Progetto di L.P.R.-Corso A
Laurea Triennale in Informatica
Anno Accademico 2006/2007
prof. Laura Ricci

GOSSIP: Un servizio per lo scambio istantaneo/differito di messaggi su WAN

Marco Cornolti e Alessandro Franchini

1 Specifiche

Il progetto fornisce un servizio di scambio di messaggi tra utenti tramite Internet. Ogni utente, una volta effettuata la registrazione, ha la possibilità di inviare messaggi ad altri utenti. Il servizio supporta anche la possibilità di inviare messaggi ad utenti off-line.

2 Descrizione dell'architettura del sistema

L'architettura del sistema è composta di tre parti:

- un Bootstrap server;
- un insieme di Message Server;
- un insieme di Servlet (gli utenti del servizio).

Queste parti hanno come unico 'contratto' quello definito dalle interfacce RMI contenute nel package common.interfaces. Qualunque implementazione di server che rispetti queste interfacce può entrare a far parte del sistema Gossip. Questo permette anche una ampia libertà nelle implementazioni. Per fare solo un esempio, sarebbe possibile modificare i servlet in modo tale che non comunichino mai direttamente tra loro in modo p2p, ma che utilizzino sempre i server, come in un normale servizio di posta elettronica.

2.1 Message Server

Un sistema Gossip è composto da uno o più Message server (abbreviati MS). Ogni utente alla registrazione viene assegnato ad uno e un solo MS, che usa per scambiarsi messaggi con utenti off-line. In seguito, la gestione di un utente può essere spostata da un Message server ad un altro. Quando un messaggio viene inoltrato da un utente al proprio MS, questo si occupa di recapitarlo al MS del destinatario, in modo che l'utente destinatario, quando si collega, possa richiedere e ricevere la propria coda di messaggi in arrivo. Svolge dunque le funzioni di inoltro dei messaggi che nell'e-mail sono attribuite dei server POP e SMTP.

Inoltre quando un utente si collega, invia al proprio MS l'indirizzo IP e la porta TCP a cui gli altri client si possono collegare per lo scambio diretto di messaggi, in modo da poterli fornire agli altri servlet che ne fanno richiesta, permettendo così il contatto diretto. Questo meccanismo è necessario, essendo l'unico modo che hanno due peer di contattarsi per la prima volta.

Per diminuire più possibile le richieste al Bootstrap, i Message server implementano un sistema di cache, che entra in gioco ad ogni richiesta che riguarda un utente non gestito localmente, ad esempio quando un servlet (*Mitt*) chiede al proprio MS (*MS1*) l'inoltro di un messaggio verso un utente (*Dest*) gestito da un altro MS (*MS2*), oppure quando il Bootstrap chiede lo spostamento di un utente presso un altro MS (*MS2*). Per soddisfare questo tipo di richieste, che costituiscono il principale impiego dei MS, questi devono inoltrare al Bootstrap la richiesta su quale sia il MS a cui l'utente *Dest* è assegnato. Il funzionamento della cache prevede che questa richiesta avvenga una sola volta per ogni utente *Dest*, mentre per le successive viene utilizzata la corrispondenza salvata all'interno della cache tra *Dest* e *MS2*. In questo modo, per ogni utente sconosciuto a *MS1*, deve essere inoltrata una sola richiesta verso il BS, e questo meccanismo alleggerisce il carico sul BS. L'unica altra occasione in cui la cache inoltra una richiesta al Bootstrap è quella in cui, nel caso di inoltro di un messaggio, il presunto MS2 del destinatario risponde che l'utente in realtà non è gestito da lui. Questo significa che l'utente che in passato era gestito da MS2 adesso è stato spostato in un altro MS3, ignoto, per ottenere il quale è necessaria un'altra interrogazione al Bootstrap.

2.2 Bootstrap server

All'interno di un sistema Gossip è presente un solo Bootstrap server. I suoi compiti sono di coordinare il funzionamento dei Message server, gestire la registrazione e la cancellazione degli utenti e assegnare questi utenti ai Message server.

