UNIVERSITÀ DI PADOVA **FACOLTÀ DI INGEGNERIA**

TESI DI LAUREA

STUDIO E SVILUPPO DI UN'INTERFACCIA DI COMUNICAZIONE CON LA PIATTAFORMA DELL'OPERATORE TELEFONICO PER L'EROGAZIONE DI SERVIZI VIA SMS

Study and development of a communication interface with the platform of the telephone company to deliver services via SMS

Laureando: Stefano Mandruzzato

Relatore: Federico Filira

Corso di Laurea Triennale in Ingegneria Informatica

24 Febbraio 2011

Anno Accademico 2010-2011

Sommario

L'obiettivo del tirocinio svolto c/o **ne-t by Telerete nordest** é quello di trattare lo studio lo sviluppo di un interfaccia di comunicazione con la piattaforma di un operatore telefonico prestabilito (nel nostro caso Vodafone) per l'erogazione di servizi via SMS.
Esso consiste quindi di creare un programma che permetta di comunicare con il server di messaggistica aziendale mobile di Vodafone per eseguire tutti i comandi principali per l'invio e la ricezione di SMS e successivamente implementarlo nei servizi che ne fanno uso.
Principalmente i servizi via SMS sviluppati in questa tesi saranno due:

- l'acquisto di biglietti di un'azienda di trasporto pubblico tramite SMS per mezzo di un'applicativo per cellulari

- una piattaforma web per mandare SMS a piu' contatti con la possibilitá di creare e gestire campagne promozionali.

Il capitolo 1 ha lo scopo di introdurre i due progetti presi a carico nella tesi al fine di sviluppare lo scopo e il contenuto di essi.
Il secondo capitolo invece spiegherá al meglio le principali funzionalitá del server MAM di Vodafone, per analizzare a fondo come verrá effettuata l'implementazione in java dell'interfaccia di comunicazione spiegata nel capitolo 3.
Nel quarto capitolo invece affronteremo lo schema del progetto dell'interfaccia web creata per la gestione delle campagne promozionali e tutte le classi java utilizzate a tal proposito.
Nel quinto capitolo anilizzeremo il database creato ad hoc per gestire i dati necessari che vengono utilizzati nel nostro progetto, le varie relazioni tra le tabelle e le loro funzioni.
Nel sesto invece affronteremo la modalità utilizzata per creare le pagine web, il linguaggio utilizzato e le sue funzionalità.
Nel settimo capitolo analizzaremo nel dettaglio lo scopo dell'utlizzo del Javascript, i vari controlli dei form delle pagine e la sua sintassi.
Nel capitolo otto parleremo dell'utilizzo delle Sessioni per rendere il progetto in modalità multi utente, e nell'ultimo parleremo dello stile grafico utilizzato e l'importanza dell'utilizzo dei fogli di stile in un sito web.

Ringraziamenti

Desidero ringraziare il prof. Filira, relatore di questa tesi, per la grande disponibilità e cortesia dimostratemi, e per tutto l'aiuto fornito durante la stesura.
Un sentito ringraziamento ai miei genitori e ai miei fratelli, che, con il loro incrollabile sostegno morale ed economico, mi hanno permesso di raggiungere questo traguardo.
Desidero inoltre ringraziare i colleghi di Telerete, per quanto hanno fatto per me durante il periodo di stage.
Ringrazio inoltre ai compagni di studi: Ida, Chiara, Lea, Vale, Matteo, Loris, Libra e Alberto; per essermi stati vicini nei lunghissimi periodi di studio. Sono stati per me grandi amici oltre che semplici compagni.
Ringrazio con affetto Nicola, per avermi dato una mano negli ultimi esami, sono stati quelli più tosti ma più interessanti.
Un'ultimo ringraziamento va ai miei "fratelli" Andre, Ale e Roby per aver condiviso con me la passione della musica, che ci ha portato momenti di gioia e fatica, ma soprattuto grandi soddisfazioni.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Acquisto di biglietti APS tramite SMS

Al giorno d'oggi l'acquisto di biglietti per un servizio di trasporto urbano può avvenire in molti modi:

- tramite un'edicola o in qualsiasi tabaccheria.
- per mezzo dell'autista, pagando in molti casi una sovrattassa in relazione al prezzo del biglietto.
- tramite le macchinette automatiche presenti in qualche fermata dell'autobus, ma non in tutte.

Capita spesso a molte persone di prendere un autobus al di fuori dell'orario d'ufficio, trovando quindi tutti negozi adibiti alla vendita di biglietti chiusi; o magari di trovarsi in una fermata dove non sono disponibili le macchinette automatiche.
Pagare una sovrattassa per aver avuto la consapevolezza di voler acquistare il biglietto ma di non aver avuto la possibilità prima dell'utilizzo del mezzo pubblico non sembra molto opportuno.
L'opportunità di acquistare il biglietto attraverso il cellulare (mezzo di comunicazione di massa) può evitare di pagare una sovrattassa inutile o semplicemente evitare di rischiare di prendere un mezzo pubblico senza aver il biglietto.

Questo servizio quindi necessita di una piattaforma per l'acquisto di titoli di viaggio tramite SMS per il trasporto pubblico di Padova in dotazione ad APS Holding.
L'importo del biglietto viene accreditato, attraverso il circuito Movincom, direttamente sulla carta di credito dell'intestatario della SIM da cui è stato inviato l'SMS. In particolare viene utilizzato l'HUB Movinbox, ovvero di una piattaforma che permette alle aziende consorziate a Movincom di addebitare pagamenti disposti tramite telefono cellulare. Questo tipo servizio è offerto da Bemoov®.

