti nell'applicazione, rimuoverli o aggiornarli. In particolare la sintassi del comando *install* richiede una stringa con il protocollo e il percorso del bundle da installare (ad esempio "`install file:bundle/CORE.jar`"). Gli altri comandi non richiedono URL per funzionare: basta inserire il codice identificativo del bundle (visibile grazie al comando *ps*). È possibile uscire dall'applicazione con il comando *shutdown*.

1.4.4 Installazione di un bundle

Per poter utilizzare le librerie necessarie alle classi del nuovo bundle è importante aggiornare il file di configurazione di Felix come già specificato in precedenza. Se alcune librerie non sono state importate si otterranno eccezioni a run time.

Dal momento della sua installazione nel framework, il bundle passa attraverso diversi stati di funzionamento, i principali dei quali sono Installed, Resolved, Active e Stopped. Attraverso l'interfaccia utente testuale di Felix è possibile far passare i bundle da uno stato all'altro.

Un bundle deve funzionare solo nello stato Active, ed è responsabilità del programmatore rilasciare ogni risorsa utilizzata e fermare eventuali thread secondari quando Felix invoca il comando stop del bundle activator.

L'attivazione dei bundle di comunicazione deve essere gestita tramite data base: questo è compito del bundle interprete, che una volta avviato si occupa di leggere i nomi dei bundle richiesti dal data base e di attivarli uno dopo l'altro. Per questo motivo non dovrebbe più essere necessario interagire con Felix attraverso la sua interfaccia testuale. Questa possibilità è stata lasciata per permettere operazioni di amministrazione straordinaria, come l'aggiornamento di un bundle di comunicazione o la sua disattivazione in caso di problemi imprevisti.

Capitolo 2. Servizi di comunicazione

In questo capitolo verranno trattati in dettaglio i bundle di RoboAdmin relativi ai servizi di comunicazione. Ognuno di questi bundle si riferisce ad un diverso protocollo di comunicazione e può essere abilitato o disabilitato indipendentemente dagli altri, come specificato nel capitolo precedente.

Dal momento che la maggior parte dei protocolli di messaggistica istantanea è stata oggetto di approfonditi studi da parte della comunità Open Source, sono già disponibili gratuitamente ottime librerie Java che implementano le più importanti funzionalità di ciascun servizio. La scelta di queste librerie al posto di una nuova implementazione ad hoc per RoboAdmin è stata preferita perché offre maggiori garanzie di sicurezza, grazie all'importante community che contribuisce a correggere, ottimizzare e aggiornare costantemente questi componenti.

Tutte le classi principali di questi bundle implementano l'interfaccia *IComService*, che viene usata dalla classe interprete per rispondere ai messaggi e gestire lo stato dei servizi di comunicazione in base ai comandi impartiti dall'amministratore. Il diagramma UML risultante è riportato nella prossima pagina.

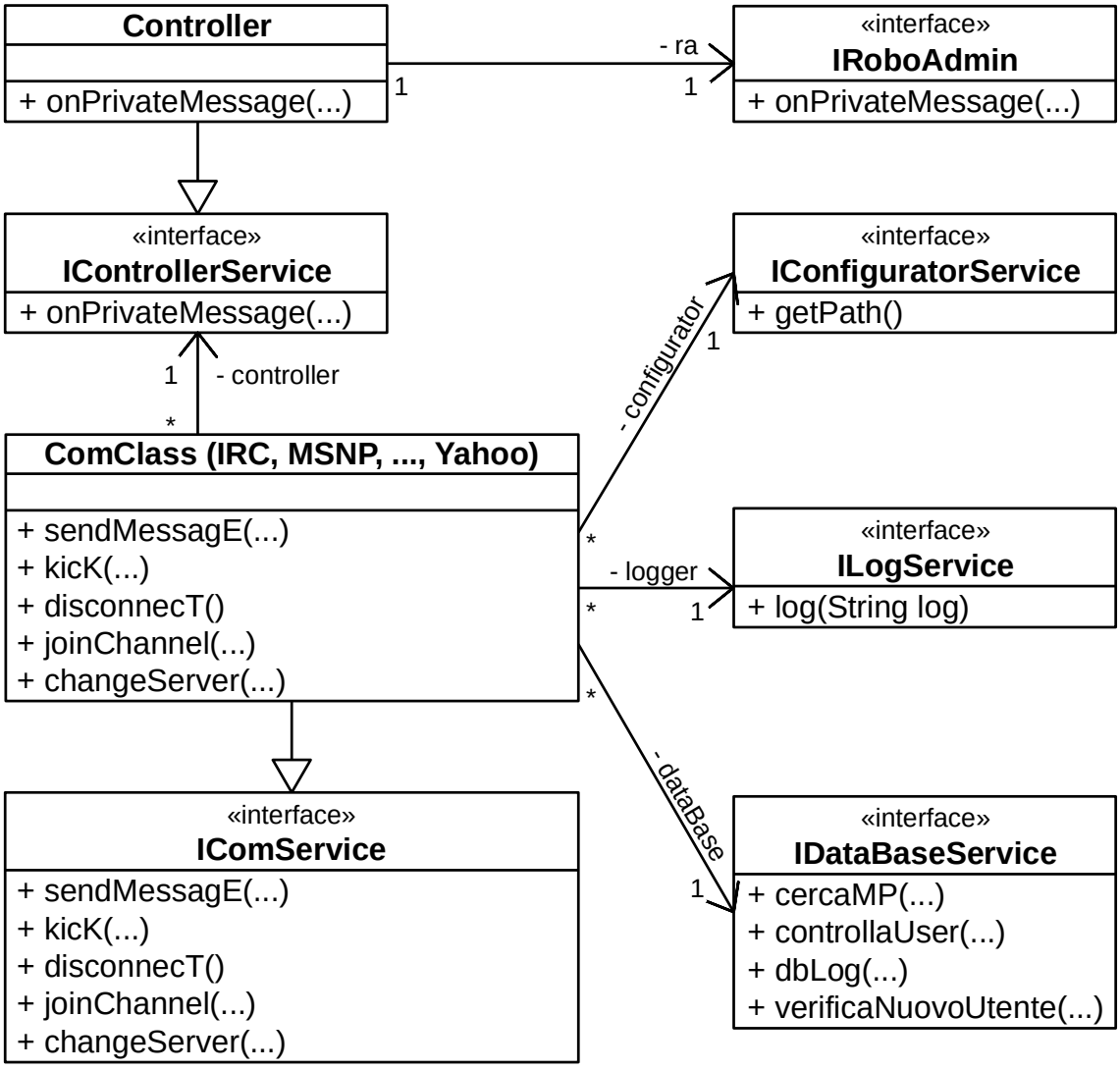


Figura 4: relazioni tra un qualsiasi servizio di comunicazione e le altre classi di RoboAdmin

2.1 Specifiche dei bundle di comunicazione

Per rendere più omogeneo l'utilizzo di RoboAdmin da parte dell'amministratore si è preferito uniformare i requisiti di tutti i bundle di comunicazione.

Il bundle per il servizio IRC, che verrà comunque trattato in un prossimo paragrafo, segue solo in parte le specifiche generali indicate di seguito, dal momento che non prevede un servizio di presenza vero e proprio per il controllo degli utenti autorizzati.

Gli obiettivi perseguiti per ogni bundle di comunicazione sono i seguenti:

- *Login e logout*: al momento dell'attivazione, il bundle deve connettersi al servizio usando le informazioni sull'account salvate dall'amministratore nel file *nome_servizio.properties* presente nella directory *properties* del progetto RoboAdmin. La fase di login è necessaria per connettere RoboAdmin alla rete in uso dal servizio e per renderlo visibile all'amministratore. Alla chiusura di RoboAdmin deve essere garantita la disconnessione dell'utente, secondo le modalità

imposte dal servizio stesso. La disconnessione può rendersi necessaria anche in caso di attacco da parte di hacker o se comandato dall'amministratore.

- *Eliminare la formattazione dei messaggi*: per essere elaborato correttamente dall'interprete, un messaggio ricevuto non deve contenere informazioni sulla formattazione del testo. La formattazione dei messaggi dipende dal servizio di comunicazione in uso. Utilizzare, se necessario, le funzioni di libreria per eliminare la parte del messaggio che non deve essere interpretata.
- *Inviare i messaggi di risposta*: se l'interprete richiede l'invio di uno o più messaggi da parte del servizio di comunicazione, significa che è stato ricevuto un comando in precedenza dallo stesso utente a cui verrà inviata la risposta. Deve essere possibile inviare, senza alterazioni, i messaggi di risposta forniti dall'interprete, al destinatario da esso indicato.
- *Uscire da chat con più utenti*: i messaggi scambiati tra RoboAdmin e l'amministratore devono rimanere riservati. Inoltre non deve essere concesso a più amministratori di comandare contemporaneamente RoboAdmin. Il bundle di comunicazione deve negare gli inviti a chat con più utenti, anche per evitare di esporsi ad utenti non autorizzati.
- *Gestire la visibilità di RoboAdmin*: molti servizi di messaggistica istantanea sono pensati per garantire all'utente un livello di privacy accettabile. Normalmente un utente deve permettere esplicitamente ad un altro utente di vedere i propri dati riservati. I bundle di comunicazione devono sfruttare questa caratteristica, se disponibile, per nascondersi il più possibile ad utenti non autorizzati. Gli utenti autorizzati, al contrario, devono poter vedere lo stato e le informazioni sull'account di RoboAdmin, senza restrizioni di alcun tipo.
- *Bloccare gli utenti non autorizzati*: può capitare che un utente tenti di comunicare con il sistema pur non essendo autorizzato a farlo. Molti dei servizi di messaggistica presi in considerazione permettono di limitare l'interazione con alcuni utenti bloccandoli: un utente bloccato non ha il permesso di vedere lo stato di RoboAdmin e non può comunicare con esso. Questo può tornare utile per fermare attacchi da parte di utenti malevoli che tentano di indovinare la password dell'amministratore (con attacchi di forza bruta). Il blocco dell'utente è da preferirsi rispetto alla chiusura completa del canale di comunicazione, in caso di attacco, per non limitare inutilmente i canali a disposizione dell'amministratore. L'operazione di blocco deve

essere reversibile: se in futuro l'utente viene autorizzato, ogni comunicazione deve essere consentita automaticamente.

- *Gli utenti autorizzati possono sempre comunicare con RoboAdmin*: il fatto che RoboAdmin non sia presente nella lista contatti dell'amministratore non deve impedire la comunicazione tra i due. Se RoboAdmin non è presente tra i contatti dell'utente remoto, nella maggior parte dei casi, quest'ultimo non potrà ricevere aggiornamenti sullo stato di RoboAdmin in tempo reale. Una scelta di questo tipo, pur essendo scomoda, può essere utile per nascondere RoboAdmin ad altri utenti che accedono al servizio di chat usando lo stesso account dell'amministratore.

2.2 Interazione con altri bundle

Ciascun bundle di comunicazione deve interagire con altri bundle per portare a termine i propri obiettivi. Il bundle interprete deve inviare le risposte e gestire il canale di comunicazione in base ai comandi impartiti dall'amministratore. Il bundle controller ha il compito di ricevere i messaggi in arrivo.

L'interfaccia *IControllerService*, usata dal controller, contiene soltanto il metodo:

```
public void onPrivateMessage(IComService service, Object sender,
String message)
```

Grazie a *onPrivateMessage* sarà possibile inviare un messaggio attraverso il parametro *message* (testo semplice, senza informazioni sul formato) all'interprete. Questo risponderà invocando le funzioni della classe che implementa *IComService* (passata come primo parametro) che può essere uno qualsiasi dei bundle di comunicazione.

Il parametro *sender* contiene i dati necessari ad identificare l'utente o la sessione di chat su cui spedire un messaggio di risposta. Il tipo del parametro dipende dal servizio di comunicazione in uso.

Il metodo principale dell'interfaccia *IComService* è quello che consente di inviare i messaggi di risposta:

```
public void sendMessagE(Object receiver, String message)
```

Grazie a questo metodo verrà spedito a *receiver* un messaggio in testo semplice.

Altri metodi disponibili in *IComService* sono:

```
public void kicK(String channel, String nick, String reason)
public void disconnecT();
public void joinChanneL(String channel);
public void changeServeR(String server);
```

Questi metodi permettono di bloccare un utente, chiudere la connessione o modificare alcuni parametri del canale di comunicazione. Il loro funzionamento dipende dal servizio di comunicazione in uso.

Nella maggior parte dei bundle è necessario verificare se un amministratore è autorizzato a stare nella lista contatti di un determinato servizio di comunicazione: per questo potrebbe essere necessario l'intervento del bundle data base. La funzione che permette di controllare se un utente è autorizzato è la seguente:

```
public boolean verificaNuovoUtente(String userName, String protocol)
```

Il parametro *userName* è una stringa che si riferisce al nome dell'utente presente nella lista contatti. Il parametro *protocol* è necessario perché un utente potrebbe usare quel nome utente per un servizio ma non per un altro. Il metodo restituisce *true* solo se è stata trovata corrispondenza nel data base, e quindi l'utente è autorizzato.

2.3 Servizio di comunicazione con protocollo IRC

Il bundle IRC si connette a uno o più canali pubblici del server IRC. L'amministratore può comunicare con RoboAdmin rintracciandolo sul canale pubblico e avviando con esso una conversazione in un canale privato.

2.3.1 Protocollo IRC

IRC (Internet Relay Chat) [08] è un protocollo aperto che permette ad utenti collegati ad un server *IRCd* di scambiare messaggi testuali con gli altri utenti dello stesso network. Vari server possono interagire tra loro formando una vera e propria rete: all'interno di essa si possono distribuire gli utenti per formare gruppi di discussione detti canali, di dimensioni anche piuttosto consistenti.

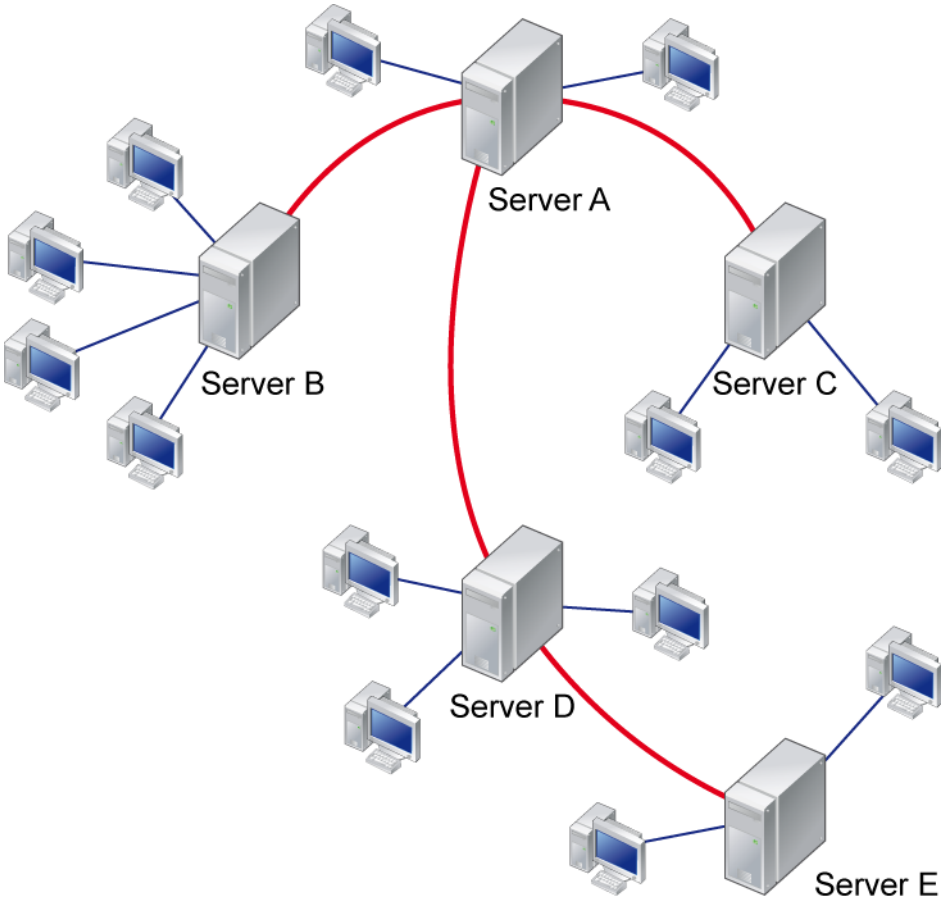


Figura 5: schema di una rete IRC di piccole dimensioni

All'interno di un canale tutti gli utenti hanno la possibilità di scrivere messaggi e leggere messaggi scritti da altri. Possono essere presenti utenti con privilegi particolari detti operatori, che controllano lo scambio di messaggi e sono in grado di bloccare altri utenti che non rispettano i criteri di civiltà imposti dal network. Tipicamente è concessa la creazione di nuovi canali a chiunque voglia usufruire del servizio: in questo caso chi crea il canale ne diventa operatore, e può cedere questo diritto ad altri utenti. Il protocollo prevede l'esistenza di utenti con il massimo livello di privilegi, gli amministratori, che hanno come responsabilità l'amministrazione dell'intero network.

Gli operatori possono assegnare o togliere ad utenti e canali alcune proprietà dette *modes*, rappresentate da singole lettere. Esistono diverse implementazioni del protocollo e in base alla versione del server *IRCd*, il comportamento dei modes può variare leggermente dal comportamento standard.

Nonostante non sia obbligatorio, è possibile per un utente registrarsi presso il network IRC di suo interesse e accedervi tramite una procedura di login con username e password. Questa operazione si rende necessaria per operatori e amministratori, a cui il server deve assegnare il giusto livello di privilegi. Normalmente l'utente viene

identificato all'interno del canale grazie ad un nickname scelto dall'utente stesso e univoco nel network.

I server sono collegati tra loro con struttura ad albero: i messaggi vengono instradati solo ai rami a cui gli utenti del canale sono collegati. Solo lo stato della rete viene ricevuto da tutti i server, che generalmente comunicano con un alto grado di fiducia reciproca.

2.3.1.1 Comunicazione con tra client e server

Per accedere ad una rete IRC è necessario aprire una connessione TCP con uno dei server IRCd disponibili. La chiusura della connessione comporta l'uscita dell'utente dal servizio. La comunicazione con il server, come avviene anche per il protocollo HTTP, consiste nello scambio di stringhe di testo con un certo formato.

Per supportare anche client che non hanno interfaccia grafica, i server possono elaborare alcuni comandi inviati direttamente dall'utente all'interno dei messaggi di chat. Per poter inviare un comando, il messaggio deve iniziare con il carattere di escape '/'. Questi comandi possono essere usati, ad esempio, per ricevere la lista dei canali disponibili, gli utenti collegati o per fare il "join" ad un canale. Gli operatori usano frequentemente questi comandi per impostare i *modes* agli utenti del canale.