Gli unici tre tipi di richieste che il bootstrap accetta da un servlet sono la registrazione di un utente, la sua cancellazione, e la richiesta da parte dell'utente del Message server a cui egli è assegnato.

Questo non deve far pensare che il funzionamento del bootstrap sia banale. Infatti questa componente di Gossip è quella che tiene insieme il tutto, anche se la sua compromissione non determina il crollo del sistema.

Il Bootstrap si occupa di ricevere i dati con le statistiche di carico da parte dei MS, e in base a questi dati determina il server al quale assegnare i nuovi utenti registrati. In particolare, i Message server dicono al Bootstrap il numero di messaggi inoltrati negli ultimi 3 giorni e la dimensione della memoria libera. In base a questi dati, il Bootstrap decide come assegnare i nuovi utenti.

Fornisce anche all'amministratore del sistema la possibilità di aggiungere e rimuovere MS e utenti, di spostare utenti da un message server ad un altro, di interrogare i MS per elaborare delle statistiche sul funzionamento del sistema e di spegnere i MS a cui non siano assegnati degli utenti.

Come i MS, anche il BS ha un sistema di backup che tiene sempre aggiornato un file, in modo che in caso di crash si possa recuperare lo stato degli utenti e dei MS che compongono il sistema Gossip.

2.3 Servlet

Oltre che fornire un'interfaccia grafica per l'utente finale, il servlet deve gestire le comunicazioni sia con i due tipi di server sia con gli altri servlet.

L'interazione con i due server, come visto sopra, avviene tramite l'utilizzo del protocollo RMI. All'avvio viene creato il riferimento all'oggetto remoto del bootstrap; questo permette di invocare i metodi remoti necessari alla fase di registrazione.

Effettuata la registrazione, la fase di login comprende diversi passaggi: richiesta al Bootstrap del Message server assegnato all'utente, creazione del riferimento all'oggetto remoto del Message server e l'invocazione su di esso dei metodi remoti necessari a settare il proprio stato in on-line e a scaricare i messaggi ricevuti nel periodo in cui era off-line.

Una volta entrati nella finestra principale, l'interazione con i due server è limitata solo alle operazioni di modifica della lista dei contatti e per l'operazione di disconnessione. Questa scelta implica il fatto che tutte le comunicazioni tra utenti del sistema avvengano senza coinvolgere i due server (l'unico caso è l'invio di messaggi ad utenti off-line).

La comunicazione con altri utenti avviene tramite la visualizzazione di una finestra nella quale è possibile scrivere i messaggi da inviare e visualizzare messaggi ricevuti e inviati. Le comunicazioni sono effettuate tramite socket TCP per favorire la corretta ricezione dei dati. Ogni tipo di eccezione (chiusura socket,..) viene gestita nella maniera opportuna.

Ogni servlet dispone di un ServerSocket in ascolto su una porta; questo riceve i messaggi e a seconda del testo del messaggio compie l'operazione associata.

Data la struttura del sistema un'operazione da gestire in maniera appropriata è il contatto di un altro utente. Infatti una volta contattato il Message server per richiedere le informazioni sull'utente, tutto avviene a livello P2P. Questo comporta la gestione in maniera corretta di eventi successivi al contatto, quali bloccaggio e sbloccaggio di un utente quando non si vogliono più ricevere messaggi da questo utente. Infatti per es. nel caso di bloccaggio di un utente c'è una comunicazione di cambio di stato in off-line e dall'altra parte il settaggio di Ip e porta a null.

3 Metodo di lavoro e di sviluppo.

Per lo sviluppo abbiamo seguito un metodo di progettazione che trae ispirazione dall'Extreme programming e si avvicina molto all'Unified Process, in cui le varie fasi di sviluppo (analisi, progettazione, implementazione, testing) sono da seguire iterativamente per diverse volte. Tuttavia l'Unified Process è un modello di sviluppo per grandi progetti, ed essendo noi soltanto in due persone abbiamo snellito il processo cercando di seguirlo soltanto là dove ci avrebbe effettivamente

aiutato. Ad esempio, le iterazioni non sono state 10 come indicato nell'UP ma solo 2, non molto rigorose, e alcuni grafi previsti dall'UP non sono stati realizzati.