1.1.1 Attori

Gli attori in gioco in questo servizio sono:

- **Cliente**: Cliente che richiede l'acquisto di titoli del biglietto APS.
- **SMS Gateway**: Dispositivo identificato da un numero di cellulare in grado di:
  - spedire/ricevere SMS.
  - tradurre gli SMS in un formato consultabile via web.
- **ne-t Application**: Applicativo che ha il compito di comunicare in maniera biunivoca con l'SMS Gateway, con l'Hub Movincom e con il database per titoli di viaggio di proprietà dell'APS.
- **Hub Movincom**: Server in grado di interrogare gli operatori bancari ed effettuare gli accreditamenti direttamente sui conti correnti degli intestatari del numero di cellulare da cui viene spedito l'SMS di richiesta di acquisto.
- **Hub Movincom**: software gestionale in dotazione all'azienda di trasporti ASP Holding.
- **IVR di ne-t by Telerete**: Viene interrogato ogni qual volta ci sia la necessità da parte del cliente di effettuare una nuova richiesta di invio SMS quando il precedente non sia stato ricevuto
- **Personale ispettivo**: Personale a bordo del mezzo predisposto alla verifica della validazione dei titoli di viaggio.
- **Strumento di backoffice per l'azienda di trasporti**: Strumento via WEB che permette al personale dell'APS Holding di visualizzare le transazioni effettuate ed il loro stato.
- **Help Desk di primo livello**: Help desk tecnico di primo livello per i tecnici Movincom.

1.1.2 Flusso di esecuzione del servizio

Analizziamo nel dettaglio il flusso completo per l'acquisto di un biglietto per un singolo viaggio.

1. Il cliente invia un SMS opportunamente formattato ad un numero che fa capo ad un server SMS Gateway. Per farlo utilizza un widget. L'SMS viene quindi tradotto in una forma consultabile via web. Tale SMS conterrà al suo interno tra le varie informazioni la tipologia del biglietto oggetto dell'acquisto con la relativa quantità.
2. Le nuove richieste vengono passate dalla ne-t Application all'Hub MovinBox.

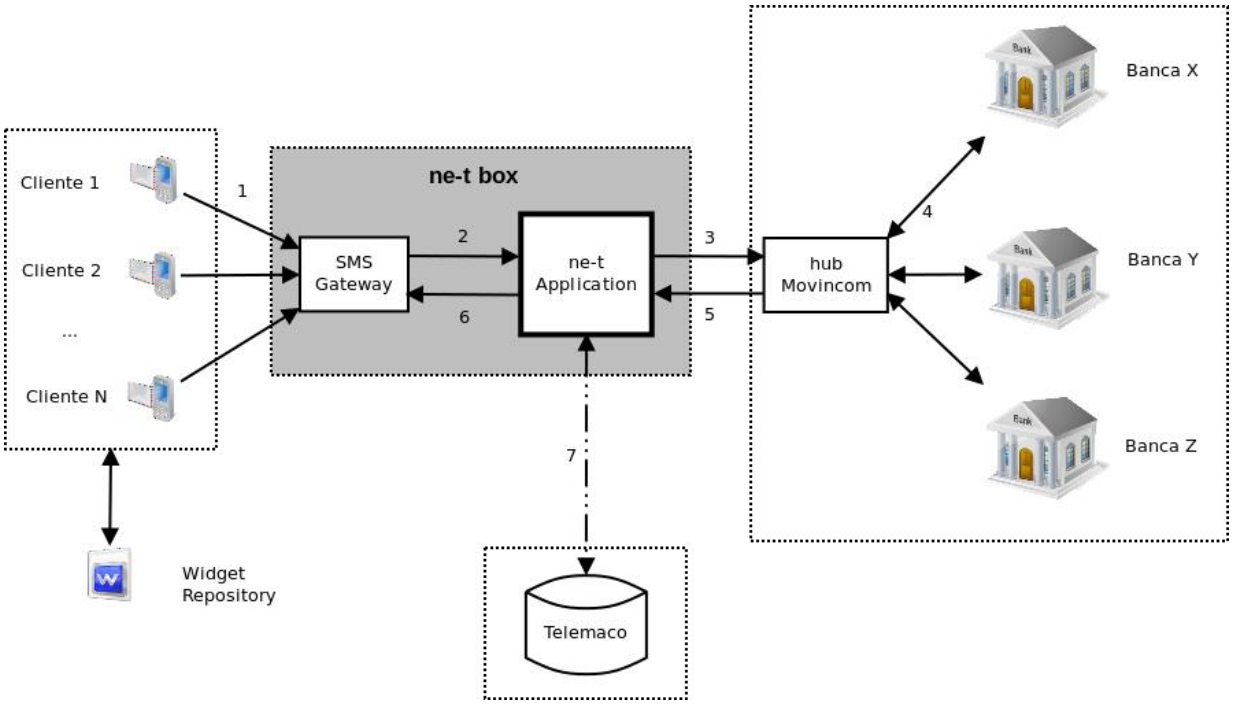

Figura 1.1: *flusso esecuzione per l'acquisto di un biglietto*

3. La ne-t Application, tramite l' Hub Movincom, effettua una richiesta di autorizzazione al pagamento all'operatore bancario presso cui il cliente ha fatto l'accoppiamento numero di cellulare/strumento di pagamento. Nel caso l'autorizzazione abbia esito positivo la ne-t Application avvia il processo di contabilizzazione che comporta l'invio da parte dell'operatore bancario di un SMS di avvenuto pagamento.

4. Viene comunicato l'esito del pagamento dall' Hub Movinbox alla ne-t Application .

5. Viene inviato dalla ne-t Application all'SMS Gateway una richiesta di inoltro SMS al cliente che ha eseguito l'acquisto. Tale SMS contiene un codice necessario alla stampa delle ricevuta o per la richiesta di fattura.

6. Viene contattato il gestionale dell'Azienda di Trasporti al fine di inserire i dati relativi all'oggetto acquistato. E' quindi ritornato un codice identificativo che sarà quello riportato nella ricevuta stampabile dal cliente.

In caso di avvenuto pagamento il cliente riceverà due SMS: uno dalla ne-t Application e uno dall'emittente dello strumento di pagamento.