Il protocollo IRC è nato per permettere agli utenti di scambiare messaggi liberamente: non vengono prese particolari precauzioni per nascondere la comunicazione tra client e server, e nei canali pubblici chiunque può entrare e partecipare attivamente o passivamente alla discussione. Ultimamente si stanno affermando versioni criptate del protocollo basate su TLS [09]: sebbene molti client le supportino, non tutti i server offrono queste funzionalità.

2.3.2 Libreria PircBot

Il bundle IRC usa la libreria PircBot [10], scritta interamente in linguaggio Java, che implementa le più comuni caratteristiche del protocollo IRC.

La classe *PircBot*, nel package *org.jibble.pircbot*, concentra al suo interno tutti i metodi utili a RoboAdmin, tra i quali quello che apre la connessione con il server e quello che si occupa dell'invio di messaggi istantanei. Per ricevere in modo asincrono le notifiche inviate dal server è necessario estendere questa classe e sovrascrivere i metodi che corrispondono agli eventi che si vuole considerare. Al di fuori dei metodi di notifica, sono presenti soltanto metodi *final*, che non possono essere modificati dalla classe derivata.

2.3.2.1 Metodi di interazione con il server

Di seguito sono stati elencati i metodi della classe PircBot usati per comunicare con il server IRCd.

- `protected final void` **`setName`**`(String name)` imposta il nickname di RoboAdmin. Il metodo deve essere invocato prima della connessione al server, altrimenti verrà usato il nickname di default.
- `protected final void` **`setLogin`**`(String login)` imposta lo username, da usare per autenticarsi sul server.
- `protected final void` **`setVersion`**`(String version)` modifica il nome del client e la sua versione da quelli di default, se invocato prima di connettersi al server.
- `public void` **`setAutoNickChange`**`(boolean autoNickChange)` con il parametro settato a *true* controlla se il nickname richiesto dal server è già in uso e, in caso affermativo, aggiunge un numero sequenziale.
- `public final void` **`setMessageDelay`**`(long delay)` imposta quanto ritardo (in millisecondi) deve esserci tra l'invio di un messaggio e il successivo. Il server considera "spammer" gli utenti che inviano messaggi troppo frequentemente sul canale, pertanto la libreria inserisce automaticamente un ritardo prima di inviare i messaggi istantanei. Il valore di default di un secondo è troppo alto per i nostri scopi, ed è stato dimezzato.
- `public final void` **`connect`**`(String hostname, int port, String password)` apre una connessione con il server passato come parametro. Se *password* vale *null* non verrà eseguita l'autenticazione.
- `public final boolean` **`isConnected`**`()` restituisce *true* se la connessione con il server e aperta.
- `public final void` **`disconnect`**`()` invoca il metodo *quitServer*, per concordare con il server una chiusura dolce della connessione.
- `public final void` **`quitServer`**`()` invia al server la richiesta di disconnessione.
- `public final void` **`joinChannel`**`(String channel)` richiede il join al canale passato come parametro. Se il canale non esiste viene creato, e RoboAdmin ne diventa operatore.
- `public final void` **`op`**`(String channel, String nick)` garantisce all'utente *nick* privilegi di operatore sul canale passato come parametro.

- `public final void` **`sendMessage`**`(String target, String message)` invia un messaggio istantaneo a *target*. Il parametro *target* può indicare il canale pubblico (se preceduto dal carattere '#') o il nickname dell'utente per un messaggio privato.
- `public final void` **`kick`**`(String channel, String nick)` esclude dal canale passato al primo parametro l'utente con il nickname nel parametro *nick*. L'operazione ha successo solo se RoboAdmin è operatore sul canale in questione

2.3.2.2 Metodi di notifica

I pacchetti ricevuti dal server vengono indirizzati ad uno dei metodi di notifica presenti nella classe PircBot. I metodi usati in RoboAdmin distinguono i messaggi ricevuti da canali pubblici e privati.

- `protected void` **`onMessage`**`(String channel, String sender, String login, String hostname, String message)` riceve i messaggi inviati sui canali pubblici in cui RoboAdmin è presente. I parametri consentono di identificare il mittente e di leggere contenuto del messaggio.
- `protected void` **`onPrivateMessage`**`(String sender, String login, String hostname, String message)` riceve i messaggi inviati a RoboAdmin su un canale privato. Per controllare l'affidabilità del mittente vengono forniti il nickname (parametro *sender*), il nome di login e il nome dell'host da cui il messaggio è stato inviato.

2.3.2 Implementazione del bundle IRC

La classe *IRC* del bundle estende la classe *PircBot* e funziona da wrapper, integrando le funzioni della libreria e adattandole ai metodi richiesti dall'interprete.
All'attivazione del bundle viene avviata la connessione verso il server specificato nel file *IRC.properties* (presente nella directory *properties* del progetto) e viene eseguito il join ai canali indicati dall'amministratore nel file *IrcChannelList.txt*. Questa operazione è necessaria per rendere visibile pubblicamente RoboAdmin e per dare all'amministratore la possibilità di individuarlo all'interno del canale. RoboAdmin tenta di imporsi come operatore su ogni canale: se il tentativo non ha successo verrà considerato come un normale utente del servizio. Dopo una eventuale registrazione di RoboAdmin presso il server, è possibile configurare nel file *properties* anche lo username e la password necessari al login.

RoboAdmin non può scambiare messaggi con l'amministratore su un canale pubblico: in questo caso chiunque potrebbe leggere la conversazione e verrebbe svelata troppo facilmente l'identità del bot. Nel caso in cui un utente tenti di contattare RoboAdmin pubblicamente, il messaggio verrà intercettato dal metodo *onMessage*, opportunamente ridefinito. I messaggi pubblici non verranno recapitati al controller perché non devono essere interpretati: alla ricezione di un messaggio pubblico contenente riferimenti a RoboAdmin, il bundle di comunicazione risponderà sul canale pubblico invitando il mittente ad avviare una sessione di chat privata.

L'assenza di una identificazione univoca degli utenti causa problemi di collisione di nomi: l'uso del nickname non è sufficiente per identificare l'amministratore e in questo caso la gestione dei contatti non è praticabile. I messaggi ricevuti dal metodo *onPrivateMessage* verranno consegnati direttamente al controller. L'interprete potrà eseguire i comandi richiesti dall'amministratore solo dopo che questo ha eseguito con successo il login.

I metodi *kicK*, *disconnecT* e *joinChanneL* dell'interfaccia *IComService* richiamano gli omonimi metodi già presenti nella libreria. Il metodo *changeServeR* esegue prima la disconnessione dal server, poi apre una nuova connessione con il server passato come parametro.

Sono stati creati anche due metodi privati:

- `private void` **`connectTO`**`(String server, int port, String password)` integra la procedura di connessione al server specificato come parametro.
- `private void` **`channelJoin`**`()` legge il file *IrcChannelList.txt*. Ogni riga del file contiene il nome del canale (senza il carattere '#'), pertanto è sufficiente invocare il metodo *joinChanneL* per ogni riga letta.

2.4 Servizio di comunicazione con .NET Messenger Service

Il bundle MSNP implementa le principali funzionalità del servizio di messaggistica istantanea sviluppato da Microsoft.

2.4.1 Protocollo MSNP

Microsoft Notification Protocol (MSNP) [11] è il protocollo proprietario di Microsoft che sta alla base del servizio di messaggistica istantanea .NET Messenger Service. Il client che ne fa uso è Windows Live Messenger, disponibile ufficialmente in versione desktop per sistemi operativi Windows e Mac OS X, su alcuni modelli di smartphone e

palmari, e in versione Web Messenger per browser web. Il servizio permette ad utenti collegati utilizzando un account Windows Live ID di comunicare con altri utenti collegati allo stesso modo ad un qualsiasi server del network. Oltre all'invio dei classici messaggi di testo, il protocollo offre anche un servizio di presenza proprietario, che permette di monitorare lo stato dei propri contatti.

La comunicazione attraverso il network è basata sul modello client/server: è necessario innanzitutto aprire una connessione con un Notification Server (NS) che offre il servizio di presenza, per notificare il client sugli aggiornamenti dei propri contatti. Dopo questo passaggio sarà possibile connettersi a uno degli Switchboard Server (SB) disponibili, che offrono il servizio di messaggistica istantanea vera e propria.

Per poter comunicare, entrambi i client devono essere collegati allo stesso Switchboard Server: l'utente che avvia la comunicazione invita implicitamente l'altro utente sul suo stesso SB. Aprire più conversazioni sul client comporta l'apertura di altrettante connessioni con il Switchboard Server. Per conversazioni con più utenti, il Switchboard Server funziona come un proxy, inoltrando i messaggi a tutti gli interessati.

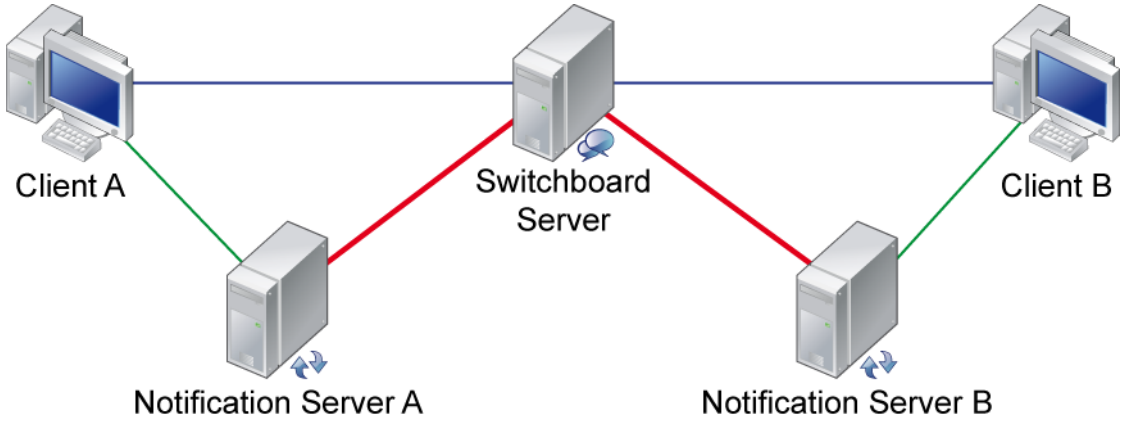


Figura 6: schema dei collegamenti tra client e server in una rete MSNP

2.4.1.1 Comunicazione con i server

La maggior parte della comunicazione con i server è in chiaro, pertanto questo protocollo non è adatto per trasferire password o dati sensibili. A partire dalla nona versione di MSNP il login al Notification Server usa la crittografia del protocollo TLS (in precedenza anche questa parte era in plain text).

A parte il diverso set di comandi disponibili, lo scambio di messaggi tra il client e i server NS e SB mantiene la stessa sintassi. Un esempio di comunicazione tra RoboAdmin e il Notification Server è riportato di seguito:

```
VER 1 MSNP12 CVR0
VER 1 MSNP12
CVR 2 0x0409 winnt 5.1 i386 MSNMSGR 8.1.0178 MSMSGS user@live.it
```

```
CVR 2 14.0.8089 14.0.8089 14.0.8089
http://msgruser.dlservice.microsoft.com/download/ […]
USR 3 TWN I user@live.it
[…]
USR 4 OK user@live.it 1 0
```

I messaggi del client sono indicati in rosso, mentre le risposte del server sono indicate in blu. Per ogni messaggio inviato vengono specificati tre caratteri che identificano il comando, un numero intero crescente che consente di associare le risposte del server (non sempre ordinate) ai messaggi inviati, e infine sono presenti eventuali parametri dipendenti dal comando.

Nell'esempio precedente la prima stringa indica al server la versione del protocollo utilizzata, e con il messaggio *CVR* vengono inviate le informazioni sul client in uso e sulla piattaforma. In questo caso la risposta del server indica che è presente una versione più aggiornata del client (in particolare *14.0.8089*), e fornisce un link per farne il download. I comandi *USR* vengono usati per fare il login: dopo aver specificato il nome utente, il server risponde con le coordinate per aprire una connessione sicura, su cui si può inviare la password.

2.4.1.2 Liste contatti

Ciascun utente del servizio viene identificato univocamente dall'indirizzo e-mail usato in fase di registrazione. Per ogni utente il server mantiene più liste contatti, con significati diversi l'una dall'altra. Subito dopo il login il client invia il comando *SYN*, che gli permette di scaricare le informazioni sulle liste contatti dell'utente collegato.

La *Forward List* (FL) è la lista che il client ufficiale mostra all'utente. I cambiamenti di stato dei contatti in FL verranno aggiornati in tempo reale, in base alle notifiche ricevute dal Notification Server.

La *Reverse List* (RL) è la lista dei contatti che hanno l'utente correntemente collegato nella loro *Forward List*, e quindi devono ricevere i suoi cambiamenti di stato.

La *Allow List* (AL) contiene i contatti autorizzati esplicitamente dall'utente a iniziare una conversazione con esso e a riceverne i cambiamenti di stato. Un contatto rimosso dalla *Forward List* può comunque far parte della *Allow List* e comunicare con l'utente.

La *Block List* (BL) contiene i contatti bloccati esplicitamente dall'utente. Per questi contatti l'utente risulterà sempre offline, e se tentano di invitarlo a una sessione di chat verrà restituito loro un messaggio di errore.

2.4.2 Libreria Java Messenger Library

La libreria Java Messenger Library (JML) [12] è stata usata per implementare il bundle MSNP di RoboAdmin. La libreria si presenta come un insieme di API Java che permettono di interfacciarsi con il .NET Messenger Service. I protocolli supportati vanno da MSNP8 a MSNP12: sono presenti tutte le funzionalità di messaggistica istantanea e una completa gestione della lista contatti.

2.4.2.1 Interfaccia MsnMessenger

Componente principale della libreria è l'interfaccia *MsnMessenger*, presente nel package *net.sf.jml*, implementata dalle classi *SimpleMessenger* e *BasicMessenger*. I metodi principali di *MsnMessenger* usati da RoboAdmin sono i seguenti:

- `public void` **`login`**`()` apre la connessione con il Notification Server e avvia la procedura di login. Il login è necessario per poter eseguire qualsiasi altra operazione.
- `public void` **`logout`**`()` chiude la connessione con il NS e si disconnette da tutte le sessioni aperte con i Server Switchboard.
- `public void` **`setSupportedProtocol`**`(MsnProtocol[] supportedProtocol)` dichiara la compatibilità con alcuni tra i protocolli supportati dalla libreria. Il parametro è un array di protocolli, ottenuti dall'enumerativo *MsnProtocol*.
- `public void` **`addMessageListener`**`(MsnMessageListener listener)` collega l'istanza di *MsnMessenger* ad un listener per i messaggi di chat in entrata.
- `public void` **`addContactListListener`**`(MsnContactListListener listener)` associa un listener per ricevere i cambiamenti alla lista contatti.
- `public void` **`addSwitchboardListener`**`(MsnSwitchboardListener listener)` associa un listener per ricevere le notifiche se nuovi utenti si collegano alla Switchboard.
- `public void` **`addFriend`**`(Email email, String friendlyName)` aggiunge il contatto indicato nel primo parametro alle liste *Forward* ed *Allow*.
- `public void` **`removeFriend`**`(Email email, boolean block)` all'opposto del metodo precedente, rimuove un contatto dalla *Forward List* e, se *block* è impostato a *true*, lo aggiunge alla *Block List*.
- `public void` **`blockFriend`**`(Email email)` e `public void` **`unblockFriend`**`(Email email)` permettono rispettivamente di bloccare un

contatto (anche se non presente in nessuna lista) e di sbloccarlo (se presente nella *Block List*).

Oltre ai metodi di *add* per i listener, sono presenti gli analoghi metodi *remove*. Esistono altri metodi `getOwner`, `getContactList` e `getConnection` per ottenere le rispettive classi ausiliarie. Altre funzioni, ad esempio per l'apertura di una nuova sessione di chat, sono presenti ma non sono state considerate in quanto non deve essere possibile per RoboAdmin iniziare una chat autonomamente.

È possibile ottenere un'istanza di MsnMessenger solo attraverso la classe factory *MsnMessengerFactory*, che offre il metodo statico:

```
public static MsnMessenger createMsnMessenger(String email,
String password)
```

2.4.2.2 Classi listener

Grazie alla libreria è possibile astrarre dai comuni canali di connessione TCP utilizzati per ricevere le notifiche da parte del Notification Server e i messaggi dagli Switchboard Server: ciascun comando in ingresso viene decodificato e viene istanziata una opportuna classe ausiliaria in grado di ospitare i dati di interesse. Implementando in modo opportuno le interfacce del package *net.sf.jml.event* è possibile ricevere le notifiche dai server in modo asincrono. Per comodità, la libreria contiene già delle implementazioni vuote di tutte le interfacce listener: è possibile estendere queste classi, chiamate *adapter*, sovrascrivendo soltanto i metodi che interessano al programmatore.

In questo paragrafo verranno illustrati brevemente soltanto i metodi delle interfacce listener utilizzati all'interno di RoboAdmin.

L'interfaccia *MsnMessageListener* riceve i messaggi, suddividendoli in base al loro contenuto: sono di nostro interesse soltanto i messaggi istantanei e non verranno considerati i messaggi di controllo o di sistema. L'unico metodo utile ai nostri scopi è il seguente:

```
public void instantMessageReceived(MsnSwitchboard switchboard,
MsnInstantMessage message, MsnContact contact)
```

L'interfaccia *MsnContactListListener* permette di ricevere i cambi di stato degli utenti nelle proprie liste. È stata utilizzata nel nostro caso per controllare gli utenti che vogliono aggiungere RoboAdmin alla propria lista contatti e per controllare se i contatti sono autorizzati non appena la lista viene inizializzata. I metodi di interesse per i nostri scopi sono:

```
public void contactListInitCompleted(MsnMessenger messenger)
```

`public void` **`contactAddedMe`**`(MsnMessenger messenger, MsnContact contact)`

Per controllare che le conversazioni siano soltanto con due utenti, si è scelto di monitorare il comportamento della switchboard, che all'interno della libreria è intesa come una sessione di conversazione tra due o più utenti. I requisiti dei bundle impongono che una switchboard contenente più di due utenti venga chiusa. Per fare questo si invoca il metodo *close()* del parametro di tipo *MsnSwitchboard*. I due metodi implementati sono:

`public void` **`switchboardStarted`**`(MsnSwitchboard switchboard)`

`public void` **`contactJoinSwitchboard`**`(MsnSwitchboard switchboard, MsnContact contact)`

2.4.2.3 Classi ausiliarie

Le classi ausiliarie presenti nella libreria JML sono utili per personalizzare il comportamento dell'applicazione. Nel package *net.sf.jml* sono presenti soltanto le interfacce: per ottenere le implementazioni sarà necessario interrogare l'istanza di *MsnMessenger* o riceverle nei listener.

La classe *MsnOwner* contiene i dati dell'utente che sta utilizzando l'applicazione, e permette di vedere e modificare qualsiasi parametro. Di seguito sono riportati alcuni metodi *set*, di cui è ovviamente disponibile anche la versione *get*.