Nella prima fase di analisi abbiamo fatto la raccolta delle caratteristiche che le componenti del sistema dovevano fornire, e come doveva avvenire la loro interazione. Di conseguenza abbiamo definito quali fossero i canali di comunicazione (dove usare RMI e dove altri protocolli) e di conseguenza abbiamo definito le interfacce RMI tra i diversi componenti.

Per avere un'idea comune sul funzionamento delle procedure che riguardano più di una componente, sono stati disegnati col programma StarUML dei grafi UML di sequenza, di analisi dei requisiti e di deployment. La creazione di questi grafi ha richiesto molto tempo e ragionamento, ma ha permesso di risolvere in anticipo molti problemi e ha costituito un riferimento utilissimo nella fase di implementazione.

Per la progettazione e l'implementazione ci siamo divisi il lavoro in modo che ognuno potesse concentrarsi sulla propria parte e lavorarci indipendentemente dall'altro. Ad Alessandro Franchini è stato assegnata la programmazione dei servlet mentre a Marco Cornolti quella dei server Bootstrap e Message.

Nella fase di progettazione ognuno ha definito le classi Java necessarie per il funzionamento dei propri componenti. Sono stati definiti quattro package: il `bootstrapP`, il `messageServerP`, il `servlet`, il cui scopo è ovvio, ed i package `common`, `common.interfaces` e `common.interfaces.exceptions`. che hanno la proprietà di essere utilizzate da più di un package, ad esempio la classe `common.User` rappresenta un utente, e la classe viene utilizzata sia dai servlet, che la usano per rappresentare gli utenti contattati, sia dal Message server, che la estende con `ManagedUser` per rappresentare un utente gestito dal Message server.

La fase di implementazione è avvenuta indipendentemente per ogni componente del progetto, e durante questa fase in diverse occasioni sono stati corrette alcune decisioni assunte nella fase di progettazione. Tuttavia nella fase di implementazione molte scelte sono state rivisitate e non sempre i grafi sono stati aggiornati.

Il testing è avvenuto con delle classi scritte appositamente per provare tutte le possibili modalità di comunicazione e invocazione dei metodi.

4 Tipi di comunicazione

In base alle proprietà dei protocolli di comunicazione e alla facilità di utilizzo sono stati scelti i modi in cui le varie componenti comunicano tra loro.

Per la comunicazione diretta tra servlet è stato scelto il protocollo TCP, che garantisce un basso overhead e un facile invio dei messaggi. Per l'interazione tra servlet e i due server, e tra Message server e Bootstrap, è stato scelto RMI, che aiuta la programmazione tramite la definizione delle interfacce RMI. In fine, il protocollo UDP è stato scelto per le notifiche periodiche che i Message server spediscono al Bootstrap per comunicare il proprio stato di carico. Questo

protocollo è stato scelto perché annulla l'overhead, snellendo il traffico verso il bootstrap. Anche se i pacchetti UDP possono perdersi, queste perdite non pregiudicano il funzionamento del sistema, dato che le informazioni di questo tipo sono sono necessarie e comunque vengono inviate periodicamente sempre aggiornate.

5 Note sulla scalabilità

Un'attenzione particolare è stata dedicata alla possibilità di rendere scalabile il sistema rispetto al numero di utenti e di messaggi trasferiti. A condizioni normali gli utenti, tramite i servlet, comunicano tra loro tramite p2p, per cui la maggior parte del traffico non passa tramite il sistema dei server e non li appesantisce, ma ricade unicamente sui peer. Gli unici messaggi che passano ai server sono quelli spediti a utenti offline, che vengono inoltrati tra i Message Server. Se vi fosse un solo Message server, questo diverrebbe un collo di bottiglia all'aumento del numero di utenti a causa dei messaggi scambiati off-line, potenzialmente in grande numero. Per questo il sistema prevede la possibilità dell'attivazione di altri Message server, che dovrebbero essere in numero proporzionale agli utenti. Questo permette l'espansione del sistema ad un numero altissimo di utenti. Il Bootstrap, che è l'unico elemento non riproducibile del sistema, è snellito al massimo, e non è coinvolto nel traffico dei messaggi, ma si occupa unicamente di stabilire una corrispondenza tra gli utenti ed i Message server che li gestiscono. Questi due tipi di richieste non costituiscono un traffico di dimensioni elevate e sono estremamente ridotte grazie ad un sistema di cache dei Message server. Questa componente, essendo non riproducibile, è l'unica a rappresentare una limitazione al numero di utenti, sebbene questo numero sia altissimo. Questa limitazione si può risolvere solo modificando la struttura del sistema e rendendo riproducibile anche il Bootstrap, oppure delegando ai Message server il coordinamento tra loro, creando un sistema che non abbia bisogno di Bootstrap.