Il cliente riceverà un SMS dalla ne-t Application nei casi in cui si verifica un errore nella procedura; quindi anche nel caso l'autorizzazione abbia esito positivo ma non la contabilizzazione.

Verifica dell'avvenuta validazione Per la validazione da parte del personale ispettivo è sufficiente presentare l'SMS di ritorno dalla ne-t application riportante l'ora di acquisto del biglietto (che corrisponde a quella di invio da parte della ne-t Application).
In alternativa il controllore può richiedere di contattare un servizio IVR in cui una volta inserito il numero di cellulare del cliente vengono ritornati gli ultimi acquisti effettuati.
A disposizione del personale ispettivo c'è anche un file di testo che viene caricato nel loro palmare tramite la connessione WiFi presente negli autobus e che riporta tutti gli acquisti effettuati durante la giornata.

1.1.3 studio del progetto

La nostra applicazione farà parte del dispositivo **SMS Gateway**, e sarà in grado di comunicare al server di messaggistica aziendale mobile dell'operatore prescelto per l'invio di SMS e la gestione di report di messaggi inviati e ricevuti. Possiamo notare come il nostro lavoro è solo una minima parte del progetto ma di estrema importanza in quanto grazie a questa, l'intero progetto potrà sfruttare il mezzo di comunicazione scelto per ottenere il determinato scopo: l'SMS.

1.2 Creazione di interfaccia web di gestione di invio SMS e di campagne pubblicitarie

1.2.1 Scopo del progetto

Lo scopo di questo progetto è quello di creare un interfaccia web che riesca ad usufruire in maniera ottimale nella connessione al **server MAM** (Messaggistica Aziendale Mobile) della Vodafone utilizzando quindi un numero breve messo a disposizione dell'azienda.
Oltre ad inviare singoli messaggi, è possibile creare delle vere e proprie **campagne** che riescano a mandare lo stesso SMS a più contatti definendo una data e ora precisa, con la possibilità di inviare il messaggio ripetitivamente definendo l'intervallo di invio e il numero di volte che si vuole mandare l'SMS.
Questa interfaccia web dovrà inoltre essere utilizzata da più utenti utilizzando la stessa connessione al server e di conseguenza lo stesso numero breve affidato all'azienda.

La finalità di questo progetto è molto ampia: riesce infatti a coprire i principali servizi di gestione del personale di qualsiasi azienda nonché riuscire a comunicare direttamente e in maniera veloce a tutti i dipendenti o parte di essi con pochi click di mouse.
Può inoltre soddisfare le richieste di marketing mandando SMS pubblicitari ai principali clienti di un'azienda o a tutti i contatti che ne hanno dato il consenso.

Si provi ad immaginare un grande azienda che debba comunicare in maniera tempestiva un'informazione urgente a tutti i dipendenti. Essa può inviare semplicemente un e-mail a tutti con il rischio però che alcuni dipendenti, impegnati a lavori manuali e molto pratici, non riescano a leggere frequentemente la propria casella di posta, magari anche perchè privati di un ufficio o

di un PC a portata di mano.
Come tutti sanno invece ogni persona, sia durante che al di fuori del normale orario di lavoro, ha con sè un cellulare a portata di mano con il quale si può comunicare in maniera tempestiva e immediata. Ecco quindi che l'utilizzo di un applicazione del genere possa soddisfare questi requisiti.

Anche se l'azienda abbia pochi dipendenti, ma debba promuovere a tutti i clienti un tipo di prodotto a lei caro, è necessario effettuare un buona strategia di marketing per cercare di promuoverlo al massimo. Un semplice SMS che informa tutti i clienti dell'uscita in commercio di questo prodotto è un buon punto di partenza promozionale.

Capitolo 2

MAM Vodafone

2.1 Premessa

Il protocollo MAM è un protocollo proprietario esposto dal servizio di Messaggistica Aziendale Mobile (MAM) di Vodafone [3]. Esso regola la comunicazione tra MAM Server (Vodafone) e MAM Client (applicazione del Cliente MAM) per gestire l'invio e la ricezione di messaggi SMS. Il MAM Server è colui che garantisce l'invio e la ricezione dei messaggi di tutti i client MAM, invece il suo client, è il mezzo utilizzato dalle varie aziende per usufruire del servizio SMS. Ogni MAM client quindi è associato un numero breve a disposizione all'azienda il quale numero rappresenta:

- il **mittente** di tutti i messaggi SMS-MT originati dall'applicazione MAM Client connessa all'account;
- il **destinatario** a cui inviare i messaggi SMS-MO affinchè giungano all'applicazione MAM Client connessa all'account (se l'account stesso è abilitato a ricevere).

La comunicazione tra MAM Server via TCP/IP avviene il questo modo:

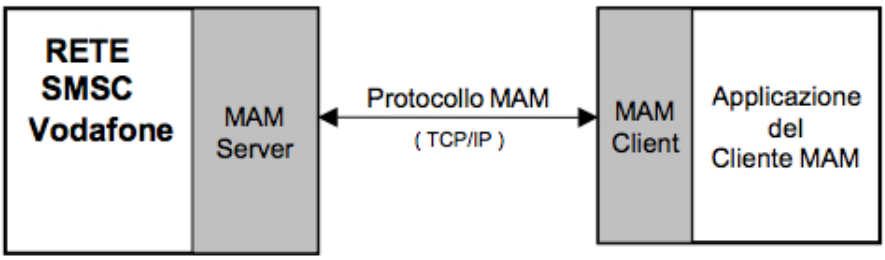

Figura 2.1: *Architettura MAM Client-Server*

La comunicazione tra Server e Client avviene tramite telnet mediante la porta 8000.