- `public void` **`setDisplayName`**`(String displayName)` permette di impostare il nickname.
- `public void` **`setPersonalMessage`**`(String info)` imposta un messaggio personale.
- `public void` **`setStatus`**`(MsnUserStatus status)` imposta lo stato dell'utente, scegliendo tra quelli forniti dall'enumerativo *MsnUserStatus*.
- `public void` **`setInitStatus`**`(MsnUserStatus status)` imposta lo stato che avrà l'utente subito dopo il login.

MsnContact offre alcuni metodi utili a reperire le informazioni su un utente presente in una lista contatti. I metodi *get* funzionano in maniera analoga a quelli di *MsnOwner*, dal momento che entrambi derivano da *MsnUser*. Mancano del tutto i metodi *set*, in quanto un contatto non è modificabile localmente. L'unico metodo di questa classe utilizzato in RoboAdmin è:

`public boolean` **`isInList`**`(MsnList list)` valuta se il contatto è presente o no nella lista indicata dall'enumerativo *MsnList*.

MsnContactList permette di ottenere gli utenti presenti nelle varie liste del servizio e offre metodi di utilità per reperire un utente in base all'id o all'indirizzo e-mail. I metodi sono i seguenti:

- `public MsnContact` **`getContactByEmail`**`(Email email)` estrae un contatto dalle liste cercandolo in base all'indirizzo e-mail.
- `public MsnContact[]` **`getContactsInList`**`(MsnList list)` restituisce un array con i contatti presenti nella lista passata come parametro.

La classe *MsnSwitchboard* rappresenta una sessione di chat tra due o più utenti. Il nome ricorda lo Switchboard Server, al quale tutti i contatti devono essere collegati per comunicare a vicenda. Sono disponibili tutti i metodi per ottenere i dati dei contatti nella conversazione e per inviare messaggi istantanei. I principali sono:

- `public void` **`sendText`**`(Email email, String text)` invia un messaggio di chat in plain text al contatto specificato dalla classe ausiliaria *Email*. È disponibile anche una funzione `sendMessage` per l'invio di testo con formato.
- `public MsnContact[]` **`getAllContacts`**`()` restituisce un array con i dati di tutti i contatti nella conversazione, compreso l'utente corrente. Se la dimensione dell'array è maggiore di due, RoboAdmin deve abbandonare la conversazione.
- `public void` **`close`**`()` chiude la connessione con la switchboard, uscendo dalla sessione di chat.

2.4.3 Implementazione del bundle MSNP

All'avvio il bundle MSNP legge indirizzo e-mail e la password per l'accesso dal file *MSNP.properties*, e ottiene un'istanza di *MsnMessenger* dalla classe *MsnMessengerFactory*.

Vengono agganciati immediatamente tutti i listener attraverso i metodi di *MsnMessenger*. I listener sono stati ottenuti derivando dalle classi adapter, così è stato possibile sovrascrivere soltanto i metodi di interesse per l'applicazione (elencati nel paragrafo **2.4.2.2**). I metodi sono implementati dalle seguenti classi, presenti all'interno del bundle:

- `class` **`MsnContactListRoboAdminAdapter`** `extends MsnContactListAdapter` aggiorna i contatti presenti nelle contact list non appena questa viene caricata, e controlla se un utente può aggiungere RoboAdmin alla propria lista contatti per vederne i cambiamenti di stato.

- `class` **`MsnMessageRoboAdminAdapter`** `extends MsnMessageAdapter` riceve i messaggi dalla switchboard e, prima di inoltrarli al controller, si accerta che l'utente che li invia sia un amministratore autorizzato e presente nella lista contatti. Se l'utente non è presente nella lista lo aggiunge automaticamente. Se non è autorizzato ed è comunque presente in lista, lo rimuove.
- `class` **`MsnSwitchboardRoboAdminAdapter`** `extends MsnSwitchboardAdapter` controlla che nella switchboard sia presente un solo interlocutore ed esce automaticamente da conversazioni con più utenti.

Prima del login vengono impostati alcuni parametri nell'istanza di MsnMessenger. In particolare, lo stato iniziale viene settato a online, e si impone l'uso di un protocollo successivo alla versione 9, per usare il login con crittografia TLS.

Per quanto riguarda i metodi derivati dall'interfaccia IComService, gli unici utili sono i seguenti:

- Il metodo *sendMessagE* ha come parametri una stringa contenente il messaggio e la switchboard su cui il messaggio deve essere inviato. Non vengono fatti controlli sugli utenti della switchboard prima dell'invio perché la loro attendibilità viene verificata in fase di ricezione, prima che il messaggio passi al controller.
- Con il metodo *kicK* è possibile bloccare un utente malevolo per evitare ulteriori interazioni con RoboAdmin.
- Il metodo *disconnecT* invoca il logout sull'istanza di MsnMessenger per chiudere la connessione con i server.

Sono presenti alcuni metodi privati per gestire situazioni che si verificano di frequente:

- `private void` **`printContactList`**`()` viene invocato per questioni di logging, non appena la lista contatti è stata inizializzata. Mostra tutti i contatti presenti suddividendoli in base alla lista di appartenenza.
- `private String` **`getContactEmailAddress`**`(MsnContact contact)` estrae dalla classe ausiliaria *MsnContact* una stringa contenente l'indirizzo e-mail del contatto, per identificarlo univocamente nel servizio. L'e-mail viene utilizzata spesso per verificare se l'utente è presente nel data base o per salvare l'utente sui log.
- `private void` **`addContact`**`(MsnContact contact)` verifica sul data base (tabella *Accept*) se un utente è autorizzato e lo aggiunge alla *Allow List* (e *Forward List*) o alla *Block List*, in base al risultato della query. Se l'utente era stato bloccato ed è autorizzato, viene automaticamente sbloccato.

- `private void` **`updateContactList`**`()` viene invocato all'avvio dell'applicazione per controllare i contatti presenti nelle liste. Aggiunge i contatti in attesa di essere aggiunti, rimuove e blocca i contatti nella Forward List che non risultano più autorizzati.

2.5 Servizio di comunicazione su rete Skype

Il bundle Skype si basa sul client ufficiale per comunicare in modo testuale con amministratori presenti sulla rete Skype.

2.5.1 Protocollo Skype

Skype [13] è un client e un protocollo proprietario nato per offrire un servizio di telefonate VoIP, e in grado di fornire anche un basilare servizio di messaggistica istantanea. Il protocollo usa un modello di rete peer-to-peer (P2P), invece del più classico modello client/server. Il carico delle telefonate viene gestito dai client ed è distribuito sull'intera rete, che conta ad oggi 521 milioni di utenti registrati.
Il protocollo, che non è stato formalizzato in nessuno standard internazionale, trasmette i dati tra i client in modo cifrato. Il livello di sicurezza dichiarato dall'azienda produttrice, però, non è verificabile a causa della riservatezza dell'algoritmo crittografico.

2.5.1.1 Struttura della rete

L'architettura della rete Skype si basa su tre tipi di elementi: i supernodi, i nodi ordinari e il Login Server.
I supernodi sono dei normali nodi della rete Skype, in grado di fornire alcuni servizi. Sui supernodi risiede parte della Global Index Distributed Directory, una sorta di elenco telefonico che permette agli utenti di trovarsi a vicenda. Grazie alla Global Index Distributed Directory è possibile creare una rete robusta e scalabile, anche se costituita da un gran numero di client non affidabili dal punto di vista della continuità del collegamento. La presenza del supernodo è trasparente all'utente, e può essere notata solo osservando il volume di traffico in transito sull'host.
Ciascun client mantiene una struttura denominata Host Cache: si tratta di una lista contenente un massimo di 200 indirizzi IP (e rispettive porte) dei supernodi. Per essere collegati in rete, tutti i client devono avere almeno una entry valida nella Host Cache, che viene rinnovata regolarmente.

La fase di login è l'unica che prevede l'uso di un server. Dopo la connessione al supernodo, il client si autentica presso un Login Server inviando lo username e la password criptati. Compito dei Login Server è quello di immagazzinare gli username e le password degli utenti registrati. Nell'eseguibile del client è già presente una lista offuscata dei login server utilizzabili.

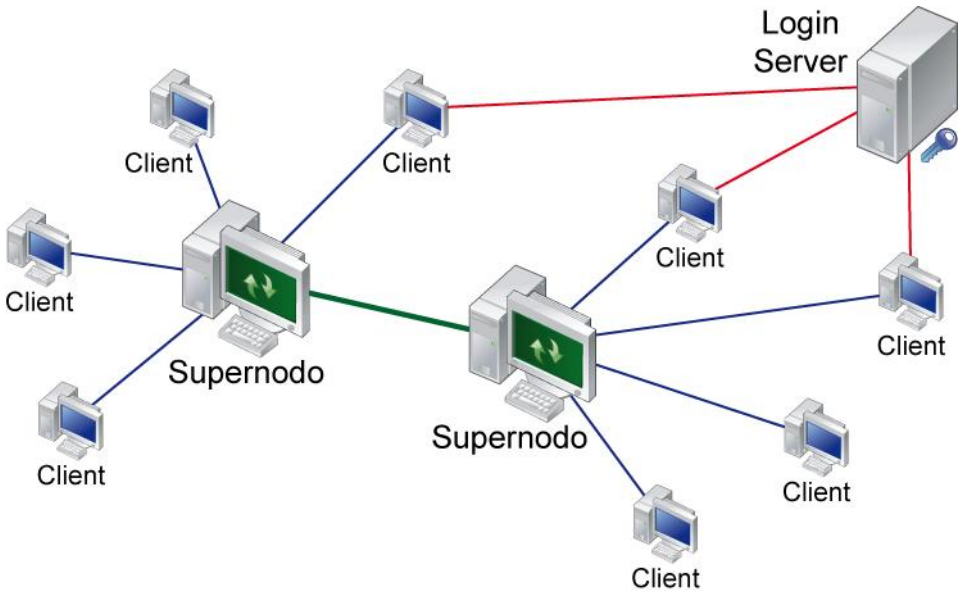


Figura 7: architettura della rete Skype

2.5.1.2 Funzionamento della rete

Tra il client e il rispettivo supernodo sono attive più connessioni: una connessione TCP è usata per inviare i segnali, mentre i dati transitano sia su connessioni TCP che in pacchetti UDP. Le porte utilizzate per la ricezione possono essere configurate attraverso il client.

I supernodi vengono creati automaticamente dal client se la quantità di banda del nodo e le condizioni di NAT, Firewall e Proxy permettono di accettare un certo numero di connessioni in ingresso.

Il protocollo di gestione del traffico prevede che la connessione tra un client e la sua destinazione finale avvenga in modo diretto. Se il destinatario non è in grado di accettare traffico in ingresso, il mittente chiede al Global Index di fare aprire la connessione al destinatario. Se anche il mittente non può ricevere connessioni in ingresso, tutto il traffico dati viene fatto passare attraverso il supernodo, che in questo caso agisce come un router.

2.5.1.3 Sicurezza della comunicazione

Il traffico della rete Skype può essere diretto tra chiamante e chiamato o può passare tra uno o più supernodi. L'intera comunicazione è criptata e decifrabile soltanto dal mittente e dal destinatario, in modo che i supernodi non abbiano accesso al contenuto delle comunicazioni di altri utenti.

All'avvio di una connessione, è necessario essere certi dell'identità dell'interlocutore: per questo i due client si scambiano i loro *Identity Certificate*. Successivamente i client decidono due chiavi (con algoritmo RSA) a 128 bit per crittografare i propri dati. Le chiavi sono uniche per la sessione e non vengono riutilizzate.

Gli *Identity Certificate* sono forniti dal Login Server quando l'utente registra il suo username. Il certificato verrà inviato per ogni login successivo al Login Server, che si comporta da certification authority.

2.5.2 Libreria Skype4Java

Dal momento che Skype usa un protocollo proprietario e un client closed-source, non esiste alcuna libreria open-source che permetta di interfacciarsi con il network. L'uso di una architettura P2P, inoltre, impone che il software sia aggiornato frequentemente, per poter utilizzare sempre l'ultima versione del protocollo. Per risolvere questi problemi non rimane altra scelta che utilizzare il client ufficiale di Skype e permettere a RoboAdmin di interagire con esso grazie ad un plug-in di interfaccia.

Skype4Java [14] è un wrapper scritto in linguaggio Java del Software Development Kit (SDK) fornito dai produttori di Skype. Grazie al SDK è possibile creare il plug-in utile a RoboAdmin per dialogare con l'amministratore attraverso Skype.

Skype4Java si connette al client Skype grazie ad apposite librerie di interfaccia dette *connector*. I connector dipendono sistema operativo su cui il client viene installato e offrono funzioni in linguaggio nativo. Sono disponibili per Windows, Linux e Mac OS X. L'installazione dei connector è indispensabile per poter aprire un canale di comunicazione tra il client e il bundle di RoboAdmin.

2.5.2.1 Protocollo Skype API

La comunicazione tra il wrapper e il client ufficiale consiste in uno scambio bidirezionale di stringhe con un certo formato [15]. Le stringhe iniziano tutte con l'identificatore del comando, seguito da zero o più parametri. La risposta riporta lo stesso identificatore, affiancato dall'eventuale valore restituito. Un esempio di scambio di messaggi in RoboAdmin è il seguente:

```
GET CURRENTUSERHANDLE
CURRENTUSERHANDLE robo.admin
SET PROFILE MOOD_TEXT RoboAdmin
PROFILE MOOD_TEXT RoboAdmin
```

In rosso sono riportati i messaggi diretti al client Skype, in blu le risposte del client a RoboAdmin. Il primo comando è stato usato per ottenere l'Id dell'utente collegato al client. Il secondo comando è stato usato per impostare un messaggio di mood.

Nell'esempio seguente viene mostrata la sintassi dei messaggi di errore:

```
CALL echo123
ERROR 39 user blocked
```

2.5.2.2 Classe Skype

La classe *Skype* del package *com.skype* accorpa tutti i metodi che permettono di eseguire i comandi attraverso il client. I metodi sono tutti statici e hanno il compito di tradurre ciascuna richiesta in un messaggio compatibile con il protocollo di comunicazione del client Skype. Per ogni richiesta al client il metodo attendere la risposta, tradotta a sua volta con il tipo di dato più appropriato. Gli errori ricevuti vengono automaticamente segnalati come eccezioni dell'applicazione.

Per comunicare con il client Skype la classe istanzia automaticamente il singleton *Connector* che, in base al sistema operativo in uso, si aggancia al client e trasmette i comandi con una codifica appropriata.

Il plug-in funziona solo sotto certe condizioni: non solo il client Skype deve essere installato e in esecuzione, ma deve essere collegato alla rete. La libreria non rende disponibili i metodi per fare login o logout.

Sono presenti metodi per associare all'applicazione le classi listener che gestiscono la ricezione asincrona dei messaggi ricevuti. Analogamente sono disponibili i metodi per rimuovere l'associazione con il listener.

Di seguito sono riportati i metodi utili agli scopi di RoboAdmin. I metodi per effettuare chiamate e per avviare sessioni di chat non verranno presi in considerazione. Tutti i metodi possono scatenare l'eccezione *SkypeException*, che deve essere opportunamente catturata e gestita.

- `public static boolean` **`isInstalled`**`()` restituisce *true* se il client è installato. Questo metodo non comunica con il client, ma verifica l'esistenza della directory di installazione di Skype (o di una opportuna chiave nel registro di sistema di Windows).

- `public static boolean` **`isRunning`**`()` controlla se l'applicazione è attualmente in esecuzione e restituisce *true* in caso affermativo.
- `public static synchronized Profile` **`getProfile`**`()` restituisce il profilo dell'utente che sta usando il client.
- `public static ContactList` **`getContactList`**`()` restituisce attraverso la classe ausiliaria *ContactList* gli utenti presenti nella lista contatti.
- `public static void` **`addCallListener`**`(CallListener listener)` associa un listener per le chiamate in entrata.
- `public static void` **`addChatMessageListener`**`(ChatMessageListener listener)` associa un listener per i messaggi di chat in entrata.

2.5.2.3 Classi listener

Come nel caso del bundle MSNP, anche in questo caso le interfacce listener (tutte presenti nel package *com.skype*) sono accompagnate da implementazioni vuote dette adapter. Le notifiche che possono essere ricevute dal client Skype riguardano le telefonate ricevute e i messaggi di chat ricevuti. Sono pertanto presenti le due interfacce *CallListener* e *ChatMessageListener*.

L'interfaccia *CallListener* notifica il plug-in di una chiamata entrante. Il metodo che permette di ricevere le chiamate è:

`void` **`callReceived`**`(Call receivedCall) throws SkypeException.`

L'interfaccia *ChatMessageListener* riceve i messaggi di chat provenienti da un interlocutore remoto. L'unico metodo utilizzato nel nostro caso è il seguente:

`void` **`chatMessageReceived`**`(ChatMessage receivedChatMessage) throws SkypeException.`

2.5.2.4 Classi ausiliarie

Le classi ausiliarie di Skype4Java sono utilizzate dal wrapper per contenere insiemi di operazioni per dati con caratteristiche comuni. Tutte queste classi sono indicate come *final*, quindi non possono essere estese. Questo deriva dal fatto che la libreria è strettamente basata sui servizi offerti dal client. Non viene data la possibilità al programmatore di usare comandi per future versioni del client, senza una nuova versione della libreria. Ogni metodo può lanciare l'eccezione *SkypeException*, che deve essere obbligatoriamente trattata.

La classe *ChatMessage* non è istanziabile direttamente: può essere ricevuta dal listener dei messaggi o dalla classe Skype. Offre i metodi utili alla gestione di messaggi di chat, fornendo il mittente e il contenuto. I metodi utilizzati sono:

- `public User` **`getSender`**`()` restituisce nella classe ausiliaria *User* tutte le informazioni necessarie a identificare il mittente.
- `public String` **`getContent`**`()` restituisce il testo contenuto nel messaggio.
- `public Chat` **`getChat`**`()` riporta un riferimento alla sessione di chat con l'utente corrente, su cui si possono inviare rapidamente le risposte.

La classe *Chat* rappresenta una sessione di chat con uno o più interlocutori. Grazie a questa classe ausiliaria è possibile ottenere tutti i vecchi messaggi e inviare nuovi messaggi. I metodi principali sono quelli che seguono:

- `public User[]` **`getAllMembers`**`()` mostra tutti gli utenti in chat. In base alla dimensione dell'array si riconosce se la conversazione è privata (solo tra due utenti) o pubblica.
- `public void` **`leave`**`()` chiude la sessione di chat. Utile nel nostro caso per uscire da chat con più utenti.
- `public ChatMessage` **`send`**`(String message)` invia il messaggio specificato come parametro a tutti gli utenti in chat e restituisce una copia dell'oggetto trasmesso.

La classe *User* permette di accedere a tutte le proprietà relative ad un utente. È possibile istanziare la classe fornendo al metodo *getInstance* lo Skype Id dell'utente. Grazie a questa classe è possibile inviare messaggi o telefonare all'utente. I metodi utili ai nostri scopi sono:

- `public final String` **`getId`**`()` restituisce lo Skype Id che viene usato per identificare un utente in chat.
- `public final boolean` **`isAuthorized`**`()` controlla se l'utente è presente nella lista contatti e quindi è stato esplicitamente autorizzato.
- `public final void` **`setAuthorized`**`(boolean on)` se *on* vale *true*, l'utente viene aggiunto alla lista contatti. Altrimenti viene rimosso.