6 Descrizione threads attivati e strutture dati utilizzate

6.1 Message server

L'invocazione dei metodi remoti del Message server avviene col multithread implicito di RMI, implementato in quasi tutte le virtual machine di Java, nelle quali le richieste non vengono processate sequenzialmente ma contemporaneamente.

Inoltre il Message server attiva due thread:

- `LoadThread`, che periodicamente invia tramite UDP le statistiche di carico al bootstrap;
- `HumanListener`, che resta in attesa di comandi provenienti dall'interfaccia dell'amministratore umano, ed esegue le azioni richieste.

Le strutture dati utilizzate di particolare rilevanza sono contenute nelle classi `Cache` e `ManagedUserDatabase`.

- La classe `Cache` si occupa di mantenere una corrispondenza tra gli utenti e i Message server che li gestiscono. Ovviamente contiene solo gli utenti gestiti dagli altri Message server. Per questo utilizza tre vettori:

  1. `Vector <SafeMSForMSInterface> safeMSInterfs`;
  2. `Vector <MessageServer> messageServers`;
  3. `Vector<List<User>> nonManagedUsers`.

  Queste classi sono gestite in modo da avere sempre dimensione uguale tra loro, e tale che, data una posizione *i* di ogni vettore, l'elemento `messageServers.get(i)` restituisca uno dei message server, `safeMSInterfs.get(i)` restituisce la sua interfaccia tramite cui invocare i metodi RMI, e `nonManagedUsers.get(i)` restituisce la lista di utenti gestiti dal Message server. Le liste degli utenti contenute in `nonManagedUsers` sono ordinate alfabeticamente, in modo tale da potervi effettuare una ricerca binaria. Grazie a ciò, la classe `Cache` è in grado di restituire il Message server a cui un dato utente è assegnato in $O(m \cdot \log(n/m))$ dove n è il numero di utenti e m il numero di Message server presenti nel sistema.
- La classe `ManagedUserDatabase` ha invece il compito di mantenere la lista degli utenti gestiti dal Message server locale. Anche in questo caso le liste di utenti sono sempre ordinate, in modo che la ricerca di un utente nel database possa avvenire in $O(\log(n))$ dove n è il numero di utenti contenuti.

6.2 Bootstrap server

Riguardo ai metodi RMI del Bootstrap server vale quanto detto per il Message server.

Il bootstrap attiva anche i due thread:

- `LoadListener`, duale del LoadThread del Message server, che riceve e processa le informazioni di carico dai pacchetti UDP;
- `HumanListener`, equivalente all'omonimo thread del Message server.

Le strutture dati per il Bootstrap server sono rappresentate dalle classi `ManagedMassageServerDatabase`, che contiene l'insieme dei Message server presenti nel sistema, e `UsersDatabase`, che contiene l'elenco degli utenti registrati.

La classe `ManagedMassageServerDatabase` contiene la lista `List<ManagedMessageServer> vector` in cui sono archiviati i Message server ordinati secondo il loro carico. La prima posizione è occupata dal Message server meno carico, l'accesso a questo elemento richiede dunque un tempo costante indipendente dal numero di Message server, mentre l'ordinamento, che avviene ogni volta che arriva una nuova segnalazione di carico da parte dei message server, richiede $O(n \cdot \log(n))$, dove n è il numero di Message server contenuti nella lista.