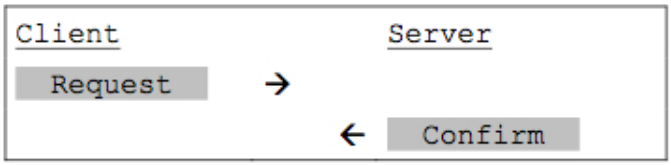

Figura 2.2: *Request-confirm*

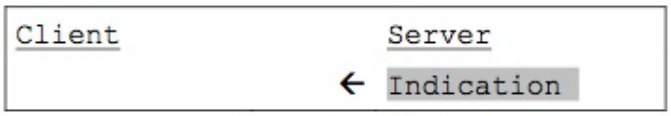

Figura 2.3: *Indication*

2.2 definizione di protocollo

Il protocollo MAM è composto di primitive leggibili del tipo:

- *Request/Confirm* (comunicazione sincrona per iniziativa del client).
- *Indication* (notifiche asincrone provenienti dal server, per le quali non è prevista risposta esplicita da parte del client).

La comunicazione sincrona è quella più affidabile ed efficiente il quanto il client prende l'iniziativa della comunicazione e attende sempre e comunque l'emissione della Confirm da parte del server, prima di inviare una nuova Request.

2.3 Elementi sintattici

Il protocollo è composto dai seguenti elementi sintattici sintetizzati in maniera ottimale da questi diagrammi logici forniti dalla Vodafone (fig. 2.4).

2.3.1 Request

La *Request* è il comando che il client invia al server MAM. La struttura principale del request è definita in questo modo:

```
<directive>
<param>=<value>
<param>=<value>
<blank line>
```

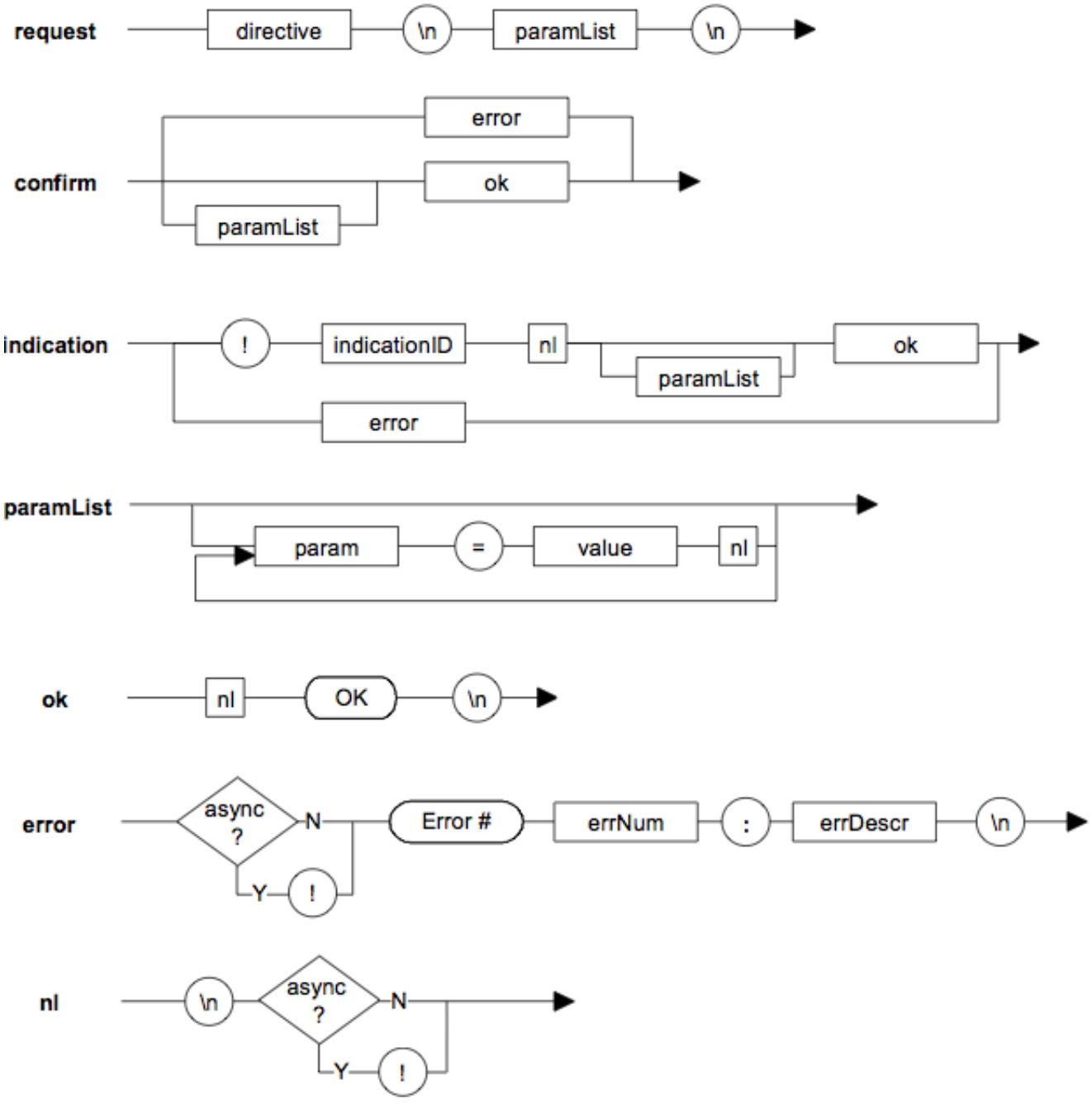

Figura 2.4: *diagramma logico*

Il tag **directive** presenta i vari comandi che il client può inviare al Server con i vari parametri se esso ne dispone.
Come possiamo notare ogni Request finisce con una riga vuota (effettuata tramite il doppio invio). Dopo solo questo momento l'intera Request viene spedita al Server MAM la quale restituisce un messaggio di conferma (**Confirm**), o di errore.

2.3.2 Confirm

Il messaggio di conferma da parte del Server in riposta di una Request può essere di due tipi.