La classe *Friend* estende la classe *User*, e viene usata per identificare un utente presente nella lista contatti. Non offre metodi particolarmente utili rispetto a quelli già elencati per *User*.

La classe *Profile* rappresenta l'utente correntemente connesso a Skype. Da qui è possibile cambiare tutte le proprietà del profilo utente, modificarne lo stato e il

messaggio di mood. Di nostro interesse è solo il metodo *getId()*, che restituisce la stringa con lo Skype Id dell'utente.

La classe *ContactList* offre i metodi per esplorare la lista contatti. L'unico metodo utilizzato è:

`public Friend[] getAllFriends()` restituisce tutti gli utenti presenti nella lista contatti.

La classe *Call* gestisce le chiamate in entrata e in uscita. È stata usata per impedire ad un interlocutore di sfruttare le telefonate di Skype. La chat è l'unico mezzo per comunicare con RoboAdmin e le telefonate non sono supportate. L'unico metodo necessario è il seguente:

`public void cancel()` termina una chiamata entrante.

2.5.3 Implementazione del bundle Skype

Nonostante l'implementazione sia piuttosto semplice grazie ai metodi statici della classe Skype, si sono riscontrate alcune limitazioni dovute al sistema a plug-in offerto dal client.

Il principale problema riscontrato riguarda l'impossibilità di avviare e terminare la connessione alla rete Skype. Il client aperto, inoltre, permette a chiunque abbia accesso fisicamente al sistema di utilizzare direttamente il client Skype, per esempio per aggiungere alla lista contatti un utente malintenzionato. Dall'interfaccia grafica di Skype sono visibili chiaramente tutti i messaggi scambiati con l'amministratore, compresa la password di login. Per questo motivo, nonostante la comunicazione sicura tra i due interlocutori, è importante proteggere il sistema evitando l'accesso all'interfaccia grafica del client Skype.

Il bundle legge il parametro *skypeId* dal file *Skype.properties* nella directory *properties* del progetto, per ottenere il nome dell'utente collegato. Qualsiasi metodo del bundle viene eseguito soltanto se l'utente collegato a Skype è quello dichiarato da *skypeId*, così da evitare che RoboAdmin prenda il controllo di qualsiasi account.

Per la ricezione asincrona dei messaggi sono state predisposte due classi che estendono gli adapter forniti dal wrapper Skype4Java. Gli adapter vengono associati alla classe *Skype* tramite i metodi *add* precedentemente descritti.

- `class ChatMessageRoboAdminAdapter extends ChatMessageAdapter` implementa il metodo *chatMessageReceived*, utile per gestire la ricezione dei messaggi di chat. Per ogni messaggio si verifica che il mittente sia autorizzato a comunicare con RoboAdmin e il numero di interlocutori in chat: se le verifiche

hanno successo il messaggio viene inoltrato al controller. Se il mittente è autorizzato ma non è presente nella lista contatti viene immediatamente aggiunto, così che possa vedere il profilo completo di RoboAdmin.

- `class` **`CallRoboAdminAdapter`** `extends CallAdapter` grazie al metodo *callReceived* chiude immediatamente tutte le chiamate in ingresso.

Le implementazioni dei metodi di *IComService* più interessanti sono:

- Il metodo *sendMessagE* invia il messaggio nel secondo parametro sul canale di tipo *Chat* indicato come primo parametro.
- Il metodo *kicK* elimina un utente dalla lista contatti, in modo che non possa più vedere lo stato di RoboAdmin o comunicare con esso.
- Il metodo *disconnecT* non può disconnettere il client dalla rete, ma disabilita le operazioni del bundle, in modo che i messaggi in ingresso non vengano più considerati ed elaborati.

Sono presenti alcuni metodi privati:

- `private boolean` **`canUseSkype`**`()` è il metodo che controlla se è possibile elaborare o meno le informazioni ricevute dal client. Viene controllato se il client è installato e in esecuzione. Tramite la funzione statica *Skype.getProfile.getId()* si controlla che lo Skype Id dell'utente sia quello specificato nel file properties. Viene restituito false se uno qualsiasi di questi controlli fallisce o se è stato invocato il metodo *disconnecT*.
- `private void` **`printContactList`**`()` viene invocato per scopi di logging e mostra tutte le entry della lista contatti.
- `private void` **`updateContactList`**`()` invocato all'avvio dell'applicazione elimina dalla lista contatti tutti gli utenti non autorizzati, in base al contenuto del data base.

2.5.4 Istruzioni di installazione

Il client si trova sul sito web www.skype.com: in base al sistema operativo si dovrà scaricare e installare la versione del software più appropriata.

Dal momento che i *connector* distribuiti assieme alla libreria Skype4Java sono in linguaggio nativo, potrebbero essere necessarie alcune operazioni aggiuntive per rendere funzionante il bundle.

Il wrapper Java è compatibile con i sistemi operativi Windows, Linux e Mac OS X, e a seconda del sistema operativo in uso tenta di raggiungere le librerie necessarie. Verranno ora illustrate le procedure di installazione per Windows e Linux.

2.5.4.1 Installazione del bundle su Windows

All'interno della cartella *lib\skype* del progetto è presente il file *swt-win32-3232.dll*, che deve essere inserito in una delle cartelle del path di Windows per poter essere riconosciuto. Il path è configurabile tra le variabili d'ambiente del sistema in uso.
La libreria *skype.dll*, necessaria al funzionamento del connector, è presente nel JAR di Skype4Java e viene estratta automaticamente in una cartella temporanea prima di essere caricata.

2.5.4.2 Installazione del bundle su Linux

Nel caso di una installazione su Linux il wrapper Skype4Java provvede automaticamente ad estrarre i due file *libskype.jnilib* e *libskype.so* nella directory */tmp*. I file sono già presenti all'interno del file JAR contenente la libreria. In caso di problemi è possibile estrarre manualmente i due file dal pacchetto JAR in una delle directory specificate nella variabile d'ambiente $PATH.

2.6 Servizio di comunicazione con protocollo XMPP

Il bundle XMPP implementa la maggior parte delle funzionalità rese disponibili dal protocollo aperto XMPP, usato tra gli altri dal client Google Talk.

2.6.1 Protocollo XMPP

Extensible Messaging and Presence Protocol (XMPP) [16], precedentemente noto come Jabber, è un protocollo aperto e standardizzato basato sullo scambio di dati XML. Oltre ad offrire un servizio di messaggistica istantanea completo, permette a ciascun utente di ricevere informazioni di presenza, relative allo stato dei propri amici.
La comunicazione tra utenti avviene in modalità client/server: ciascun gestore può usare il proprio server XMPP per creare nuove reti di contatti. La comunicazione tra un server e l'altro è stata resa disponibile dopo la prima standardizzazione del protocollo e non sempre è supportata.
La connessione dell'utente alla rete avviene utilizzando un Jabber Id, contenente l'indirizzo e-mail usato in fase di registrazione, il nome DNS del server a cui connettersi (noto spesso come dominio) e un identificativo di risorsa (ad esempio *nome.utente*

@dominio.com/risorsa). Il significato del parametro risorsa dipende dal server. In alcuni casi viene usato per identificare il client: se l'utente è collegato con più client alla rete, l'utente remoto può specificare a quale client vuole inviare i messaggi. In altri casi può indicare il nickname dell'utente in una chat room.

Per consentire la connessione anche ad utenti protetti da firewall, è prevista una versione del protocollo funzionante attraverso HTTP o HTTPS, in modo pull (ormai deprecato) o push.

L'invio di un messaggio istantaneo è simile all'invio di una e-mail: il client del mittente spedisce il messaggio al proprio server; il server del mittente contatta il server del destinatario; infine il server del destinatario trasmette il messaggio al client.

Importante caratteristica del protocollo è la sua estendibilità, che ha permesso di adeguarlo ai servizi che si sono affermati col passare del tempo, come il VoIP e il file sharing.

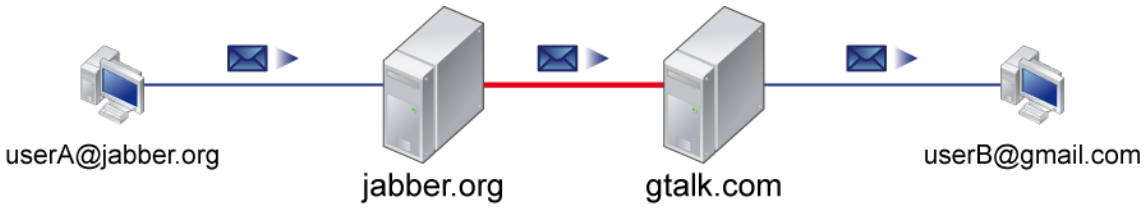


Figura 8: trasferimento di un messaggio da un server all'altro per raggiungere il client

2.6.1.1 Stream XMPP

La comunicazione con il server avviene grazie ad una coppia di documenti XML. L'intero scambio di informazioni con il server è racchiuso tra i tag <*stream*> e </*stream*>. Gli attributi del tag vengono usati per scambiare informazioni sulla versione del protocollo e sulla lingua del client e per identificare il mittente e il destinatario. Ogni stream XML è inoltre identificato da un attributo *id*, impostato dal ricevente. Una fase di negoziazione iniziale porta all'avvio di una connessione TLS con il server, che permette l'invio e la ricezione di messaggi crittografati.

Subito dopo aver stabilito la connessione, il client invierà gli elementi XML al server. Per inviare le risposte, il server negozierà un nuovo flusso XML con il client.

La principale debolezza del protocollo risiede nel grosso overhead dovuto allo scambio di informazioni di presenza ai server e ai contatti che le richiedono: solo queste occupano il 60% dello stream, e spesso risultano ridondanti.

2.6.1.2 Connessione ad altri servizi

Una importante caratteristica di XMPP è quella che abilita un utente a connettersi a reti basate su altri protocolli, come MSNP, OSCAR, SMS, e-mail. Questo avviene grazie ad un server apposito detto gateway, che esegue le operazioni necessarie per connettersi ai server in uso dall'altro protocollo e tradurre i messaggi istantanei e di presenza XMPP in pacchetti comprensibili al protocollo richiesto. A differenza di un client multi protocollo, XMPP offre questo servizio lato server: il gateway presente su un host remoto viene raggiunto direttamente dal server, senza oneri di traduzione lato client. L'uso del gateway prevede che l'utente fornisca le credenziali di accesso all'altro servizio che intende utilizzare, così da essere riconosciuto come utente dell'altro servizio di comunicazione.

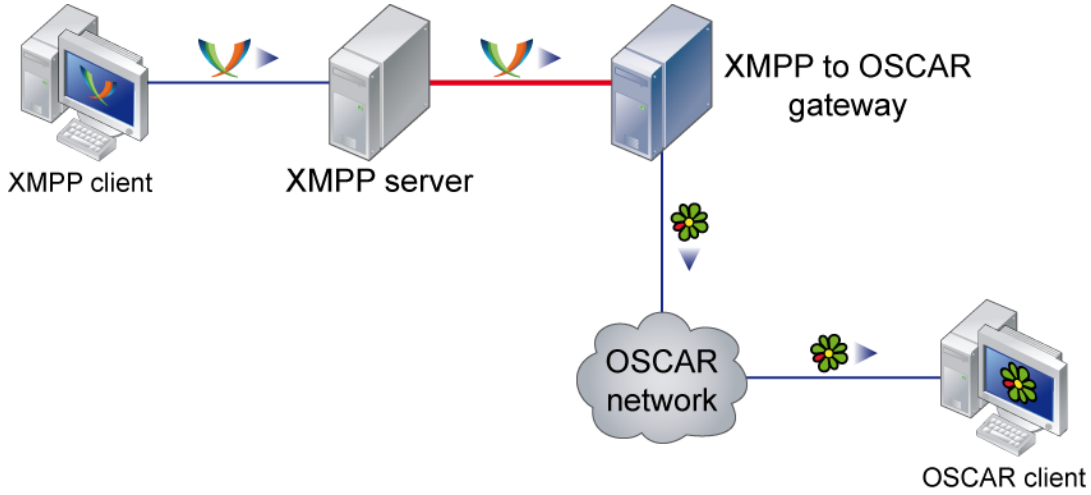


Figura 9: esempio di gateway tra reti XMPP e OSCAR

2.6.2 Libreria Smack

La libreria Smack [17], scritta interamente in Java, offre un insieme di API per interfacciarsi con una qualsiasi rete XMPP o Jabber. La vasta community che porta avanti lo sviluppo di questa libreria open source, rende disponibili anche molte estensioni che, se supportate dal server, permettono al client di eseguire chiamate VoIP, avviare chat multi utente, trasferire file, e molto altro. Per poter usare la libreria è necessario includere nel progetto due archivi JAR:

- *smack.jar* offre le principali funzionalità per connettersi e comunicare con il server. Viene reso disponibile il package *org.jivesoftware.smack,* contenente la maggior parte delle classi usate da RoboAdmin.
- *smackx.jar* contiene le estensioni più comuni del protocollo XMPP. Anche se non si usano estensioni è necessario includere questa libreria, altrimenti il progetto non può essere compilato.

2.6.2.1 Classe XMPPConnection

La classe *XMPPConnection* gestisce una connessione con il server XMPP specificato dall'utente e traduce automaticamente i dati XML ricevuti assegnandoli a classi ausiliarie facilmente interpretabili. La connessione può essere aperta e chiusa più volte e, se la connessione si interrompe, verrà riaperta automaticamente.

A questa classe possono essere associati alcuni listener, per gestire lo stato della connessione e il flusso dei pacchetti in entrata e uscita. In caso di errore alcuni metodi possono scatenare una eccezione *XMPPException*, che restituisce anche il codice d'errore definito nelle specifiche del protocollo. Se si esegue la disconnessione, tutti i listener vengono mantenuti, a meno che non si usino gli appositi metodi per rimuoverli.

Il nome del server e i parametri di connessione sono stati raggruppati nella classe *ConnectionConfiguration*, passata a *XMPPConnection* attraverso il costruttore.

I metodi utilizzati in RoboAdmin sono i seguenti:

- `public void` **`connect`**`()` apre una connessione con il server, creando le socket necessarie. Se in una precedente connessione era già stato fatto un login, questo verrà rieseguito automaticamente se si invoca il metodo *connect*.
- `public void` **`disconnect`**`()` imposta la presenza a *offline* e chiude la connessione con il server. Il metodo rilascia le risorse occupate dalla connessione, impedendo l'uso del roster e dei listener associati ad essa.
- `public void` **`login`**`(String username, String password, String resource)` esegue il login sul server e rende l'utente disponibile ai propri contatti. La libreria usa automaticamente il metodo di autenticazione più sicuro proposto dal server.
- `public void` **`sendPacket`**`(Packet packet)` invia un pacchetto al server attraverso una connessione già aperta. Nel nostro caso è stato usato per l'invio dei pacchetti presenza XMPP.
- `public void` **`addPacketListener`**`(PacketListener packetListener, PacketFilter packetFilter)` associa un listener dei pacchetti alla connessione, in modo che i pacchetti possano essere intercettati ed elaborati. Come secondo parametro è possibile specificare un filtro, per fare in modo che solo alcuni pacchetti vengano ricevuti dal listener specificato. Se il secondo parametro è *null*, nessun pacchetto verrà filtrato.
- `public Roster` **`getRoster`**`()` restituisce il roster dell'utente se la connessione lo permette, altrimenti restituisce *null*.

- `public ChatManager` **`getChatManager`**`()` restituisce un'istanza del *ChatManager*, usato per gestire tutte le sessioni di chat della connessione corrente.

2.6.2.2 Classe Roster

Il roster rappresenta la lista contatti dell'utente di un servizio XMPP. L'utente riceverà gli aggiornamenti di stato solo per i contatti presenti nel roster. Un contatto che non è incluso nel roster (o che è stato rimosso dal roster) non può contattare l'utente o vedere il suo stato: questo stratagemma è stato usato per impedire ad utenti non autorizzati di contattare RoboAdmin.

Di seguito verranno descritti brevemente i principali metodi della classe:

- `public void` **`setSubscriptionMode`**`(Roster.SubscriptionMode subscriptionMode)` imposta in che modo deve comportarsi l'applicazione quando riceve richieste di sottoscrizione dagli altri utenti. L'enumerativo *Roster.SubscriptionMode* offre tre modalità: con *accept_all* e *reject_all* la gestione dei pacchetti di sottoscrizione è automatica. Nel nostro caso è importante controllare gli utenti che vengono aggiunti alla lista contatti, quindi la modalità di sottoscrizione è *manual*.
- `public void` **`createEntry`**`(String user, String name, String[] groups)` inserisce un nuovo utente nel roster, inviando automaticamente al server i pacchetti di sottoscrizione.
- `public void` **`removeEntry`**`(RosterEntry entry)` rimuove un utente dal roster e attende che il server sia disponibile per inviare i pacchetti di sottoscrizione necessari.
- `public Collection<RosterEntry>` **`getEntries`**`()` restituisce una collezione di *RosterEntry* con tutti i contatti presenti nel roster.

2.6.2.3 Classi listener

Per elaborare i messaggi e le notifiche in ingresso sono state implementate alcune delle interfacce listener presenti nella libreria Smack. È di nostro interesse controllare l'attendibilità degli utenti nel roster e ricevere messaggi istantanei da chat non pubbliche.

L'interfaccia *PacketListener* riceve tutti i pacchetti provenienti dal server. I pacchetti possono essere filtrati dalla classe *XMPPConnector*, se richiesto dal programmatore. Nel nostro caso l'implementazione di *PacketListener* viene usata per ricevere i pacchetti riguardanti le sottoscrizioni al roster: si riceveranno le istanze della classe *Presence*, che

(se il pacchetto è di tipo *subscribe*) contengono il Jabber Id degli utenti che vogliono sottoscriversi. L'unico metodo presente nell'interfaccia è:

```
void processPacket(Packet packet)
```

L'interfaccia *ChatManagerListener*, se associata ad un'istanza della classe *ChatManager* (ottenuta da una *XMPPConnection*), riceve le notifiche quando vengono avviate nuove sessioni di chat. L'unico metodo presente permette di controllare i parametri della chat e verificare che sia stata aperta in remoto (se il secondo parametro ha valore *false*):

```
void chatCreated(Chat chat, boolean createdLocally)
```

L'interfaccia *MessageListener* può essere associata ad una istanza della classe chat, per processare i messaggi in entrata. Quando la chat viene creata è buona norma associare immediatamente un *MessageListener* per evitare di perdere dei messaggi. Il metodo seguente permette di vedere il contenuto del messaggio. Nel caso in cui l'utente che ha creato la chat non sia affidabile non è stato associato alcun *MessageListener*: il primo parametro del metodo non viene controllato, perché rappresenta una conversazione con un utente già verificato in precedenza.

```
void processMessage(Chat chat, Message message)
```

2.6.2.4 Classi ausiliarie

La libreria offre parecchie classi ausiliarie nel package *org.jivesoftware.smack*. La maggior parte di queste classi può essere ottenuta usando i metodi *get* delle classi descritte poco fa, oppure può essere ricevuta dai metodi dei listener. Istanziare queste classi è possibile, ma per i nostri scopi non è sempre necessario.