La classe `UsersDatabase` contiene invece la `List<RegisteredUser> users`, che contiene tutti gli utenti registrati nel sistema. Uno dei campi degli elementi `RegisteredUser` è il Message server che lo gestisce. Dato che questi elementi sono ordinati alfabeticamente per il nome degli utenti, per ottenere una corrispondenza tra utente registrato e il Message server a cui è assegnato, è sufficiente un tempo di $O(\log(n))$ dove n è il numero di utenti.

6.3 Servlet

I thread attivati da ogni Servlet sono:

- `clientListener`, thread utilizzato per stare in ascolto su un Server Socket; ogni qual volta riceve un messaggio, analizza la sua intestazione e compie le dovute operazioni;
- `controlStatus`, insieme di thread (uno per ogni utente della lista contatti) che dopo l'accesso controlla quali utenti sono on-line e quali no. I thread accedono ad un'unica lista contatti prelevando e rimuovendo il primo elemento della list ed è quindi necessario sincronizzare i metodi di accesso alla lista.
- `peerCommunication`, thread che gestisce lo scambio di messaggi con un singolo utente; quindi saranno attivati tanti thread quante sono le comunicazioni che l'utente ha intenzione di fare.

I servlet utilizzato diverse strutture dati:

- `contactList`, struttura utilizzata per mantenere la lista dei contatti;
- `conn`, vettore utilizzato per gestire le comunicazioni (informazioni su socket e interfaccia grafica associata) aperte dall'utente;
- vengono inoltre utilizzati vettori ausiliari, per la memorizzazione temporanea dei dati scaricati all'avvio dal message server; per es. i messaggi ricevuti dall'utente nel periodo in cui è stato off-line.

6.4 Common

Nel package `Common`, l'unica struttura dati particolarmente interessante è quella della classe `Statistics`. Questa classe contiene un vettore statico `Vector<Date> forwardedMessagesDates`, che contiene le date in cui sono avvenuti gli inoltri dei messaggi ad opera di un Message server. Questo permette di fare delle statistiche sul numero di messaggi inoltrati recentemente da un Message server, ad esempio si può chiedere quanti messaggi siano stati inoltrati negli ultimi 3 giorni. La classe mantiene traccia dei messaggi solo dell'ultima settimana. Col metodo `increaseForwardedMessages()` una nuova data viene aggiunta al vettore, e vengono cancellate quelle più vecchie di una settimana. Il fatto che le date siano ordinate nel vettore, permette al metodo `forwardedMessagesSince(int min)` di trovare il numero di messaggi inviati negli ultimi min minuti in tempo $O(\log(n))$, dove n è il numero di date contenute nel vettore, tramite un algoritmo di ricerca binaria modificata per lo scopo.

7 Descrizione delle classi definite

7.1 package Common

Il package `common` contiene delle classi di utilità da cui dipendono diversi componenti del sistema.

7.1.1 `Bootstrap` definisce una rappresentazione del Bootstrap server.

7.1.2 `ContactList` definisce una struttura dati per la lista dei contatti. Permette di aggiungere, rimuovere ed ottenere un utente; dispone anche di metodi utili al settaggio di parametri relativi ad un utente.

7.1.3 `Message` definisce la struttura dei messaggi scambiati dagli utenti. Ogni message contiene testo del messaggio, mittente, destinatario e data dell'invio del messaggio.

7.1.4 `MessageServer` questa classe rappresenta un message server.

7.1.5 `Statistics` permette di ricevere informazioni su statistiche relative ai message server(ad es. memoria libera, messaggi inoltrati,..)

7.1.6 `User` definisce una rappresentazione di un utente.

7.2 package Common.Interfaces

Queste classi definiscono interfacce le quali contengono metodi remoti necessari per la comunicazione sia tra due server che tra un server ed un Servlet. Si distinguono due tipi di interfacce: quelle offerte dal bootstrap e quelle offerte dai Message server.

Il nome di ogni interfaccia è nella forma `XforYInterface`. Questo significa che è l'interfaccia fornita da X per Y. Ad esempio, l'interfaccia `MSForBSInterface` definisce i metodi che i Message server devono implementare per permetterne l'utilizzo ai Bootstrap.

7.3 package Common.interfaces.Exception

Queste classi definiscono un insieme di eccezioni lanciate dai metodi invocati tramite le interfacce.