- **Reply**: il quale termina sempre con un OK preceduto da una riga vuota, ciò garantisce la corretta ricezione della Request da parte del Server.

```
<param>=<value>
<param>=<value>..
<blank line>
OK
```

- **Error**: ci restituisce il numero dell'errore seguito da una piccola descrizione. In questo caso la request non è andata a buon fine.

```
Error #<errnum>: <descr>
```

2.3.3 Indication

È un messaggio asincrono spedito dal server al client.
La sintassi dell'Indication è molto simile ad una Confirm; si differenzia da essa solo per il fatto che inizia con un punto esclamativo. Quando è in atto un messaggio *Indication* da parte del server, quest'ultimo non aspetta nessun tipo di risposta da parte del client alle *Indication* ricevute.
Anche in questo caso l'Indication può essere di due varianti: la Notification e l'Error.
Un fatto da prestare molta attenzione è che quando un indication è nella forma di tipo **error** provoca la chiusura della connessione MAM verso il Client.

Questo è uno dei motivi che mi ha fatto scegliere una comunicazione sincrona con il server della MAM, anche per il fatto che così all'invio di un *Request* mi aspetto solo un tipo di *Confirm* e aspetto di mandare un'ulteriore Request solo dopo la ricezione del *Confirm*.

2.4 Comandi Base

Per riassumere al meglio approfondiremo l'argomento solo i *Directive Request* che ci interessano maggiormente quali **Send**, **Report**, e parte del **Set**.

Comando	Descrizione
Send	richiede l'invio di un messaggio
Report	Richiede le informazioni inerenti uno o più messaggi
Set	Definisce le modalità da applicare alla sessione corrente
Delete	Elimina un messaggio dallo Storage del MAM
AccountInfo	Richiede informazioni relative all'abbonamento
KeepAlive	Mantiene viva una connessione
Logout	Richiede un Logout dal Server

Tabella 2.1: *Directive Request di base accettati dalla MAM*

2.4.1 Set

Questo comando definisce le modalità da applicare alla sessione corrente; tali modalità non vengono ereditate dalle sessioni successive. Il primo parametro di chiamata da analizzare è il parametro **Mode** che imposta la modalità di colloquio con MAM per la notifica degli eventi. Il colloquio può essere di tipo sincrono a asincrono.
Il secondo parametro **Storage** definisce la tracciabilità dei messaggi inviati.
Se viene definito *false* (default) il server MAM non tiene traccia del messaggio nel proprio storage; se invece viene definito *true* la MAM mantiene traccia del messaggio nel proprio storage (fino a quando il messaggio giunge in stato non transitorio).

Gli altri parametri non ci risultano particolarmente importanti al fine del nostro utilizzo; mi limito quindi a definire cosa descrivono i parametri di default.

- **NotifyStatus**: definisc quali notifiche di stato verranno emesse dal Server. L'impostazione di default è **all**.
- **SessionTimeout**: Definisce il tempo in minuti entro il quale la sessione viene chiusa dal server per intattività sulla connessione. Il timeout di default è di 3 minuti.
- **ReportFields**: Questo comando è seguito da una lista composta da nomi di campi separati da virgole. Esso definisce quali campi dovranno esser riportati nel Report e nelle notifiche StatusInd emesse dal MAM Server. Ovviamente l'impostazione di default è **all**.
- **Prefix** imposta il prefisso di default da applicare a tutti i messeggi inviati.
- **Ext** imposta l'estensione di default da applicare al mittente di tutti i messaggi inviati.
- **Orig** imposta il mittente di default per tutti i messaggi inviati. Esso può essere il numero breve assegnato dalla Vodafone per il nostro specifico Client o un alias.

Gli ultimi tre parametri richiedono l'abilitazione da parte della Vodafone e quindi sono servizi aggiuntivi.
Per quanto ci riguarda abbiamo la possibilità di poter utilizzare la funzione *alias*.

2.4.2 Send

Il comando Send effettua una Request alla MAM per la spedizione di un messaggio.
I comandi che noi utilizzeremo saranno sicuramente il **Dest** e il **Body**. Ci potrebbe risultare molto utile il parametro **Orig**, per definire il mittente un alias specifico.

Il Parametro di ritorno, se tutto è andato a buon fine è il seguente:

```
MsgID numberID

OK
```

Il **numberID** è il numero di 15 cifre univoco che identifica il messaggio di invio.
L'ID può essere utilizzato per monitorare lo stato del messaggio.

2.4.3 evoluzione dello stato dei messaggi

Il ciclo di vita di un messaggio inviato comprende i seguenti stati (fig 2.5):

- **Queued**: È il primo stato del messaggio appena viene inviata la Request *send*. Il messaggio viene così messo in coda per essere inviato.
  Nella normalità il messaggio rimane in questo stato un tempo molto limitato per essere poi catalogato immediatamente in uno dei due stati seguenti.
- **reject**: il messaggio viene rifiutato dalla rete GSM e quindi non viene inviato.
- **accepted**: il messaggio viene accettato dalla rete GSM. In questa situazione l'SMS può essere rifiutato in seconda instanza e quindi passa allo stato *reject*, oppure passare in uno dei 4 seguenti stati.
- **waitwakeup**: in questo caso il destinatario risulta non disponibile e aspetta la disponibilità del ricevente.
- **cancelled**: il messaggio viene cancellato della rete GSM.
- **expired**: in questo caso è scaduto il periodo di validità dell'SMS e il messaggio non verrà inviato.
- **delivered** il messaggio è stato inviato con successo.

Gli stati sopracitati vengono classificati in due tipi: stati transitori e stati definitivi. Gli stati definitivi risultano *reject, cencelled, expired e delivered.*
Questa differenza di stati è molto importante in quanto, quando risultano in un stato transitorio vengono mantenuti nello storage MAM fino al raggiungimento dello stato definitivo, o comunque fino a un limite massimo di 3 giorni.