La classe *ConnectionConfiguration* può essere passata come parametro al costruttore della classe *XMPPConnection*, per specificare a quale server ci si vuole connettere. Sono previsti diversi costruttori per *ConnectionConfiguration*: nel nostro caso è stato sufficiente passare come parametro una stringa contenente il nome DNS del server e un intero indicante la porta remota con cui aprire la connessione TCP. Sono stati usati degli altri metodi per obbligare la libreria ad usare una connessione sicura con il server:

- `public void setSecurityMode(ConnectionConfiguration.SecurityMode securityMode)` è stato usato per imporre l'uso della crittografia TLS durante il collegamento con il server. Il parametro può assumere tre valori: *disabled* non usa la crittografia TLS, *enabled* la usa se disponibile, *required* (usato in RoboAdmin) obbliga all'uso di TLS.

- `public void` **`setVerifyChainEnabled`**`(boolean verifyChainEnabled)` controlla l'intera catena dei certificati, se il valore del parametro è true. Il valore di default è false.

La classe *RosterEntry* raccoglie i dati di un utente presente nel roster. Per ogni entry è possibile leggere lo stato e il nome dell'utente. RoboAdmin usa questa classe per leggere l'id e verificare dal data base che si tratti di un utente autorizzato. L'unico metodo necessario è:

`public String` **`getUser`**`()` restituisce il Jabber Id dell'utente selezionato dal roster.

La classe *ChatManager* ha la responsabilità di tenere traccia delle sessioni di chat correntemente aperte. È anche in grado di notificare l'apertura di nuove sessioni di chat, se viene associato l'apposito listener. Il metodo che fa questo è il seguente:

`public void` **`addChatListener`**`(ChatManagerListener listener)`

La classe *Chat* rappresenta una sessione di chat di cui l'utente è un partecipante. È possibile associare un listener per ricevere i messaggi relativi alla chat e controllare che l'interlocutore sia un utente autorizzato. Avendo l'istanza di chat è inoltre possibile inviare un messaggio ad un utente. I metodi usati in RoboAdmin sono:

- `public void` **`addMessageListener`**`(MessageListener listener)` associa un listener per i messaggi in entrata relativi alla chat.
- `public String` **`getParticipant`**`()` restituisce il Jabber Id del partecipante alla chat, in modo da verificare se l'utente è autorizzato o meno.
- `public void` **`sendMessage`**`(String text)` invia un messaggio di testo sulla chat corrente. Esiste una versione che accetta un parametro *Message*, per inviare messaggi di diverso tipo.

La classe *Presence* rappresenta un pacchetto presenza XMPP. Nel nostro caso è necessario controllare quando un utente vuole essere aggiunto al roster di RoboAdmin. Questo può essere fatto osservando i pacchetti presenza di tipo *subscribe*: quando si riceve un pacchetto presenza è possibile controllare il mittente. Se è attendibile verrà immediatamente aggiunto al roster e la sua richiesta verrà confermata inviando un pacchetto *Presence* di risposta con tipo *subscribed*. Se il mittente non è attendibile la sua richiesta verrà rifiutata inviando un pacchetto *Presence* di risposta con tipo *unsubscribed*. Il tipo del pacchetto presenza viene impostato nel costruttore tramite il parametro *Presence.Type*, e può essere letto nei pacchetti ricevuti tramite il metodo *getType()*. Altri metodi usati sono:

- `public void` **`setTo`**`(String to)` imposta il Jabber Id del destinatario del pacchetto presenza, nei pacchetti che devono essere inviati.
- `public String` **`getFrom`**`()` legge il Jabber Id del mittente del pacchetto presenza, nei pacchetti ricevuti.

La classe *Message* rappresenta un pacchetto XMPP contenente un messaggio. Nel nostro caso gli unici messaggi di interesse sono quelli di chat, che vengono ricevuti dall'amministratore all'interno di una sessione di chat già verificata. L'unico metodo interessante per RoboAdmin è:

`public String` **`getBody`**`()` restituisce il testo del messaggio come stringa.

2.6.3 Implementazione del bundle XMPP

Quando il bundle viene attivato vengono letti i parametri specificati nel file *XMPP.properties*: oltre all'indirizzo e-mail e la password sono configurabili anche il nome host, la porta e il dominio XMPP di riferimento. Prima di aprire la connessione viene imposto l'uso della crittografia TLS.

Vengono immediatamente associati i listener alla connessione, al roster e al chat manager. I listener sono stati ottenuti implementando le opportune interfacce. In particolare si sono ricavate tre classi:

- `class` **`PacketRoboAdminAdapter`** `implements PacketListener` nel metodo *processPacket* riceve i pacchetti presenza di tipo *subscribe*. In base al mittente del pacchetto decide se aggiungere il mittente al roster oppure no.
- `class` **`ChatManagerRoboAdminAdapter`** `implements ChatManagerListener` quando una chat viene creata (metodo *chatCreated*) controlla l'attendibilità del partecipante e, se attendibile, associa il *MessageListener*. Se l'utente partecipante non è presente nel roster, lo aggiunge automaticamente.
- `class` **`MessageRoboAdminAdapter`** `implements MessageListener` passa al Controller i messaggi ricevuti dalle sessioni di chat già verificate.

I metodi implementati dall'interfaccia IComService sono i seguenti:

- Il metodo *sendMessagE* invia il messaggio presente nel secondo parametro sfruttando l'omonimo metodo della classe *Chat* passata come primo parametro.
- Il metodo *kicK* rimuove dal roster l'utente con il Jabber Id specificato dal parametro *nick*.
- Il metodo *disconnecT* invoca l'omonimo metodo sull'istanza della classe *XMPPConnection*, per chiudere la connessione e rilasciare le risorse.

Sono presenti quattro metodi privati:

- `private void` **`addEntry`**`(String address, String name)` controlla nel data base se l'utente passato come primo parametro è un amministratore. Se è autorizzato a comunicare viene aggiunto al roster e gli viene inviato un pacchetto presenza di tipo *subscribed* come conferma. In caso contrario la sua richiesta viene rifiutata inviando un pacchetto presenza di tipo *unsubscribed*. Se l'utente non è autorizzato ma è presente nel roster, viene immediatamente rimosso.
- `private String` **`addressOf`**`(String xmppAddress)` viene usato per estrarre l'indirizzo e-mail dal Jabber Id dell'utente che partecipa alla chat, rimuovendo la parte relativa alla risorsa.
- `private void` **`printRoster`**`()` usato per scopi di logging mostra tutti gli utenti presenti nel roster.
- `private void` **`updateRoster`**`()` rimuove dal roster gli amministratori che non sono più presenti nel data base.

2.7 Servizio di comunicazione con protocollo OSCAR

Il protocollo OSCAR è stato usato per permettere a RoboAdmin di comunicare con utenti che usano i client ICQ, AIM e iChat.

2.7.1 Protocollo OSCAR

OSCAR (Open System for Communication in Realtime) [18] è un protocollo proprietario di AOL che consente lo scambio di messaggi istantanei e di informazioni di presenza tra gli utenti presenti in una "buddy list".

I client che lo supportano ufficialmente sono ICQ (o *I Seek You*) e AIM (*Aol Instant Messenger*). Nel 2002 AOL ha firmato un contratto con Apple per permettere l'uso del protocollo OSCAR anche nel client iChat, preinstallato sui sistemi operativi Mac OS. Questo ha permesso agli utenti di iChat di comunicare con utenti che utilizzano gli altri due client.

ICQ è il client più anziano tra i tre precedentemente accennati. In questo caso, all'atto della registrazione, l'utente specifica soltanto la propria password, e il server gli fornisce un numero identificativo univoco detto Universal Internet Number (UIN), che rappresenta il proprio nome utente. Per gli altri client la procedura di registrazione è più moderna, e consente di decidere sia il nome utente (detto *screenname*) che la password per l'accesso.

2.7.1.1 Funzionamento del protocollo

I servizi offerti dal protocollo sono distribuiti su più server. Inizialmente il client esegue il login su un server noto: se l'autenticazione ha successo, il messaggio di risposta conterrà il numero IP del server BOSS (Basic OSCAR Service Server). Il server BOSS fornisce il servizio di presenza, l'instradamento dei messaggi istantanei e altri servizi generali. La chiusura della connessione con il server BOSS provoca l'uscita dal servizio. Il client può fare uso di un altro server, detto BART (Buddy Art), che consente di fare il download dell'avatar, dei suoni e degli sfondi scelti dall'utente.

OSCAR è un protocollo di tipo binario: la catasta dei protocolli prevede l'uso di TCP a livello di trasporto, sovrastato da un livello FLAP (Frame Layer Protocol) usato per il framing e un livello SNAC, che contiene il comando OSCAR vero e proprio. Nelle ultime versioni del protocollo, tra i livelli TCP e FLAP può essere inserito un livello TLS o SSL (se il client lo supporta), che garantisce la riservatezza dell'intera comunicazione con i server.

Lo header aggiunto nel livello FLAP contiene il canale su cui viene inviato il pacchetto, il numero di sequenza e la dimensione della parte dati. I canali disponibili sono tre: il canale 1 è riservato ai pacchetti di connessione, sul canale 2 si trasmettono i dati, mentre il canale 3 contiene gli errori. Lo header di SNAC può essere usato per identificare il servizio che il pacchetto sta richiedendo al server o il messaggio che deve essere spedito attraverso i server.

L'invio di un messaggio istantaneo può avvenire in tre modi diversi:

- Modo *DIM* (Direct Instant Messaging). Nella maggior parte dei casi, quando il server riceve una richiesta di avvio di una sessione di chat, tenta di avviare una connessione diretta tra i client coinvolti per lo scambio di messaggi in peer to peer.
- Modo *standard*. Se la connessione tra i client non può essere instaurata (ad esempio per problemi di firewall o NAT) il client userà il server BOSS come router, e tutti i messaggi diretti al client destinatario passeranno attraverso il server.
- Modo *chat*. Il protocollo prevede la presenza di chat pubbliche risiedenti su un server: in questo caso il client invia il messaggio istantaneo al proprio server BOSS, e questo si occuperà di instradarlo al server che ospita la chat.

2.7.1.2 Buddy List

La Buddy List è l'elenco contatti che il client mostra all'utente una volta stabilita la connessione. Il protocollo permette al client una gestione diretta e completa della Buddy

List, con il rischio che questa venga corrotta definitivamente. Il server esegue comunque alcuni controlli per evitare gli errori più grossolani.

Gli utenti, detti Buddy, vengono suddivisi in gruppi dall'utente. Per essere considerata valida, la Buddy List deve contenere almeno un gruppo: subito dopo la registrazione viene reso disponibile ad ogni utente il gruppo predefinito *Buddies*.

È compito del client decidere in che modo viene gestita la Buddy List: tipicamente l'inserimento di un Buddy avviene solo dopo la conferma dell'utente (per questioni di privacy), ma è previsto dal protocollo anche un inserimento automatico senza conferma di tutti gli utenti che ne fanno richiesta.

2.7.2 Libreria AccSDK

Fino a pochi anni fa l'uso del protocollo OSCAR da parte di client non autorizzati da AOL non era permesso. La comunità Open Source aveva comunque a disposizione alcune librerie che consentivano l'accesso al servizio, ottenute tramite reverse engineering.

Dal 2006 la stessa AOL ha fornito ai programmatori interessati al protocollo un insieme di librerie che permettono la creazione di client personalizzati. La licenza d'uso delle librerie definisce un codice comportamentale che il programmatore deve obbligatoriamente seguire per non rovinare l'intero servizio.

La libreria fornita da AOL e usata per il bundle di RoboAdmin prende il nome di AccSDK [19]. Permette di integrare le funzionalità del client di messaggistica istantanea AIM in software scritti in un qualsiasi linguaggio supportato dal framework .NET su Windows, in C/C++ su altri sistemi operativi e in Java. Per il funzionamento di AccSDK sono richieste alcune librerie aggiuntive dipendenti dal sistema operativo su cui l'applicazione viene installata: sono supportati i sistemi Windows, Linux e Mac OS X.

La libreria Java è comunque closed source. Sul sito AOL Developer Network è disponibile un Javadoc piuttosto esauriente, ma mancano le descrizioni di alcune funzioni usate in RoboAdmin.

2.7.2.1 AIM Bots

L'uso della libreria AccSDK è vincolato da una licenza d'uso piuttosto restrittiva. Nel nostro caso sono state accettate le condizioni imposte dalla Open Bot Developers License Agreement [20].

In particolare è necessario che il programmatore richieda ad AOL una chiave identificativa dell'applicazione che vuole creare. La chiave verrà inviata al login server assieme al nome utente e alla password in fase di autenticazione. Tramite una form sul sito web di AOL è inoltre possibile registrare lo screenname usato da RoboAdmin come Bot: nonostante non sia necessario al funzionamento dell'applicazione, questa procedura è fortemente consigliata perché offre alcune garanzie aggiuntive, prima tra tutte la fluidità nell'invio dei messaggi da parte di RoboAdmin. Un Bot può ricevere una grande quantità di messaggi da molti utenti del servizio, ma non può iniziare autonomamente conversazioni con utenti che non hanno il Bot nella propria Buddy List. Una restrizione importante da tenere in considerazione è il fatto che non possano essere inviati più di 10.000 messaggi al giorno e 150.000 messaggi al mese. Se viene superato questo limite i nuovi messaggi inviati non verranno consegnati al destinatario. Ulteriori restrizioni sono imposte per quanto riguarda il contenuto dei messaggi inviati:
Alcune limitazioni imposte dai termini d'uso possono essere superate se si concorda con AOL un account a pagamento Enterprise Bot. In questo caso è possibile ricevere anche il numero IP dell'interlocutore, accedere a dati riservati degli utenti con cui si comunica e rendere illimitato il numero di messaggi che possono essere scambiati con gli utenti.

2.7.2.2 Classe AccSession

La classe *AccSession* presente nel package *com.aol.acc* gestisce la connessione con i server richiesti dal protocollo e viene associata alla classe listener per essere notificata degli eventi che riguardano l'utente. Una volta eseguita l'autenticazione è possibile usare i metodi di *AccSession* per inviare nuovi messaggi di chat e per ottenere informazioni sull'identità dell'utente e sulla connessione.
La ricezione degli eventi non è sincrona come nelle librerie già descritte in precedenza, ma è necessario estrarre periodicamente i messaggi da una coda FIFO gestita direttamente da AccSession.
Prima di avviare la connessione è necessario indicare la chiave che identifica il programma (registrata tramite una form presente sul sito web di AOL) e impostare alcune preferenze.
I metodi utilizzati in RoboAdmin sono descritti di seguito. La maggior parte di essi può lanciare una eccezione di tipo *AccException*, nel caso si verifichino errori di protocollo. Da *AccException* è possibile riconoscere il codice dell'errore verificatosi grazie all'enumerativo *AccResult*, restituito dalla variabile pubblica *errorCode*.

- `public void` **`setPrefsHook`**`(AccPreferencesHook piAccPrefsHook)` attraverso il parametro di tipo *AccPreferencesHook* consente di modificare le preferenze della connessione.
- `public void` **`setIdentity`**`(String identity)` deve essere usato per inserire lo screenname necessario al login.
- `public void` **`signOn`**`(String password)` si collega al login server per l'autenticazione e apre automaticamente la connessione con il server BOSS indicato nel messaggio di risposta.
- `public void` **`signOff`**`()` chiude la connessione con il server BOSS, rendendo l'utente offline ai propri contatti.
- `public void` **`setEventListener`**`(AccEvents eh)` associa un listener per gli eventi che riguardano l'utente. Il parametro di tipo *AccEvents* contiene tutti gli eventi supportati dalla libreria.
- `public AccBuddyList` **`getBuddyList`**`()` restituisce la Buddy List dell'utente. La Buddy List viene ricevuta dal server BOSS subito dopo che la connessione è stata stabilita.
- `public static void` **`pump`**`(int i)` estrae e processa un certo numero di eventi contenuti nella coda FIFO di *AccSession*.
- `public AccIm` **`createIm`**`(String text, String type)` crea un nuovo messaggio istantaneo, che può essere inviato in una sessione di chat già stabilita. Il parametro *type* indica il MIME Type con cui il messaggio deve essere codificato: nel nostro caso vale *text/plain*.

2.7.2.3 Gestione della Buddy List

Per gestire la visibilità di RoboAdmin agli amministratori che ne fanno uso sono state implementate la maggior parte delle funzionalità di gestione della Buddy List. I Buddy autorizzati vengono aggiunti alla Buddy List solo quando aprono una sessione di chat con RoboAdmin.

La classe *AccBuddyList* può essere ottenuta da *AccSession* dopo la fase di login. Non è possibile manipolare direttamente gli utenti da questa classe, è necessario accedere prima al gruppo in cui i contatti sono salvati. Per fare questo sono disponibili due metodi: *getGroupByIndex* restituisce il gruppo corrispondente all'indice specificato, mentre *getGroupByName* restituisce il gruppo il cui nome è stato passato come parametro.

Il gruppo è rappresentato dalla classe *AccGroup*: è possibile ottenere un utente in base al nome o alla posizione. I metodi usati sono i seguenti:

- `public AccUser` **`getBuddyByIndex`**`(int position)` è utile in un ciclo (da zero al valore restituito da *getBuddyCount*) per ottenere i dati di un utente della Buddy List.
- `public AccUser` **`insertBuddy`**`(AccUser user, int position)` aggiunge un nuovo utente nel gruppo. Se la posizione vale -1 l'utente verrà aggiunto in fondo al gruppo.
- `public void` **`removeBuddy`**`(int position)` rimuove dal gruppo l'utente nella posizione specificata.

La classe *AccUser* permette di accedere alle informazioni su un utente di AIM. È importante poter bloccare un utente se lo si considera non attendibile e sbloccarlo in caso contrario.

- `public String` **`getName`**`()` restituisce l'id dell'utente. Questo può essere lo screenname di AIM o il numero UIN di ICQ.
- `public void` **`setBlocked`**`(boolean blocked)` blocca l'utente e gli impedisce di contattare RoboAdmin se il parametro *block* vale *true*, altrimenti lo sblocca.