7.4 package servlet

Il package `servlet` contiene le classi utili allo sviluppo della parte che interagisce con l'utente finale. È composto di 12 classi qui sotto elencate e descritte:

7.4.1 AccessOperation

Questa classe fornisce metodi per l'interazione con il bootstrap e il message server (ad es. ricerca dell'oggetto remoto); dispone inoltre di un metodo che consente di contattare un utente la prima volta.

7.4.2 ClientCore

Questa è la classe che viene lanciata all'avvio; dispone di diverse variabili pubbliche utili in tutto il package (ad es. utente, bootstrap,..). Effettua i controlli iniziali e poi lancia un'interfaccia grafica.

7.4.3 ClientListener

Viene utilizzata per rimanere in ascolto su un Server Socket; estende la classe thread e quindi è in continua funzione qualsiasi cosa stia facendo l'utente. Ogni qual volta viene ricevuto un messaggio, verifica il tipo e svolge le dovute operazioni. Nel caso di messaggi ricevuti da utenti da lui bloccati, scarta il messaggio.

7.4.4 ConfirmInterface

Questa interfaccia grafica viene visualizzata quando un utente riceve una richiesta di amicizia. L'utente deve scegliere l'opzione di risposta e premere il bottone di conferma; in base alla risposta data, vengono modificate opportunamente la lista dei contatti e le liste associate alle interfacce grafiche.

7.4.5 Connection

Questa classe mantiene informazioni sulle conversazioni aperte dall'utente; in particolare tiene traccia dell'utente con cui sta comunicando, del socket utilizzato e dell'interfaccia grafica associata alla comunicazione.

7.4.6 ContattInterface

Questa interfaccia grafica serve per cambiare lo stato di amicizia nei confronti di un utente e per aggiungere un nuovo utente alla propria lista dei contatti. L'interazione con l'utente avviene tramite la selezione del contatto da spostare e la pressione del bottone di conferma.

7.4.7 ControlStatus

Questa classe è utilizzata per il controllo sullo stato dei propri contatti. La classe fornisce metodi per l'accesso alla lista contatti e l'aggiunta del contatto modificato ad una nuova contact list. A questa classe è associata la classe threadStatus: questa preleva il primo utente della lista e ne controlla lo stato; finita l'operazione lo aggiunge alla nuova lista.

7.4.8 GossipInterface

Questa interfaccia grafica dispone di due costruttori: uno lanciato all'avvio, il quale visualizza il logo del sistema e il bottone di accesso alla finestra di login; l'altro invece visualizza la finestra principale di ogni utente, contenente la lista dei contatti on-line e off-line. All'avvio di questo costruttore viene lanciato il thread predisposto alla ricezione continua di messaggi (`ClientListener`).

7.4.9 LoginInterface

Questa interfaccia grafica è utilizzata dall'utente per la fase di registrazione e login. Una volta che l'utente inserisce username e password e preme il bottone di conferma, vengono richiamati i metodi remoti di uno dei due server (bootstrap nel caso della registrazione; message nel caso del login).

7.4.10 MailInterface

Questa interfaccia grafica visualizza, una volta effettuato l'accesso, i messaggi ricevuti nel periodo off-line.

7.4.11 PeerCommunication

Questa classe gestisce le operazioni di comunicazione con un altro utente. Dispone di un metodo per l'invio di un messaggio e del metodo `run()` (la classe estende thread) per ricevere messaggi e stampare il testo del messaggio ricevuto nella relativa interfaccia grafica.

7.4.12 PeerInterface

Questa interfaccia grafica, visualizza la finestra di comunicazione tra due utenti. Una volta che l'utente inserisce il testo da inviare e preme il bottone di conferma, viene inoltrato il messaggio e stampato ciò che lui ha scritto.

7.5 Classi simili nei package messageServerP e bootstrapP

1.1. I due server hanno una struttura delle classi simile tra loro, dato che molte delle azioni a cui sono preposti sono simili e molte classi hanno nomi uguali anche se ovviamente eseguono operazioni diverse. Le classi analoghe sono `BackupperInterface`, `FileBackupper`, `HumanInterface`, `TextualInterfaceIt`, `Command`, `HumanListener`, `Communication`, `SafeMSForMSInterface`, `SafeMSForBSInterface`, `MessageServerCore` e `BootstrapCore`.