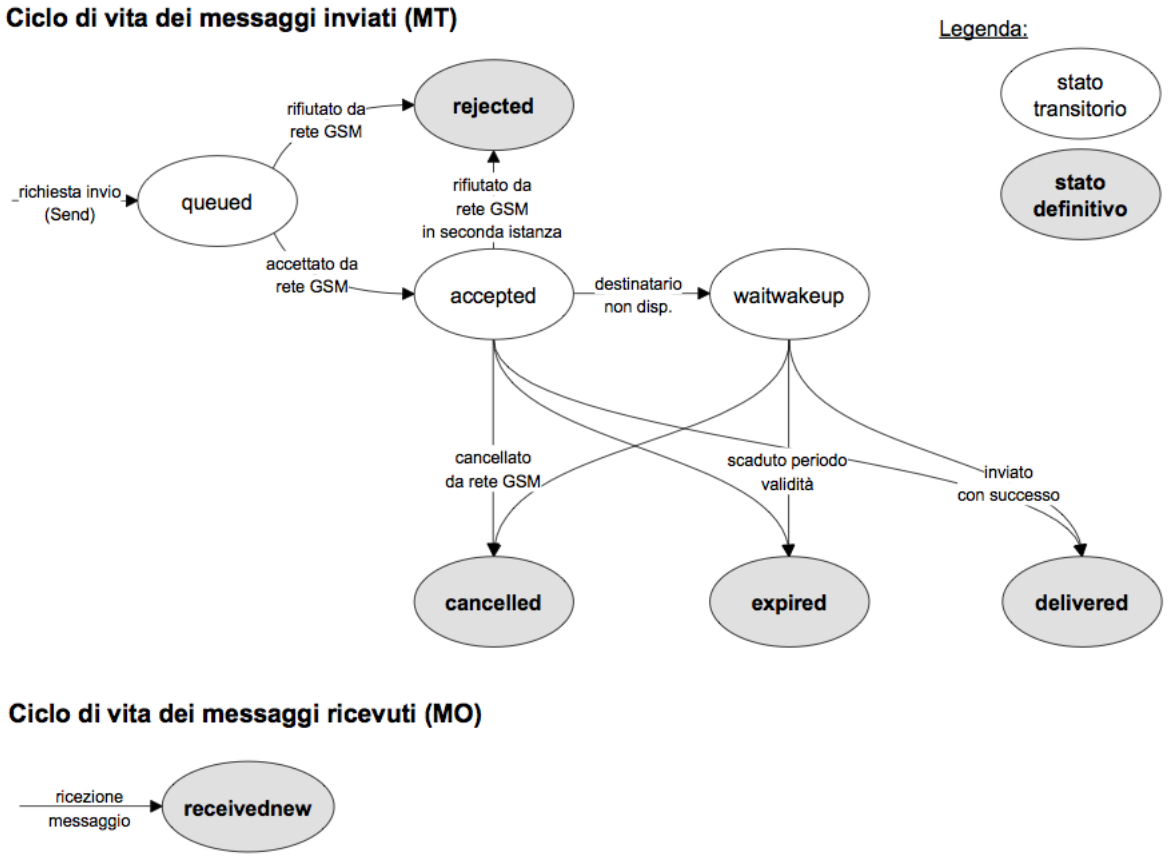

Figura 2.5: *evoluzione dello stato dei messaggi*

I messaggi pubblicati (tramite *Report*) in stato definitivo vengono **eliminati dallo storage dopo la pubblicazione** per far fronte al problema di memoria limitata del server MAM.

Il ciclo di vita invece di un messaggio in ricezione comprende solo uno stato definitivo: **receivednew**.

2.4.4 Report

Il comando Report richiede le informazioni inerenti uno o più messaggi spediti o ricevuti in modalità "storage" (vedi comando *Set* al paragrafo 2.4.1). I parametri di chiamati sono specificati in maniera dettagliata nella tabella A.1 presente nella sezione Appendici.
Se viene specificato il primo parametro (MsgId), tutti gli altri parametri sono ignorati. Se non viene specificato alcun parametro, vengono restituite informazione relative a tutti i messaggi disponibili.

I parametri di ritorno sono invece analizzati nella tabella A.2 nella sezione Appendici.

2.4.5 Utilizzo della MAM Vodafone

Per poter realizzare in maniera corretta il nostro progetto è necessario delineare al meglio come utilizzeremo la console MAM di Vodafone.
Il primo problema da affrontare sono le limitate sessioni che il Server MAM ci dedica.
L'utente infatti può stabilire verso il MAM Server una sola sessione per volta (non sono ammesse sessioni contemporanee).
Una sessione quindi rimane attiva fino a quando il Client manda la Request **Logout** al Server o fino a quando non scade il Timeout di inattività, in questo caso è il Server che imposta la sconnessione.
Per ovviare questo problema i casi possibili possono essere 2:

- Mantenere la connessione sempre attiva, evitando i continui *Login/Logout* da parte del client al Server. Ogni istruzione quindi utilizzerà lo stessa connessione per poi rilasciarla all'istruzione successiva.
  In questo caso si pone un problema di concorrenza in mutua esclusione: il socket di connessione sarà la risorsa (unica) condivisa da ogni istruzione, le quali non potranno usufruire la risorsa contemporaneamente (mutua esclusione).

- Ogni istruzione, prima di eseguire la sua determinata Request al Server Vodafone, dovrà eseguire l'istruzione di *Login*, e successivamente di *Logout*. In questo caso si evita l'utilizzo di una risorsa condivisibile.
  Le istruzioni, comunque, saranno lo stesso in mutua esclusione in quanto non potranno effettuare il *Login* al Server MAM contemporaneamente o prima del logout dell'istruzione che ha aperto la connessione.

Il primo caso è consigliato per Request continue molto vicine tra loro. In questo caso si evita di stressare il Server effettuando polling di *Login/Logout* (considerando che il Server, in questo caso, può penalizzare il Client).
Nel caso in cui il Client effettua Request al Server di continuo, si eviterà inoltre la scadenza del timeout per inattività senza costringere l'uso della Request **KeepAlive** (metodo non affrontato da questi tesi, serve soltanto a resettare il tempo di Timeout prolungando quindi la connessione al Server per più tempo in caso di inattività).
La seconda proposta invece è consigliata a Client che effettuano Request al Server in maniera sporadica e non continuativa (ad esempio invio di un messaggio ogni 5 min, o controllo del report ogni 10 min), senza così stressare il Server MAM e senza andare contro alle "regole di buona educazione".