2.7.2.3 Interfaccia AccEvents

Implementando l'interfaccia *AccEvents* possono essere intercettati tutti gli eventi inviati dal server. Dal momento che la libreria non separa gli eventi su interfacce diverse, è stato necessario implementare anche i metodi riguardanti eventi non necessari a RoboAdmin. I metodi di interesse per l'applicazione sono i seguenti:

- `void` **`OnStateChange`**`(AccSession session, AccSessionState state, AccResult hr)` permette di controllare lo stato della connessione. Il parametro *state* è un enumerativo utile per distinguere le transizioni tra uno stato e l'altro. Dal parametro *hr* si può vedere il motivo di una disconnessione improvvisa o di un login non riuscito.
- `void` **`OnSecondarySessionStateChange`**`(AccSession session, AccSecondarySession secondarySession, AccSecondarySessionState state, AccResult hr)` deve essere obbligatoriamente implementato: se il valore dell'enumerativo *AccSecondarySessionState* è *ReceivedProposal* significa che l'utente è stato invitato in una nuova sessione di chat. Per poter comunicare con l'interlocutore deve essere

accettata la sessione secondaria usando il metodo *accept* di *secondarySession*. Se il parametro *secondarySession* non è istanza di *AccImSession* o se l'utente che ha inviato la richiesta non è autorizzato, la sessione può essere rifiutata con il metodo *reject* così da evitare di ricevere i messaggi indesiderati.

- `void` **`OnParticipantJoined`**`(AccSession session, AccSecondarySession secondarySession, AccParticipant participant)` viene invocato quando un utente remoto accede ad una sessione di chat. Da qui è possibile verificare l'identità dell'utente e controllare che il numero di partecipanti alla chat non sia maggiore di due.
- `void` **`OnImReceived`**`(AccSession session, AccImSession imSession, AccParticipant participant, AccIm im)` riceve i messaggi istantanei e li inoltra al controller.

2.7.2.4 Classi ausiliarie

Prima di aprire una connessione con il server è possibile configurare la connessione stessa passando ad *AccSession* (con metodo *setPrefsHook*) una implementazione della classe astratta *AccPreferencesHook*. All'interno della classe deve esistere una struttura privata in grado di contenere le preferenze del servizio, costituite da una chiave (stringa formata da un insieme di identificatori separati dal carattere punto) e da un valore (anch'esso di tipo stringa). La scelta della struttura dati più adatta a questo scopo è lasciata al programmatore, che deve anche creare i metodi adatti a leggerne il contenuto. Per impostare o leggere i dati dalla struttura che li contiene devono essere presenti i metodi *getValue*, *setValue*, *getDefaultValue* e *reset* (il valore di default e di reset deve essere *null*). Esiste anche un metodo **`getChildSpecifiers`**`(String specifier)` che richiede una parte della chiave e restituisce un array con tutte le chiavi figlie, ovvero quelle che iniziano con la stringa passata come parametro.

Il metodo *getClientInfo* di *AccSession* restituisce un'istanza della classe *AccClientInfo*. Da qui è possibile impostare la chiave identificativa dell'applicazione, richiesta in fase di autenticazione. Al metodo indicato di seguito deve essere passata come parametro una stringa con formato "`nome_applicazione (Key:chiave_applicazione)`":

`public void` **`setDescription`**`(String Description)`

La classe astratta *AccSecondarySession* rappresenta una qualsiasi sessione avviata con un utente remoto. Da questa classe derivano classi più specifiche, che individuano sessioni di chat, chiamate VoIP, trasferimenti di file, ecc.. L'accettazione di una sessione secondaria non è automatica: quando viene proposta dall'utente remoto si

riceverà una notifica e sarà necessario decidere se procedere o terminare la sessione richiesta. I metodi usati in RoboAdmin sono:

- `public void` **`accept`**`()` invia una conferma all'utente e informa il server che è possibile procedere.
- `public void` **`reject`**`(AccResult reason)` se invocato quando una nuova sessione viene proposta, blocca la sessione secondaria. Il parametro *reason* indica al server la ragione del rifiuto, ma può anche essere omesso. Non tutti i valori disponibili per *reason* sono accettati dal server.
- `public void` **`endSession`**`()` termina una sessione secondaria già stabilita in precedenza.
- `public AccParticipant[]` **`getParticipants`**`()` restituisce un array con i partecipanti alla sessione secondaria.

La classe *AccImSession* deriva da *AccSecondarySession* e rappresenta una sessione di chat con uno o più utenti remoti. Da qui è possibile inviare i messaggi di risposta grazie al metodo:

```
public void sendIm(AccIm im)
```

Il messaggio che viene ricevuto (o che viene inviato) dall'utente remoto è istanza della classe *AccIm*. La creazione di un nuovo messaggio deve essere fatta usando il metodo *createIm* di *AccSession*. Il metodo riportato di seguito è stato usato per leggere il contenuto del messaggio convertito in *text/plain*:

```
public String getConvertedText(String type)
```

2.7.3 Implementazione del bundle OSCAR

La funzione *pump* di *AccSession* richiede obbligatoriamente un ciclo per essere eseguita continuativamente. Per evitare che l'attivazione del bundle OSCAR sia la causa del blocco dell'intera applicazione, è stata creata una nuova classe *OSCARThread* per aprire la connessione su un thread secondario. La comunicazione tra la classe OSCAR (l'unica che implementa l'interfaccia *IComService*) e il thread secondario avviene grazie ad una coda di messaggi presente in *OSCARThread*: la classe *OSCAR* può depositare un comando nella coda dei messaggi di *OSCARThread* e quest'ultima lo processerà prima di leggere gli eventi notificati dal server.

Oltre alle classi sopracitate, che verranno dettagliate nei prossimi paragrafi, sono presenti le implementazioni di *AccPreferencesHook* e del listener *AccEvents*. La classe *AccPreferencesHookRoboAdminImpl* estende la classe astratta *AccPreferencesHook* e implementa un elenco di preferenze, costruito su una struttura dati di tipo

HashMap<String, String>. La classe *AccEventsRoboAdminImpl* implementa i metodi dell'interfaccia *AccEvents*, già descritti in precedenza.

2.7.3.1 Classe Action

Per identificare più facilmente i comandi è stata creata la classe *Action*. Questa classe non può essere istanziata direttamente: quando si vuole creare un comando è sufficiente invocare uno dei metodi statici disponibili, che oltre all'azione, salveranno anche i parametri richiesti. Ad esempio con il metodo seguente viene creata (e restituita al chiamante) una nuova istanza di *Action* contenente il comando *Action.CONNECT* e i parametri *screenname* e *password*:

```
public static Action connect(String screenName, String password)
```

La classe *OSCARThread* potrà leggere il contenuto della classe *Action* grazie ai metodi:

- `public int getAction()` restituisce il codice del comando che deve essere eseguito.
- `public Object[] getArgs()` restituisce un array con i parametri del comando.

2.7.3.2 Classe OSCAR

La classe *OSCAR* ha il compito di implementare i metodi richiesti dall'interfaccia *IComService* e di leggere i parametri dal file *OSCAR.properties* presente nella directory *properties* del progetto.

Quando il bundle viene attivato, vengono letti dal file properties lo *screenname* e la *password*. Successivamente viene avviato il thread secondario *OSCARThread* e viene richiesta ad esso la connessione passando un comando di tipo *Action.CONNECT*.

I metodi di *IComService* non vengono eseguiti direttamente, ma viene richiamato il metodo di *OSCARThread* che salva il comando nella coda dei messaggi.

2.7.3.3 Classe OSCARThread

All'interno del metodo *run*, derivante dalla classe *Thread*, è stato inserito il ciclo per il metodo *pump* di *AccSession*. Nello stesso ciclo si controlla la presenza di comandi inviati dalla classe *OSCAR* e li si processa immediatamente.

I metodi presenti in *OSCARThread* sono analoghi a quelli richiesti da *IComService*, ma sono tutti privati. L'unico metodo visibile dall'esterno (oltre al metodo *run*) è `addAction(Action action)`, che si sincronizza sulla variabile contenente la coda dei messaggi per evitare problemi di concorrenza, e permette di inserire nuovi comandi in fondo alla coda FIFO di tipo *LinkedList<Action>*.

Il metodo privato `performAction(Action action)`, viene richiamato passandogli come parametro l'oggetto *Action* in cima alla coda FIFO. Al suo interno è presente uno switch che, in base al comando ricevuto, estrae gli argomenti e richiama il metodo appropriato di *OSCARThread*.

I metodi privati sono i seguenti:

- `private void` **`connect`**`(String screenName, String password)` apre la connessione con il server BOSS, se il login ha successo.
- `private void` **`sendMessage`**`(Object receiver, String message)` analogamente a quanto avveniva nei precedenti bundle, invia un messaggio di testo ad un amministratore. Il messaggio viene inviato con formato *text/plain* sulla sessione di chat (di tipo *AccImSession*) contenuta nel parametro *receiver*.
- `private void` **`disconnect`**`()` invoca il metodo *signOff* di *AccSession* per chiudere la connessione con il server.
- `private void` **`updateBuddyList`**`()` scandisce tutti i gruppi della Buddy List per eliminare gli amministratori che non sono più autorizzati.
- `private void` **`printBuddyList`**`()` viene usato per scopi di logging. Mostra il contenuto della Buddy List, analizzando tutti i gruppi presenti.
- `private void` **`addBuddy`**`(AccUser user)` controlla che *user* sia un amministratore autorizzato e lo inserisce nella Buddy List in caso affermativo. Se *user* non è autorizzato viene bloccato (e rimosso dalla Buddy List se ne faceva parte).
- `private void` **`removeBuddy`**`(AccUser user)` blocca un utente non autorizzato, rimuovendolo anche dalla Buddy List.

2.7.4 Istruzioni di installazione

Dal momento che la libreria Java usata in RoboAdmin fa uso di alcuni componenti scritti in linguaggio nativo e dipendenti dal sistema operativo, è importante comprendere quali altre operazioni devono essere fatte per rendere il bundle funzionante. Il file JAR incluso nel progetto RoboAdmin è multi piattaforma e si adatta automaticamente alle librerie native installate.

L'ultima versione delle librerie di AOL è compatibile con Windows, Linux e Mac OS X. Le procedure riportate nei prossimi paragrafi riguardano i sistemi Windows e Linux.

2.7.4.1 Installazione del bundle su Windows

Per far funzionare il bundle su Windows sono richiesti un gran numero di file *dll*. I file presenti nella directory *lib\oscar* del progetto, devono essere copiati in una delle cartelle del path di Windows per poter essere caricati correttamente.

2.7.4.2 Installazione del bundle su Linux

La libreria AccSDK deve essere compilata per la distribuzione del sistema operativo che si intende usare. In particolare è richiesta la presenza di NSS e NSPR per poter usare la crittografia.

L'ultima versione dei sorgenti di NSS (che includono anche NSPR) possono essere trovati sul sito *ftp.mozilla.org*. Estraendo il contenuto dell'archivio e spostandosi nella directory *mozilla/security/nss* via shell, è possibile avviare il comando *make nss_build_all* (testato solo su sistemi a 32 bit, per sistemi a 64 bit bisogna specificare il parametro *USE_64=1* dopo il comando *make*), che compila e installa i componenti di NSS.

Le librerie native per OSCAR su Linux possono essere scaricate dal sito AOL Developer Network (*dev.aol.com*). Una volta estratti tutti i file possono essere copiati quelli con estensione *so* dalla directory *lib/release* alla directory di sistema */lib*, per fare in modo che la libreria venga riconosciuta correttamente.

2.8 Servizio di comunicazione con Yahoo! Messenger Protocol

Yahoo! Messenger Protocol (YMSG) [21] è il protocollo usato dal client di messaggistica istantanea Yahoo! Messenger. Grazie alla libreria Open Source OpenYMSG, RoboAdmin può accedere a questa rete e sfruttare la maggior parte delle funzionalità di questo servizio.

2.8.1 Protocollo YMSG

YMSG è il protocollo proprietario di Yahoo! usato nel client Yahoo! Messenger. Oltre alle classiche funzionalità di messaggistica istantanea, il protocollo offre un servizio di presenza, ricezione di messaggi non in linea, trasferimento file, conferenze testuali e vocali e videoconferenze.

Il protocollo è di tipo client/server, e più server (gestiti direttamente da Yahoo!) vengono usati dai client per scopi diversi. Esattamente come avveniva con il protocollo MSNP descritto in precedenza, anche in questo caso le informazioni scambiate con i server sono in chiaro e possono essere intercettate da un attaccante. Nelle ultime

versioni del protocollo la fase di autenticazione viene eseguita in modo sicuro grazie alla crittografia di TLS, ma lo scambio dei messaggi istantanei e degli eventi riguardanti l'elenco contatti (detto roster) vengono ancora trasmessi in chiaro.

Il reverse engeenering che ha permesso di capire il significato dei messaggi scambiati con il server, non è ancora stato completato. Questo è dovuto al fatto che il significato dei comandi inviati e ricevuti è legato ad un semplice numero, che può cambiare in base alla versione del protocollo utilizzata.

2.8.1.1 Struttura di un pacchetto YMSG

Ad eccezione della fase di autenticazione, lo scambio di messaggi e la gestione del roster vengono gestite da un unico server. La comunicazione con il server si basa su una connessione TCP. La chiusura di questa connessione esclude l'utente dal servizio e lo rende offline ai propri contatti.

Il protocollo YMSG si posiziona nel livello applicativo, ed è costituito da un header di 20 byte. Nella prima parte dello header sono presenti il codice identificativo del protocollo (4 byte, che corrispondono sempre alla stringa ASCII 'YMSG'), la sua versione (2 byte) e le dimensioni della parte dati (2 byte). I successivi 4 byte vengono usati per identificare il servizio. Subito prima della parte dati si trovano 4 byte per lo stato del server e 4 byte usati per identificare la sessione.

La parte dati contiene i parametri del comando. Anche questa zona del pacchetto ha una struttura ben definita: ciascun parametro è identificato da una chiave numerica e da un valore, entrambi di dimensione variabile. Per separare i campi della parte dati viene inserito un blocco di due byte contenenti il numero esadecimale C0 80.

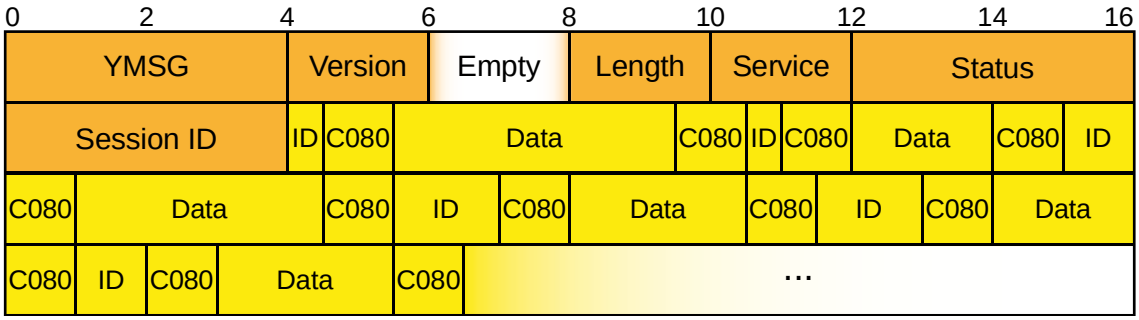


Figura 10: struttura di un pacchetto YMSG

2.8.1.2 Fase di login

La fase di login viene gestita da un server separato detto login server. Con esso il client apre una connessione cifrata usando il protocollo TLS. L'autenticazione al servizio avviene in due passi distinti.

Inizialmente il client invia il proprio username (noto anche come Yahoo! Id) e, in caso di risposta affermativa, vengono restituiti dal server l'id di sessione e una MD5 challenge string.

Nel secondo passo il client aggiunge alla password il valore della MD5 challenge string ed esegue un hash MD5. Il risultato ottenuto viene inviato al server assieme ad alcune informazioni riguardanti il client e il sistema.

Se l'operazione ha successo la comunicazione procede sul server principale, e la connessione con il login server viene chiusa.

2.8.2 Libreria OpenYMSG

La libreria OpenYMSG [22], scritta in Java e attualmente in fase di alfa test, è una evoluzione della precedente libreria JYMSG9 [23]. A differenza del predecessore, OpenYMSG supporta la versione 16 del protocollo e permette il login cifrato con TLS.

Per il momento sono disponibili soltanto le funzionalità di messaggistica istantanea, il trasferimento di file, le conferenze testuali e le chatroom.

2.8.2.1 Classe Session

La classe *Session*, presente nel package *org.openymsg.network*, rappresenta una connessione con protocollo YMSG e offre una serie di metodi per accedere al servizio e per interagire con esso.

Le impostazioni di connessione di default possono essere cambiate passando al costruttore un'istanza della classe *DirectConnectionHandler*. Se ha avuto successo la procedura di autenticazione, attraverso la classe Session si può impostare il proprio stato di presenza, accettare inviti da parte dell'utente remoto, inviare messaggi e molto altro ancora. È possibile associare una istanza di *Session* ad una implementazione dell'interfaccia *SessionListener*, per essere notificati in modo asincrono degli eventi ricevuti dal server.

I metodi utili al funzionamento di RoboAdmin sono i seguenti:

- `public void` **`login`**`(String username, String password)` apre la connessione con il server ed esegue l'autenticazione cifrata con TLS. Questo metodo è bloccante: quando si ritorna al chiamante è possibile invocare il metodo *getSessionStatus* della classe *Session* per capire se la connessione ha avuto successo. In caso di insuccesso vengono comunque lanciate le eccezioni *LoginRefusedException*, *FailedLoginException* e *AccountLockedException*. Se

l'eccezione è di tipo *LoginRefusedException* si può leggere la risposta del server usando il metodo *getStatus()*.

- `public void` **`logout`**`()` chiude la connessione con il server, impostando lo stato dell'utente ad offline.
- `public void` **`addSessionListener`**`(SessionListener sessionListener)` associa alla classe *Session* una implementazione di *SessionListener*, per ricevere in modo asincrono gli eventi inviati dal server. È disponibile anche il metodo *removeSessionListener*, per non ricevere più gli eventi del server.
- `public String` **`sendMessage`**`(String to, String message)` invia un messaggio di testo all'utente il cui Yahoo! Id è passato come primo parametro.
- `public SessionState` **`getSessionStatus`**`()` restituisce lo stato della connessione. *SessionState* è un enumerativo, contenente tutti gli stati possibili. Se lo stato è *SessionState.LOGGED_ON*, significa che la connessione è avvenuta correttamente.
- `public Roster` **`getRoster`**`()` permette di accedere all'elenco contatti dell'utente.
- `public void` **`acceptFriendAuthorization`**`(String friend)` può essere invocato se si vuole rispondere affermativamente ad un utente che vuole aggiungere RoboAdmin al proprio elenco contatti.
- `public void` **`rejectFriendAuthorization`**`(SessionAuthorizationEvent ev, String friend, String msg)` invia una risposta negativa se l'utente che ha richiesto di aggiungere RoboAdmin al proprio roster non è affidabile. Il parametro *msg* può essere usato per indicare le motivazioni del rifiuto.
- `public void` **`declineConferenceInvite`**`(SessionConferenceEvent ev, String msg)` rifiuta un invito ad una chat con più utenti.

2.8.2.2 Interfaccia SessionListener e classe SessionAdapter

L'interfaccia *SessionListener*, presente nel package *org.openymsg.network.event*, può essere implementata per ricevere gli eventi dal server in modo asincrono. L'unico metodo reso disponibile dall'interfaccia è il seguente:

`void` **`dispatch`**`(FireEvent event)`

Il metodo riceve qualsiasi pacchetto del protocollo e deve trasformarlo in un pacchetto comprensibile dall'applicazione.