7.5.1 `BackupperInterface` definisce un'entità di backup, dando una descrizione generale del servizio. E' possibile pensare a una unità di backup basata su un unico file, magari compresso, oppure basata su file e directory, basata su un database SQL, o remota. Per il Message server l'unità di backup deve poter salvare e recuperare lo stato degli utenti, mentre quella del Bootstrap deve poter salvare anche i Message server.

7.5.2 `FileBackupper` è un'implementazione molto semplice di `BackupperInterface`. Salva in un unico file gli oggetti utilizzando gli `ObjectOutputStream` e li legge con la funzione inversa. Non bada a spazio utilizzato né al tempo necessario per la scrittura e ripristino dei dati.

7.5.3 `HumanInterface` è l'interfaccia che stabilisce le proprietà dell'interfaccia con l'amministratore umano. Possibili implementazioni di questa interfaccia sono le interfacce testuali, grafiche o sonore nelle varie lingue.

7.5.4 `TextualInterfaceIt` è una semplice implementazione dell'interfaccia umana, in italiano e basata unicamente sul testo. Tramite testo vengono raccolti i comandi dall'amministratore e tramite testo vengono stampate le informazioni di output del server.

7.5.5 `Command` rappresenta un comando fornito dall'utente tramite l'interfaccia umana. Può essere di vari tipi e contenere degli argomenti.

7.5.6 `HumanListener` è il thread che resta in attesa di comandi provenienti dall'interfaccia umana corrente ed esegue le azioni opportune.

7.5.7 `Communication` è la classe principale che contiene i metodi veri e propri di comunicazione con gli altri server e con i servlet. E' un oggetto remoto che viene pubblicato in modo che gli altri ne abbiano accesso e possano interrogare il server.

7.5.8 `SafeMSForMSInterface` e `SafeMSForBSInterface` sono implementazioni delle interfacce di comunicazione tra server, e chiamano tutti i metodi relativi, con la particolarità che nel caso che i metodi falliscano, provano per una volta a ristabilire la comunicazione con il server. Questo meccanismo è necessario quando il riferimento ad un server remoto viene perso, ad esempio a causa di un crash, ed il server è stato riattivato. In tal caso, invocando il metodo su una normale interfaccia verrebbe lanciata la `RemoteException`, mentre rinnovando il riferimento all'interfaccia tutto funziona.

7.5.9 `MessageServerCore` e `BootstrapCore` contengono il metodo `main` dei due server, col quale è possibile attivare tutti i servizi.

7.6 Il Message server

Oltre alle classi simili tra MS e BS, il Message server è composto da altre classi, alcune di entità e altre di controllo. Si evita di dare una nuova descrizione delle classi thread e di strutture dati già descritte nel paragrafo 6.

7.6.1 `ManagedUser` è la classe entity che rappresenta un utente gestito localmente.

7.6.2 `MyselfMessageServer` è la classe entity che contiene una rappresentazione nel Message server di se stesso (ad esempio, contiene la password di gestione).

7.6.3 `Cache` che realizza il sistema sopra descritto di caching delle corrispondenze tra utenti non gestiti localmente e Message server a cui questi sono assegnati.

7.7 Il Bootstrap

7.7.1 `ManagedMessageServer` è una classe entity che rappresenta un Message server parte del sistema Gossip.

7.7.2 RegisteredUser è un'altra classe entity che rappresenta un utente registrato nel sistema.

8 Note sulla validità degli stati e la robustezza del sistema.

Il sistema è stato pensato per essere robusto e sopportare, limitando i danni, l'improvviso shutdown di uno dei server o dei servlet.

Grazie al sistema di backup, lo stato dei server viene, ad ogni modifica, salvato. Questo permette ai server, in caso di crash, di recuperare lo stato e riprendere immediatamente il normale funzionamento.