Un'altra buona regola nel caso in cui un client deve ricevere messaggi o tener traccia dell'esito degli invii è quello di evitare l'uso del polling *Report* utilizzando invece la ricezione in modalità asincrona.
Questo regola dobbiamo però infrangerla in quanto ci risulta difficile rimanere in ascolto e contemporanemente inviare delle Request in quanto può succedere che, mentre noi mandiamo al

server una request di tipo *Send*, si riceve un report di un messaggio in modalità asincrona.
Questo infatti è uno dei motivi che utilizzeremo sempre la modalità sincrona cercando di non "stressare" troppo il Server MAM.

Capitolo 3

Implementazione in Java

3.1 Schema principale

Per creare un client MAM adattato ai nostri scopi, abbiamo deciso di utilizzare il linguaggio Java, in quanto l'intero progetto **ne-t box** (*SMS Gateway* e *ne-t Application*) è implementato in Java.
Lo schema principale dell'intero progetto è diviso in due blocchi.

- il primo blocco gestisce la comunicazione con il server MAM di Vodafone; prende quindi il report dei messaggi ricevuti e il report dei messaggi inviati, ed effettua l'invio dei messaggi.
- il secondo blocco invece effettua tutte le operazioni di comunicazione con l'intero progetto **ne-t box** (non trattato dalla tesi).

Il processo di comunicazione tra i due sistemi viene gestito mediante la JMS presente nel framework Spring [9].

3.2 JMS

Java Message Service è l'insieme di API fornite da Java nel pacchetto Java JEE che consentono lo scambio di messaggi java distribuite sulla rete. In ambito JMS un messaggio è una pacchetto di dati che viene inviato da un sistema all'altro. Questo sistema di comunicazione è stato sviluppato in modalità asincrona: ossia l'invio di un messaggio da parte di uno dei due sistemi, consente la ricezione per opera del destinatario anche **in un secondo momento**. Questo comporta la piena libertà dei due processi in quanto non sono vincolati tra loro.
Infatti quando un client invia un messaggio al ricevente, non deve aspettare la sua ricezione da quest'ultimo ma può continuare ad eseguire il suoi compiti e addirittura inviare altri messaggi.

Inoltre il JMS è basato su un paradigma **peer-to-peer**: un client può ricevere e spedire messaggi ad un qualsiasi altro client attraverso un provider.

Questo tipo di sistema permette una comunicazione distribuita del tipo **loosely coupled** (debolmente accoppiata): il mittente e il ricevente, per comunicare, non devono essere disponibili allo stesso tempo, lasciando quindi una piena libertà di esecuzione di altri processi.
Tutto ciò che il mittente e il ricevente devono conoscere è il formato (message format) e la destinazione (destination) del messaggio.
Inoltre la JMS premette una comunicazione di tipo **affidabile** in quanto garantisce che un messaggio sia consegnato una e una sola volta.

Sono presenti anche dei livelli di sicurezza garanti in JMS: tramite il **Guaranteed Message Delivery** è possibile prevenire una perdita di informazioni in caso di malfunzionamento o di crash del message server, rendendo i messaggi persistenti prima di essere recapitati dai consumatori (per esempio mediante JDBC).

Nel nostro caso la comunicazione oltre che essere asincrona è anche bidirezionale: ossia entrambi i sistemi inviano e ricevono messaggi all'altro per mezzo di code.

3.3 MAM controller

Il sistema di comunicazione con il server MAM è gestito interamente dalla classe java *MAMcontrollerImpl.java* con l'interfaccia *MAMController.java* e comprende i seguenti metodi:

- **stampaRisposta(BufferedReader inFromServer)**: Metodo che stampa a video la riposta del server MAM. Viene utilizzato nei metodi interni alla classe.
- **login(String host, int port, String user, String pwd)**: effettua il login tramite connessione telnet alla MAM. Vengono passati come parametri l'host, la porta utilizzata dalla connessione telnet, l'username e la password per effettuare il login al server MAM.
- **invioMsg(String numero, String Msg)**: Invio di un SMS tramite la MAM. Vengono passati come parametro il numero di cellulare del destinatario e il messaggio di testo da inviare.
- **reportMsgInviati()**: Report dei messaggi inviati in modalità sincrona.
- **reportMsgRicevuti()**: Report dei messaggi ricevuti in modalità sincrona.
- **ricezMsgAsync()**: effettua ricezione di un SMS tramite la MAM in modalita' asincrona. (non viene utilizzato)
- **logout()**: Logout della connessione con la MAM

Nella nostra interfaccia inoltre sono previste le seguenti variabili private:

- **clientSocket**: il socket di connessione;
- **MAMhost**: l'indirizzo IP del server MAM;

- **MAMport**: la porta di comunicazione con il server;
- **MAMuser**: l'username;
- **MAMpassword**: password;

I valori delle ultime 4 variabili private (MAMhost, MAMport, MAMuser, MAMpassword) vengono passati dal file properties *apsMobileProperties* utilizzando le specifiche librerie del framework Spring.

Il metodo *login()* come specificato precedentemente utilizza la variabile statica di tipo *Socket* **clientSocket** effettuando la login alla Server MAM con l'username e password specificati. Inoltre manderà un *Request* di tipo *Set* con il parametro *Storage=true*. In questo modo il server MAM (come specificato nel paragrafo 2.4.1) manterrà traccia del messaggio finchè non arriverà in uno stato definitivo (paragrafo 2.4.3).

Ho pensato di utilizzare la variabile clientSocket come privata all'interno della classe in modo che ogni metodo possa utilizzarla gestendo il problema di concorrenza (spiegato nel dettaglio nel paragrafo 3.4).