L'implementazione di default del metodo *dispatch* viene fornita (nello stesso package) dalla classe *SessionAdapter*. Vengono ricevuti tutti i pacchetti in entrata e, in base al codice del comando, si estraggono i parametri previsti e si invoca uno dei metodi pubblici resi disponibili. I metodi offerti sono quasi sempre vuoti, ma possono essere modificati in una classe derivata, così da processare gli eventi che si considerano interessanti.

I metodi di *SessionAdapter* che sono stati modificati sono i seguenti:

- `public void` **`connectionClosed`**`(SessionEvent event)` notifica all'applicazione quando una connessione viene chiusa, in modo che possano essere presi i necessari provvedimenti.
- `public void` **`messageReceived`**`(SessionEvent event)` viene invocato quando un messaggio istantaneo viene ricevuto. Il parametro contiene già le indicazioni sul mittente e sul testo del messaggio ricevuto.
- `public void` **`listReceived`**`(SessionListEvent event)` indica che il roster è stato ricevuto dal server nella sua interezza. Quando si verifica questo evento è possibile aggiornare il roster, eliminando gli utenti non autorizzati.
- `public void` **`contactRequestReceived`**`(SessionEvent event)` viene invocato quando un utente remoto vuole aggiungere RoboAdmin al proprio roster.
- `public void` **`conferenceInviteReceived`**`(SessionConferenceEvent event)` viene ricevuto quando l'utente remoto tenta di avviare una nuova chat testuale o vocale con più utenti. Nel nostro caso le conferenze devono essere evitate: l'invito si può rifiutare invocando il metodo *declineConferenceInvite* della classe *Session*.

2.8.2.3 Classi ausiliarie

Assieme alle classi precedentemente citate ne sono presenti molte altre. Di seguito verranno indicate solo quelle utilizzate nel codice del bundle di RoboAdmin.

La classe *DirectConnectionHandler*, nel package *org.openymsg.network*, viene passata al costruttore della classe *Session* per modificare le opzioni di connessione di default. Le uniche impostazioni modificabili sono il nome host e la porta remota su cui aprire la connessione. Se il server non risponde sulla porta di default, la connessione viene ritentata automaticamente su altre porte di fallback. Durante il test di RoboAdmin, le

impostazioni di default non hanno causato alcun problema. Dal file *Yahoo.properties* è comunque possibile impostare il numero di porta del server, per velocizzare l'accesso nel caso in cui il controllo della porta impieghi troppo tempo. I nuovi nome host e numero di porta possono essere passati a *DirectConnectionHandler* direttamente dal suo costruttore.

La classe *SessionEvent*, nel package *org.openymsg.network.event*, rappresenta un generico pacchetto ricevuto dal server, il cui significato cambia in base al metodo di cui *SessionEvent* è parametro. I campi principali sono quattro e possono essere ottenuti dai metodi *getFrom*, *getTo*, *getMessage* e *getTimestamp*. Esiste anche un metodo *getStatus*, che restituisce un intero con lo stato del server, letto direttamente dallo header del pacchetto. In alcuni metodi di *SessionAdapter*, il valore dei campi di *SessionEvent* può essere privo di significato. Ad esempio il campo *timestamp* è disponibile per il metodo *contactRequestReceived*, ma non in *messageReceived*.

La classe *Roster*, contenuta nel package *org.openymsg.roster*, rappresenta la lista contatti dell'utente correntemente connesso al servizio. La classe è stata costruita implementando l'interfaccia *Set<YahooUser>*, pertanto sono presenti tutti i metodi previsti dall'interfaccia. Tramite i metodi *add* e *remove* è possibile inserire o rimuovere un oggetto di tipo *YahooUser* dal roster. Oltre i metodi richiesti dall'interfaccia, è presente anche *containsUser*, a cui si passa una stringa con lo username dell'utente che si sta cercando e viene restituito un valore booleano, settato a *true* se l'utente è presente nel roster.

La classe *YahooUser* rappresenta un utente del servizio, e conserva tutti i dati che lo riguardano. Tipicamente i metodi della libreria richiedono soltanto una stringa con il Yahoo! Id dell'utente, ma può capitare che altri restituiscano un'istanza di *YahooUser*, con altri dati non necessari. In RoboAdmin i dati aggiuntivi non vengono considerati, e viene usato soltanto il metodo *getId()* quando si vuole estrarre l'Id dell'utente.

2.8.3 Implementazione del bundle Yahoo!

Dopo l'attivazione del bundle, vengono letti i parametri di connessione, lo username e la password dal file *Yahoo.properties* nella directory *properties* del progetto.

Prima di eseguire il login viene associata alla classe Session l'implementazione di SessionAdapter:

`class` **`SessionRoboAdminAdapter`** `extends SessionAdapter` implementa tutti i metodi indicati nel precedente paragrafo **2.8.2.2**, elaborando solo i pacchetti di interesse per RoboAdmin. I metodi più interessanti sono *messageReceived* e

contactRequestReceived. Il primo riceve un messaggio istantaneo: prima di passarne il contenuto al controller, controlla che il mittente sia un amministratore autorizzato (aggiungendolo al roster se non ne fa parte). Il secondo metodo riceve le notifiche quando un utente vuole aggiungere RoboAdmin al proprio roster: per procedere con l'autorizzazione, il mittente viene verificato nel data base.

La classe principale del bundle, denominata *Yahoo*, implementa l'interfaccia *IComService*. I metodi più interessanti sono:

- *disconnecT* invoca il corrispondente metodo *logout* di *Session*, che rende l'utente offline e chiude la connessione.
- *sendMessagE* invia un messaggio di testo verso l'utente specificato, invocando l'omonimo metodo nella classe *Session*. Il primo parametro, istanza di *String*, rappresenta lo Yahoo! Id dell'utente a cui si vuole spedire il messaggio.

Nonostante nel client ufficiale Yahoo! Messenger sia possibile bloccare un utente presente nel proprio roster, la libreria usata in RoboAdmin non offre questa possibilità. Gli utenti non autorizzati, in questo caso, vengono semplicemente ignorati.

Nella classe *Yahoo* sono stati aggiunti alcuni metodi privati:

- `private void addUser(String contact)` aggiunge l'utente al roster, se non ne fa già parte.
- `private void updateRoster()` legge il nome di tutti gli utenti del roster e rimuove quelli che non sono presenti nella tabella *Accept* di *RoboAdminDB*.
- `private void printRoster()` viene usata per scopi di logging. Scandisce l'intero roster e mostra lo Yahoo! Id degli utenti che ne fanno parte.

Capitolo 3. Confronto dei bundle di comunicazione

Nonostante il tentativo di implementare alcune caratteristiche comuni ad ogni bundle di comunicazione, i limiti delle librerie hanno portato ad una certa diversificazione. Questo capitolo è nato con l'intento di informare l'amministratore che fa uso di RoboAdmin dei pregi e dei difetti derivanti dall'uso di alcuni bundle di comunicazione.

Per ogni bundle verranno anche indicati i package delle librerie che devono essere aggiunti al file di configurazione *config.properties*, presente nella directory *conf* del progetto. Il nome di ciascun package (separato dagli altri con il carattere virgola) deve essere impostato nella proprietà *org.osgi.framework.system.packages.extra*.

Un parametro importante che potrebbe determinare la scelta di un bundle è la politica anti-spam (e anti-bot) adottata da ciascun servizio di messaggistica istantanea. Tipicamente questi servizi non sono pensati per inviare tanti messaggi in rapida sequenza, cosa che però si verifica frequentemente quando si richiamano comandi di shell con RoboAdmin. L'interprete di RA invia un messaggio istantaneo per ogni riga di risposta al comando restituita della shell, pertanto devono essere valutate le capacità di ciascuna libreria di regolare il flusso di messaggi in uscita e le reazioni del server se si riconoscono comportamenti sospetti. Il test anti-spam è stato fatto richiedendo a RoboAdmin l'esecuzione del comando "`cat prova.txt`" (su sistema operativo Linux). Il file *prova.txt* contiene cento righe, ciascuna numerata ordinatamente con numeri da 1 a 100.

Oltre alle caratteristiche elencate nel capitolo 2, è importante tenere in considerazione i requisiti di sicurezza dei protocolli esaminati, aggiornati alla versione implementata dalle librerie. Il contenuto dei pacchetti inviati e l'uso della crittografia sono stati controllati caso per caso con Wireshark [24]. Questo programma è in grado di catturare tutti i pacchetti in transito sulla scheda di rete, e permette di esplorare il contenuto di ciascuno di essi riconoscendo i protocolli grazie agli header.

3.1 Caratteristiche del bundle IRC

- **Login e logout**: l'autenticazione al server è prevista dal protocollo IRC, ma raramente è obbligatoria. In RoboAdmin la procedura di login viene richiamata soltanto se sono presenti nel file di configurazione i parametri *login* e *password*, e può essere usata nel caso in cui RoboAdmin sia stato registrato come operatore sul server. La disconnessione dal server è permessa, e implica il logout.

- **Invio e ricezione di messaggi istantanei**: l'elaborazione dei messaggi in ingresso viene fatta solo se questi provengono dal canale privato. I messaggi sul canale pubblico vengono comunque ricevuti e, se contengono riferimenti espliciti a RoboAdmin, il bundle risponderà invitando l'interlocutore a contattarlo in privato. I messaggi privati vengono inviati direttamente al controller, dal momento che con IRC non è possibile verificare l'identità del mittente: in questo caso la politica di sicurezza è affidata interamente all'interprete di RoboAdmin.
- **Comportamento in chat con più utenti**: il significato di "chat con più utenti" nel protocollo IRC è diverso da quello usato per altri protocolli. In questo caso RoboAdmin deve essere sempre presente su un canale pubblico se vuole essere rintracciato dall'amministratore, ma non deve interagire rispondendo ai comandi sul canale pubblico.
- **Gestione automatica dei contatti**: con il protocollo IRC la gestione dei contatti non è permessa, dal momento che non è possibile riconoscere un amministratore dal suo nickname. Per verificare che un contatto sia autorizzato, questo dovrà contattare RoboAdmin in privato e inviare un messaggio con il comando di login. Se il nome utente o la password non sono riconosciuti, RoboAdmin tenterà di fare il kick dell'utente dal canale (riesce solo se RA è operatore del canale).
- **Crittografia**: nonostante le più moderne versioni di IRC supportino la connessione cifrata, la libreria PircBot non implementa alcun meccanismo di crittografia, quindi la connessione al server e la comunicazione possono essere facilmente interpretate se intercettate da un utente malintenzionato.
- **Istruzioni di installazione**: inserire nel file di configurazione di Felix il package *org.jibble.pircbot*.
- **Test anti-spam**: la libreria PircBot si occupa di ritardare l'invio dei messaggi, conservandoli in una coda di uscita. La frequenza di invio di default è di un messaggio al secondo, abbassata a mezzo secondo in RoboAdmin. Con questa frequenza l'intero file è arrivato a destinazione con successo, anche se con molto ritardo. Con frequenze di invio inferiori, il server (il test è stato fatto su *irc.freenode.net*) ha chiuso automaticamente la connessione con motivazione *Excess Flood*.
- **Uso consigliato**: l'accesso autenticato di RoboAdmin al server è fortemente consigliato. Questo permetterebbe di impostare il bot come operatore, per eseguire il kick di utenti non autorizzati a comunicare. Questo bundle risulta utile se

RoboAdmin non viene usato per il trasferimento di informazioni riservate, dal momento che la comunicazione non è protetta. La mancanza di un sistema di riconoscimento dell'interlocutore garantisce meno protezione, rispetto a quella offerta dagli altri bundle. È necessario considerare anche un ritardo considerevole, quando si devono ricevere risposte formate da più messaggi.

3.2 Caratteristiche del bundle MSNP

- **Login e logout**: per poter fare il login è necessario registrare un Windows Live ID da usare con RoboAdmin. Le informazioni necessarie al login devono essere inserite nel file *properties*. La procedura di logout offerta dalla libreria chiude anche la connessione con il server.
- **Invio e ricezione di messaggi istantanei**: per ogni messaggio istantaneo ricevuto, RoboAdmin controlla se l'utente remoto è autorizzato a comunicare. Se il controllo ha successo, il contenuto del messaggio viene inoltrato al controller. L'invio di un messaggio di risposta viene fatto sulla stessa sessione di chat da cui è stato ricevuto il comando.
- **Comportamento in chat con più utenti**: quando una nuova sessione di chat (switchboard) viene aperta, o quando un nuovo utente si aggiunge alla switchboard, si controlla che il numero di utenti collegati non sia superiore a due. Se ci sono più di due partecipanti, RoboAdmin abbandona immediatamente la conversazione.
- **Gestione automatica dei contatti**: il bundle MSNP è in grado di aggiungere, rimuovere e bloccare gli utenti del servizio. Un amministratore che vuole ricevere i pacchetti presenza di RoboAdmin può aggiungerlo alla propria lista contatti: RoboAdmin provvederà automaticamente ad inserire l'utente nella *Allow List* se è stato riconosciuto o alla *Block List* in caso contrario. Un utente bloccato può essere sbloccato se ad una successiva richiesta viene considerato attendibile.
- **Crittografia**: nella versione 12 del protocollo usata in RoboAdmin gli unici pacchetti cifrati con TLS sono quelli relativi all'invio della password durante l'autenticazione. Il resto della comunicazione è in chiaro: i pacchetti presenza e i messaggi istantanei possono essere intercettati e facilmente interpretati.
- **Istruzioni di installazione**: inserire nel file di configurazione di Felix i package *net.sf.jml, net.sf.jml.event, net.sf.jml.impl, net.sf.jml.message*.
- **Test anti-spam**: la libreria Java Messenger Library ritarda automaticamente l'invio dei messaggi per evitare la disconnessione dal servizio. Vengono inviati in ordine nove messaggi alla volta, seguiti da una attesa di cinque secondi non configurabile.

- **Uso consigliato**: il bundle MSNP soddisfa la maggior parte dei requisiti imposti in fase di progettazione, ma non garantisce la riservatezza della comunicazione. È importante evitare lo scambio di dati sensibili con questo bundle. Si consiglia di registrare un nuovo account Windows Live da usare esclusivamente con RoboAdmin: se si tenta il login con il proprio account personale, tutti i contatti non autorizzati nel data base verranno rimossi.

3.3 Caratteristiche del bundle Skype

- **Login e logout**: con la libreria usata da RoboAdmin il login e il logout non sono permessi. Il bundle Skype controlla in ogni suo metodo che il client sia già connesso e autenticato con lo Skype Id richiesto dall'amministratore nei file di configurazione. L'interprete di RoboAdmin può comunque bloccare il plugin per non ricevere nuovi messaggi, ma il client rimarrà connesso alla rete.
- **Invio e ricezione di messaggi istantanei**: per ogni messaggio ricevuto viene verificata l'identità del mittente prima dell'inoltro del contenuto al controller. I messaggi da un utente non autorizzato vengono ignorati, e l'utente viene rimosso dall'elenco contatti. I messaggi di risposta vengono inviati sulla stessa sessione di chat da cui sono arrivate le richieste dell'amministratore.
- **Comportamento in chat con più utenti**: il protocollo prevede la presenza di chat con più utenti. Alla ricezione di un messaggio istantaneo si controlla il numero di utenti in chat, e se sono più di due RoboAdmin abbandona la conversazione.
- **Gestione automatica dei contatti**: RoboAdmin riconosce i nuovi utenti solo quando questi tentano di inviare dei messaggi istantanei. Se la verifica dal data base dimostra che si tratta di amministratori autorizzati, vengono aggiunti all'elenco contatti, altrimenti vengono rimossi. La libreria permette di bloccare un utente del servizio: in questo modo nessun evento riguardante questo utente verrà inoltrato a RoboAdmin. Questa pratica non è stata adottata nel nostro caso perché renderebbe impossibile lo sblocco automatico dell'utente.
- **Crittografia**: I pacchetti TCP e UDP scambiati in peer-to-peer con altri client Skype e con il login server hanno tutti un contenuto criptato. L'algoritmo usato per la crittografia non è stato diffuso pubblicamente, quindi non è possibile verificarne l'affidabilità.
- **Istruzioni di installazione**: aggiungere nel file di configurazione di Felix i package *com.skype, com.skype.connector, org.eclipse.swt*. Per installare i componenti in codice nativo fare riferimento all'apposito paragrafo nel capitolo 2.

- **Test anti-spam**: se si inviano fino a 9 messaggi senza attesa, questi vengono recapitati a destinazione in ordine. Con l'invio del decimo messaggio la funzione *send* della libreria Skype4Java si blocca, e i messaggi successivi non vengono inviati.
- **Uso consigliato**: l'uso della crittografia del protocollo permette di trasmettere in sicurezza anche i dati riservati. Il protocollo è anche piuttosto flessibile, e in grado di aprire connessioni anche in presenza di configurazioni di rete molto complesse. Per utilizzare il bundle è importante tenere in considerazione il fatto che il client ufficiale di Skype deve essere attivo e connesso alla rete: è necessario evitare l'accesso all'interfaccia grafica del client, sulla quale vengono visualizzati i messaggi scambiati con l'amministratore, per evitare che si vanifichi completamente la riservatezza della comunicazione. I requisiti di sistema del client limitano l'uso di questo bundle solo ad alcuni sistemi operativi e configurazioni hardware. Anche in questo caso conviene creare un nuovo account per RoboAdmin, dal momento che gli utenti non autorizzati presenti nell'elenco contatti vengono rimossi all'avvio del bundle.

3.4 Caratteristiche del bundle XMPP

- **Login e logout**: il server su cui connettersi, il nome utente e la password possono essere specificati dall'amministratore nel file properties. Gli eventuali errori di connessione e autenticazione vengono riportati a RoboAdmin sotto forma di eccezioni facilmente interpretabili. Quando si esegue la disconnessione dal server, il logout avviene implicitamente.
- **Invio e ricezione di messaggi istantanei**: prima di ricevere un messaggio, RoboAdmin viene avvisato dell'apertura di una nuova sessione di chat. In questa occasione viene controllato che l'interlocutore sia un amministratore autorizzato e si associa un listener per ricevere i suoi messaggi. Se l'utente remoto non viene riconosciuto, il listener non viene associato e tutti i suoi messaggi andranno persi. Il protocollo permette di inviare e ricevere i messaggi istantanei di utenti iscritti ad altri server XMPP. I messaggi di risposta vengono inviati da RoboAdmin sulla stessa chat da cui sono arrivate le domande.
- **Comportamento in chat con più utenti**: per il protocollo XMPP non è prevista l'esistenza di chat con più di due utenti, che sono disponibili solo come estensione nella libreria SmackX. RoboAdmin non implementa questa estensione, quindi tutti gli inviti a chat multi utente verranno automaticamente rifiutati dalla libreria.