L'unica possibilità perché la correttezza dello stato sia compromessa, è che un message server si spenga e, ad esempio per la rottura del disco di backup, perda tutti i dati relativi agli utenti. In questo caso il Bootstrap continuerà a dirigere sul message server compromesso le richieste per gli utenti, ma il Message server risponderà che questi utenti non sono gestiti da lui. In casi simili si potrebbe pensare ad un meccanismo per cui vi possa essere una nuova assegnazione di un utente ad un altro Message server. Questo meccanismo tuttavia non è implementato.

Vediamo invece cosa succede nel periodo di tempo in cui alcuni server o servlet hanno avuto un crash e non sono stati riattivati. Per cominciare, possiamo analizzare il caso in cui muore il Bootstrap server. In questa circostanza, viene completamente interrotta la registrazione di nuovi utenti, ma il resto del sistema continua a funzionare quasi completamente. Infatti i servlet che hanno stabilito un contatto diretto tra loro continuano a comunicare, e grazie al meccanismo di cache dei message server, che non hanno bisogno di interrogare ogni volta il Bootstrap per scoprire a quale Message server sia assegnato un utente, gli utenti possono continuare a spedire messaggi off-line. I problemi sorgono quando un Message server riceve la richiesta di inoltro di un messaggio verso un utente non presente nella cache e non gestito da lui. In tal caso è necessaria l'interrogazione del Bootstrap, e il Message server risponderà con un errore. Appena il Bootstrap server verrà riattivato, grazie al sistema di backup, tutto tornerà a funzionare.

Il secondo caso, in cui muore un Message server, è forse il più grave. Mentre le comunicazioni dirette tra servlet continuano a funzionare, l'invio di messaggi off-line da parte di un servlet al proprio Message server (MS1) fallirà, e non funzionerà neanche la richiesta da parte di un utente di ottenere il contatto diretto per un altro utente. Tuttavia il servlet potrà chiedere al Bootstrap il Message server (MS2) dell'utente da contattare, e a meno che il message server del mittente e del destinatario coincidano (e quindi che MS1=MS2 sia morto), potrà chiedere a lui il contatto diretto dell'utente oppure inoltrare comunque il messaggio off-line. Questa pratica ha come lato negativo l'appesantimento del Bootstrap.

Il terzo caso, in cui muore un Servlet, ovviamente non influenza il funzionamento dei server, mentre gli altri utenti si renderanno conto che il Servlet è morto e inoltreranno i messaggi al Message server invece che mandarli direttamente.

9 Note sui bug e i problemi di sicurezza

Questo sistema Gossip non è del tutto sicuro. Tutte le comunicazioni avvengono non crittate e le password vengono spedite in chiaro, salvate in chiaro nei database e visualizzate sulle interfacce dei server. Tuttavia questo avviene anche per molti sistemi commerciali di instant messaging, tra cui ICQ. Per risolvere questi problemi si dovrebbe utilizzare dei protocolli più sicuri per le comunicazioni dirette tra Servlet e per l'invocazione di metodi RMI, ad esempio SSL, oppure il crittaggio del contenuto dei messaggi, ad esempio con GPG.

Inoltre, per come è pensato lo scambio di messaggi, è molto facile produrre un messaggio con mittente falso. E' infatti sufficiente forgiare un messaggio con un utente qualsiasi e inviarlo al Message server del destinatario utilizzando il metodo forwardMessage(Message msg) e il message server lo inoltrerà senza verificarne la provenienza. Una possibile soluzione a questo problema sarebbe di permettere il forward di un messaggio ad un Message server solo se questo si è autenticato con password, in modo da poterne verificare la provenienza, e far sì che i message server destinatari accettino solo messaggi provenienti da utenti gestiti o da altri message server. In questo modo sarebbe impossibile introdurre un messaggio falso, tuttavia i Message server dovrebbero conoscere lo stato dell'intero sistema e l'identità degli altri Message server, che dovrebbero autenticarsi a loro volta. Anche in questo caso, la magra consolazione è che anche i normali sistemi di e-mail soffrono di questo difetto, e ciò nonostante sono utilizzati in tutto il mondo.

Marco Cornolti e Alessandro Franchini,

Pisa, 22 luglio 2007.