- **Gestione automatica dei contatti**: per poter contattare RoboAdmin è necessario aggiungerlo al proprio roster. In questa occasione il contatto viene inserito o rimosso anche nel roster di RoboAdmin. Il protocollo non permette di inviare messaggi ad utenti che risultano offline, quindi rimuovendo un utente dal roster di RA, questo non potrà più né contattarlo né ricevere i suoi pacchetti presenza. L'intero roster viene controllato all'avvio, per eliminare tutti gli utenti non autorizzati a comunicare con RoboAdmin.
- **Crittografia**: in RoboAdmin è stato imposto l'uso di una connessione cifrata con il server. Il protocollo di sicurezza da usare viene concordato con il server non appena la connessione si è stabilita. I dati per l'autenticazione, i messaggi istantanei e i pacchetti presenza sono tutti protetti, e non potranno essere interpretati se intercettati da un utente malintenzionato.
- **Istruzioni di installazione**: per caricare correttamente la libreria inserire nel file di configurazione i package *org.jivesoftware.smack, org.jivesoftware.smack.packet*.
- **Test anti-spam**: vengono ricevuti correttamente i primi 15 messaggi. I successivi vengono rimandati indietro a RoboAdmin come se li avesse scritti l'utente remoto (e quindi possono venire interpretati).
- **Uso consigliato**: la connessione cifrata e la rigida politica di gestione dei contatti rendono questo bundle la scelta ideale per lo scambio di informazioni riservate. È comunque necessario controllare che anche il client usato dall'amministratore apra una connessione sicura con il server, per evitare di perdere la riservatezza della comunicazione. La connessione al server potrebbe non avvenire se la configurazione della rete su cui si trova RoboAdmin è troppo complessa. Se in RoboAdmin si esegue il login con un account di uso comune, il roster verrà svuotato degli utenti non autorizzati sul data base.

3.5 Caratteristiche del bundle OSCAR

- **Login e logout**: il login è permesso usando lo screenname e la password di un account AIM precedentemente creato. Se l'host del client si trova in una rete protetta da firewall o NAT, la connessione viene comunque garantita grazie all'uso del protocollo STUN [25]. Quando si esegue il logout la connessione viene automaticamente chiusa.
- **Invio e ricezione di messaggi istantanei**: quando l'utente remoto avvia una nuova sessione di chat, RoboAdmin verifica che l'interlocutore sia autorizzato a comunicare. La sessione deve essere esplicitamente accettata o rifiutata, e solo se

accettata si riceveranno i messaggi istantanei. Questo assicura di ricevere messaggi solo da utenti affidabili, e consente di evitare di controllare il mittente di ogni messaggio ricevuto. I messaggi di risposta dall'interprete verranno spediti sulla sessione di chat già attiva con l'utente remoto.

- **Comportamento in chat con più utenti**: quando si riceve l'invito ad una nuova sessione di chat è possibile che questa sia tra più di due utenti. Alla ricezione dell'invito o se nuovi utenti si inseriscono nella conversazione, RoboAdmin controlla il numero di partecipanti e esce dalla sessione se ce ne sono più di due.
- **Gestione automatica dei contatti**: la libreria non permette di controllare quando un utente aggiunge RoboAdmin alla propria lista contatti. La lista contatti di RoboAdmin viene aggiornata quando si aprono nuove sessioni di chat. Il partecipante alla conversazione viene verificato sul data base e aggiunto o rimosso dalla Buddy List se è autorizzato oppure no. All'avvio del bundle viene controllato il contenuto del roster per eliminare le entry non autorizzate.
- **Crittografia**: controllando la comunicazione di RoboAdmin attraverso Wireshark si possono notare due fasi. Nella prima fase viene usata la crittografia TLS per autenticarsi al login server di AOL. La seconda fase prevede la connessione con crittografia SSL ad un altro server, sempre di proprietà di AOL. Sul secondo server vengono scambiati i messaggi e i pacchetti presenza, in modo protetto così da essere difficilmente interpretabili se intercettati.
- **Istruzioni di installazione**: inserire nel file di configurazione di Felix il package *com.aol.acc*. Per installare i componenti in codice nativo seguire le istruzioni riportate nell'apposito paragrafo del capitolo 2.
- **Test anti-spam**: dal momento che lo screenname usato dal bundle OSCAR era già stato registrato come bot tutti i 100 messaggi sono stati inviati in ordine, senza ritardi o blocchi di alcun tipo. In questo caso, però, le limitazioni arrivano sul lungo periodo: il contratto di licenza parla di un limite di 10.000 messaggi al giorno e di 150.000 messaggi al mese. I messaggi in eccesso vengono filtrati dal server.
- **Uso consigliato**: il bundle OSCAR può essere usato per trasmettere messaggi riservati, grazie alla connessione cifrata con SSL. Purtroppo la gestione degli utenti non è completa: un utente deve inviare un messaggio istantaneo a RoboAdmin per essere aggiunto alla sua Buddy List e riceverne i pacchetti presenza. Le librerie fornite da AOL richiedono l'installazione di componenti di componenti in codice nativo, pertanto il funzionamento del bundle è limitato solo ad alcuni sistemi

operativi. Si consiglia di creare un nuovo account per RoboAdmin e di registrarlo come bot sul sito AOL, così da evitare i filtri anti-spam applicati dal server agli utenti comuni.

3.6 Caratteristiche del bundle Yahoo!

- **Login e logout**: il login deve essere fatto per poter accedere al servizio. I dati richiesti sono lo username e la password di un account registrato sul sito Yahoo!. Dal file *properties* è possibile configurare anche la porta remota, ma attualmente la connessione avviene senza problemi anche collegandosi alla porta di default. Quando si esegue il logout viene chiusa la connessione al server e la libreria interrompe il thread secondario usato per l'invio e la ricezione dei messaggi.
- **Invio e ricezione di messaggi istantanei**: per ogni messaggio ricevuto, RoboAdmin controlla che il mittente sia autorizzato a comunicare. La libreria non permette di bloccare un utente, pertanto i messaggi inviati da utenti non autorizzati verranno semplicemente ignorati. Il protocollo non distingue le sessioni di chat, pertanto i messaggi verranno inviati semplicemente specificando il Yahoo! Id del destinatario.
- **Comportamento in chat con più utenti**: le chat multi utente del servizio Yahoo! sono chiamate conferenze, e vengono ricevute in modo diverso rispetto ai messaggi istantanei tra due utenti. Tutti gli inviti a conferenze ricevuti da RoboAdmin vengono immediatamente rifiutati.
- **Gestione automatica dei contatti**: il protocollo permette di comunicare con utenti non presenti nella propria lista contatti, pertanto RoboAdmin controlla se il mittente di ogni messaggio ricevuto è presente nel roster. L'utente viene aggiunto al roster se è autorizzato o viene rimosso in caso contrario. La libreria notifica anche quando un utente aggiunge RoboAdmin al proprio roster: se l'utente è un amministratore autorizzato, la sua richiesta viene confermata e viene aggiunto al roster di RoboAdmin, così da ricevere i pacchetti presenza. I contatti non autorizzati presenti nel roster vengono rimossi all'avvio del bundle.
- **Crittografia**: la comunicazione con il server è prevalentemente in chiaro, e il testo dei messaggi è facilmente interpretabile se vengono intercettati i pacchetti. L'autenticazione al servizio viene fatta su una connessione sicura TLS ad un secondo server, sempre gestito da Yahoo!.
- **Istruzioni di installazione**: inserire nel file di configurazione di Felix i package *org.openymsg.network*, *org.openymsg.network.event*, *org.openymsg.roster*.

- **Test anti-spam**: in questo caso vengono ricevuti soltanto i primi venti messaggi, ordinati e senza ritardo. I pacchetti successivi vengono filtrati dal server, e non raggiungono il destinatario. Il filtro dei messaggi in uscita non è definitivo: l'amministratore potrà inviare altri comandi a RoboAdmin e riceverne le risposte.
- **Uso consigliato**: si sconsiglia l'uso di questo bundle se si vogliono inviare informazioni riservate a RoboAdmin. La libreria OpenYMSG è ancora in versione alfa, pertanto il suo funzionamento non è sempre impeccabile e il numero di funzioni del protocollo implementate è ancora piuttosto basso. Per evitare di perdere i contatti considerati non autorizzati, è preferibile registrare un nuovo account Yahoo! da usare unicamente con RoboAdmin.

Capitolo 4. Conclusioni

RoboAdmin è nato per permettere ad un amministratore di comunicare con il proprio sistema attraverso un qualsiasi servizio di chat. I nuovi bundle di comunicazione sono stati creati proprio per raggiungere questo obiettivo: l'amministratore può connettersi a RoboAdmin usando il bundle di comunicazione che meglio risponde alle proprie esigenze. Il gran numero di servizi supportati permette all'amministratore di ripiegare su un altro bundle di comunicazione, se quello preferito è stato disabilitato per motivi di sicurezza.

L'uso dei ben noti client di messaggistica istantanea garantisce anche una maggiore facilità d'uso per i principianti, che non hanno più bisogno di usare i più complicati client per l'amministrazione remota. RoboAdmin viene visto sulle reti in cui si connette come un normale utente con cui comunicare: non è più indispensabile conoscere e inserire manualmente l'ubicazione del server per aprire una connessione con esso, ma basta individuare il suo username.

Una caratteristica comune a tutti i servizi di comunicazione e che deve sempre essere tenuta in considerazione è la registrazione di un account per i servizi di comunicazione che si vogliono usare. I dati inseriti durante la registrazione devono permettere all'amministratore di capire con assoluta certezza che l'utente remoto è RoboAdmin: per garantire l'identità bisogna usare un nome utente ben riconoscibile dall'amministratore e inserire alcuni dati aggiuntivi per facilitare il riconoscimento. Al contrario, bisogna evitare che il nome utente o gli altri dati inseriti permettano ad un attaccante di riconoscere RoboAdmin con troppa facilità.

È comunque necessario usare una password sicura per il login e cambiarla frequentemente. RoboAdmin non può impedire ad un altro utente di autenticarsi al servizio di comunicazione in un altro client con i suoi stessi dati. Se il login avesse successo, un attaccante potrebbe intercettare le informazioni riservate inviate dall'amministratore a RoboAdmin e vanificare ogni meccanismo di protezione adottato dal protocollo.

Le caratteristiche di sicurezza dei protocolli sono state inserite solo negli ultimi anni, e nuove funzionalità vengono aggiunte frequentemente. L'uso di librerie esterne nei bundle di comunicazione è stato preferito proprio per questo motivo: quando una libreria viene aggiornata è possibile sostituirla in RoboAdmin a quella già presente, per avere una nuova implementazione senza modificare radicalmente il bundle. Dove possibile, si è scelto di implementare la libreria che veniva aggiornata più

frequentemente tra quelle disponibili, così da avere sempre a disposizione la versione più recente del protocollo. Durante la stesura di questo testo sono state rilasciate nuove versioni per la maggior parte delle librerie inserite nel progetto. A parte qualche piccola modifica al codice dei bundle, l'aggiornamento è risultato sempre veloce e poco invasivo.

4.1 Sviluppi futuri

In questo paragrafo si ipotizzano alcune caratteristiche in grado di aumentare le funzionalità di RoboAdmin in un prossimo futuro.

4.1.1 Gestione remota dei bundle

Per adesso i bundle di comunicazione di RoboAdmin possono essere avviati soltanto dall'interfaccia locale di Felix. In futuro si potrebbero estendere le funzionalità dell'interprete per fare in modo che l'amministratore riesca a gestire via chat anche l'avvio e lo spegnimento dei bundle che non ritiene più necessari. Una caratteristica di questo tipo risulterebbe molto utile se si individua un tentativo di attacco a RoboAdmin: se l'amministratore in remoto osserva i log dell'applicazione e rileva ripetuti tentativi di accesso da parte di un utente non autorizzato, può provare ad intervenire disconnettendo il bundle dal servizio e avviando un bundle sostitutivo per permettere la sua connessione da un altro servizio di comunicazione.

Aggiungendo nuovi comandi all'interprete, l'amministratore potrebbe anche modificare gli utenti autorizzati nel data base, così da bloccare da remoto un utente non più autorizzato. Questa funzione può essere utile anche se si nota una attività sospetta da parte di un amministratore autorizzato: può capitare che un attaccante si sia infiltrato nel sistema rubando la password di un amministratore, quindi si può tentare di bloccare solo quell'account invece di disconnettere l'intero servizio di chat. I bundle di comunicazione implementati in RoboAdmin sono già predisposti per controllare frequentemente l'identità dell'amministratore, così da bloccarlo immediatamente se nel data base l'autorizzazione viene revocata.

4.1.2 Trasferimento file

Tramite messaggi di testo è possibile eseguire la maggior parte degli applicativi di amministrazione presenti in un sistema operativo. La modifica di file di testo è però gestita da applicazioni interattive, che non possono essere trasmesse da RoboAdmin come normali messaggi istantanei.

Le librerie usate in questa versione di RoboAdmin per i bundle di comunicazione contengono classi e metodi per il trasferimento di file. Funzionalità di questo tipo sarebbero molto utili per un amministratore: se fosse possibile richiedere l'invio di un file a RoboAdmin, l'amministratore lo potrebbe modificare localmente (anche con un editor visuale), per poi rispedirlo modificato a RoboAdmin.

Per abilitare il trasferimento di file non è sufficiente implementare i metodi della librerie sopra citati, ma bisogna aggiungere nuovi comandi all'interprete. Se l'amministratore vuole richiedere un file da RoboAdmin, deve esistere un comando che specifichi il percorso del file che deve essere inviato. Se l'amministratore vuole inviare un file, un apposito comando dovrà essere richiamato in RoboAdmin per predisporlo alla ricezione del file: come parametro del comando si potrebbe specificare la posizione in cui il file dovrà essere salvato. Il comando per la ricezione potrebbe essere richiamato prima o dopo che l'amministratore ha inviato il file: se viene richiamato prima, si ha la certezza che l'invio del file sia avvenuto volontariamente, così da bloccare la ricezione di file non dichiarati esplicitamente.

4.1.3 Invio di più risposte in un unico messaggio

È già stata segnalata nel capitolo 3 la presenza di sistemi anti-spam, spesso adottati lato server da ciascun servizio di messaggistica istantanea. Tali protezioni evitano l'invio di molti messaggi istantanei in rapida sequenza. Per aggirare il problema, l'invio dei messaggi in uscita viene rallentato automaticamente, ma questa procedura ritarda parecchio la ricezione di una risposta lunga inviata da RoboAdmin.

Per diminuire sensibilmente il ritardo si potrebbe pensare di inserire più righe della risposta nello stesso messaggio istantaneo. Le limitazioni possono essere date dal fatto che ciascun servizio di messaggistica istantanea non permette l'invio di messaggi con un numero arbitrario di caratteri.

Bisognerebbe studiare attentamente la dimensione massima del messaggio permessa da ciascun servizio di messaggistica implementato, e inserire in ciascun bundle di comunicazione un sistema di accorpamento dei messaggi che agisca tra l'invio di un messaggio e il successivo. Per fare questo si devono raccogliere il maggior numero di messaggi possibili e inviarli quando si è raggiunta la dimensione massima consentita o quando si è superato il timeout del meccanismo anti-spam.

4.1.4 Installazione dei bundle a runtime

La procedura di installazione del bundle in Felix non è immediata: per permettere il funzionamento del bundle è necessario specificare nel file di configurazione quali package contengono le classi delle librerie usate dal bundle. Il file di configurazione viene letto da Felix soltanto quando il framework viene avviato, pertanto l'amministratore deve prevedere quali bundle devono essere installati prima dell'avvio di RoboAdmin.

Per evitare questo limite si potrebbero inserire in ciascun bundle le informazioni sulle classi che devono essere caricate, e predisporre alcuni metodi in grado di aggiornare a runtime il classpath di Felix, quando si richiede l'installazione del bundle. In questo modo il framework non deve essere riavviato quando si vuole installare un bundle, e non serve modificare il file di configurazione Felix.

Bibliografia

[01] RoboAdmin: http://roboadmin.sourceforge.net

[02] Apache Felix: http://felix.apache.org

[03] OSGi: http://www.osgi.org

[04] MySQL: http://www.mysql.it

[05] NetBeans: http://netbeans.org/

[06] XAMPP: http://www.apachefriends.org/it/xampp.html

[07] PHPMyAdmin: http://www.phpmyadmin.net

[08] IRC: http://en.wikipedia.org/wiki/Internet_Relay_Chat

[09] TLS: http://en.wikipedia.org/wiki/Transport_Layer_Security

[10] PircBot: http://www.jibble.org/pircbot.php

[11] MSNP: http://en.wikipedia.org/wiki/MSNP
http://www.hypothetic.org/docs/msn
http://msnpiki.msnfanatic.com/index.php/Main_Page

[12] JML: http://sourceforge.net/apps/trac/java-jml
http://java-jml.sourceforge.net/javadocs/jml_1.0b3

[13] Skype: http://www.skype.com
http://en.wikipedia.org/wiki/Skype_protocol

[14] Skype4Java: https://developer.skype.com/wiki/Java_API
http://sourceforge.jp/projects/skype/devel

[15] Skype API: https://developer.skype.com/Docs/ApiDoc

[16] XMPP: http://xmpp.org/protocols
http://it.wikipedia.org/wiki/XMPP

[17] Smack: http://www.igniterealtime.org/projects/smack

[18] OSCAR: http://dev.aol.com/aim/oscar
http://en.wikipedia.org/wiki/OSCAR_protocol

[19] AccSDK: http://dev.aol.com/aim/downloads

[20] Bot License: http://dev.aol.com/aim/botLicense.jsp

[21] YMSG: http://en.wikipedia.org/wiki/Yahoo!_Messenger_Protocol
http://www.ycoderscookbook.com

[22] OpenYMSG: http://sourceforge.net/apps/trac/openymsg
http://openymsg.sourceforge.net/javadocs/openymsg_0.3.0

[23] JYMSG9: http://jymsg9.sourceforge.net

[24] Wireshark: http://www.wireshark.org

[25] STUN: http://en.wikipedia.org/wiki/STUN

Altre Fonti

M. Ramilli, M. Prandini. *RoboAdmin: A Different Approach to Remote System Administration*. Proceedings of the 5th International Workshop on Security in Information Systems - WOSIS 2007. Funchal - Madeira, Portugal. 12-16 giugno 2007. (pp. 43 - 52). ISBN: 978-972-8865-96-2.

M. Ramilli, M. Prandini. *A messaging-based system for remote server administration.* Proceedings 2009 Third International Conference on Network and System Security - NSS2009. Gold Coast, Queensland, Australia. 19-21 ottobre 2009. (pp. 262 - 269). ISBN: 978-0-7695-3838-9.

Tesi di laurea di Alessandro Busico: *Progetto e realizzazione di un sistema di comunicazione basato su messaggistica istantanea per l'amministrazione remota di server*, discussa presso Università degli studi di Bologna – Facoltà di Ingegneria in data 18/06/2009.

Tesi di laurea di Lorenzo De Giovanni: *Studio e confronto delle caratteristiche di sicurezza dei sistemi di messaggistica istantanea via Internet*, discussa presso Università degli studi di Bologna – Facoltà di Ingegneria in data 08/10/2008.