Il secondo metodo invece (*invio Msg*) effettua l'invio del messaggio inviando alla MAM una request di tipo Send con parametri *Dest* il numero del destinatario e *Body* il testo del messaggio. Il Server, come specificato nel capitolo 2, effettuerà un messaggio di Confirm che verrà stampato nel Log a disposizione del programmatore.

3.3.1 Report dei messaggi

Il terzo e il quarto parametro effettuano la visione del *Report* dei messaggi rispettivamente inviati e ricevuti. Il primo effettuerà un Request *Report* con parametro *Type=sent*, mentre il secondo utilizzerà il parametro *Type=receivednew*. I seguenti report verranno stampati in questo modo:

```
Report
Type=receivednew

MsgID=40884401181159
Encoding=7bit
LastUpdate=2011/01/18 10:59:55 GMT
Prefix=nonOPI
Orig=393479074683
Status=receivednew
Date=2011/01/18 10:59:55 GMT
Body=Fjdjrfdkeod
Dest=4112584

OK
```

```
Report
Type=sent

MsgID=1011482960154
Encoding=7bit
LastUpdate=2011/01/18 10:57:18 GMT
Orig=4112584
Status=delivered
Date=2011/01/18 10:57:11 GMT
Body=sajsdahsadljh
Dest=393479074683

OK
```

Come possiamo osservare i due Report contengono pressapoco le medesime informazioni ma posizionati in righe differenti: per quello mi è sembrato opportuno creare due metodi differenti che gestissero l'output del server MAM prendendo le loro rispettive informazioni.
Viene effettuato quindi il parsing del report stampato dal server MAM e vengono prelevate la seguenti informazioni:

- L'ID del messaggio (univoco).
- Lo stato del messaggio.
- la data di invio (o di ricezione).
- il numero del destinatario (o del mittente).
- il testo del messaggio.

Nel report (sia per i messaggi inviati che in quelli ricevuti) sono presenti due campi data: *Date* e *LastUpdate*. Essi si riferiscono solo dal fatto che nel primo viene visualizzata la data del messaggio in cui compare per la prima volta a prescindere dallo stato in cui si trova. La seconda invece si riferisce alla data di aggiornamento di stato del messaggio.
Per quanto riguarda i messaggi ricevuti, i due campi conterranno la medesima data in quanto l'evoluzione dei messaggi ricevuti (cap. 2.4.3) comprende solo lo stato *receivednew*; quindi non ci sarà nessuno aggiornamento di stato se non la prima ricezione del messaggio.

Nel particolare dell'esecuzione di un report di messaggi inviati conviene fare un esempio:
Se noi inviamo un messaggio e immediatamente eseguiamo un report dei messaggi inviati, è molto probabile che il messaggio inviato sia ancora nello stato *accepted*, in quanto il server MAM deve avere ancora la conferma di ricezione del messaggio da parte del destinatario. In console avremo queste righe:

```
Sent
Dest=393497288560
Body=prova messaggio

MsdID= 1011488764196

OK
Report
Type=sent

MsgID=1011488764196
Encoding=7bit
LastUpdate=2010/12/20 23:14:22 GMT
Orig=4112584
Status=accepted
Date=2010/12/20 23:14:22 GMT
Body=prova messaggio
Dest=393497288560

OK
```

Anche in questo caso i due campi hanno la medesima data in quanto il primo stato occupato da un messaggio inviato è lo stato *accepted*.
Nel caso in cui noi eseguiamo ulteriormente il Report dei messaggi inviati dopo un certo instante di tempo (il tempo necessario che il messaggio arrivi al destinatario) avremo questo responso:

```
Report
Type=sent

MsgID=1011488764196
Encoding=7bit
LastUpdate=2010/12/20 23:15:13 GMT
Orig=4112584
Status=delivered
Date=2010/12/20 23:14:22 GMT
Body=prova messaggio
Dest=393497288560

OK
```

In questo caso il report conterrà due campi data differenti in quanto il messaggio è stato inviato alle *23:14:22 GMT* ma ha aggiornato lo stato passando *delivered* alle *23:15:13 GMT* (secondo

il campo LastUpdate).
A noi interessa l'ora effettiva di ricezione del messaggio per cui prenderemo in considerazione il campo *LastUpdate*.
Questi dati vengono poi salvati nei corrispettivi campi nell'oggetto di tipo *SMSReceived* (vedere appendice B.3).

3.4 problema concorrenza

Esiste un problema di concorrenza in quanto è possibile effettuare solo una sessione con il server MAM. Per questo motivo ho ritenuto opportuno rendere il Socket di connessione una variabile statica della classe condivisibile da tutti i metodi della classe *MAMController*. Questa connessione infatti sarà aperta e loggata dal metodo iniziale *login()* per poi essere utilizzata dagli altri metodi.
Nel nostro caso, dato che il sistema sarà sempre in continua esecuzione (invio e ricezione dei messaggi) non verrà mai effettuato il *logout*, la connessione con il server MAM sarà sempre attiva.
Il problema però è garantire che i metodi della classe utilizzino il Socket uno alla volta: abbiamo quindi un problema di mutua esclusione dove la variabile condivisa è il Socket di connessione.

3.4.1 controllo connessione

Se controlliamo nel dettaglio il codice java di *MAMControllerImpl.java* (B.2) possiamo notare che all'inizio di ogni metodo c'è un controllo del socket di connessione: viene controllato infatti se il Socket *clientSocket* presenta ancora una connessione aperta oppure no. Nel caso in cui, per qualche motivo, la connessione fosse stata chiusa, MAMController si occuperebbe di effettuare ulteriormente il login.

3.5 ciclo di esecuzione

Appena viene creata un'istanza della classe MAMController, il costruttore si occupa di eseguire il metodo *login*. Gli altri metodi vengono eseguiti mediante un ciclo infinito da **SMSGateway**. La struttura temporale del programma è descritta dalla figura 3.1.

Si puó notare dall'illustrazione che il nostro programma, dopo aver effettuato la login, esegue il report dei messaggi inviati e ricevuti per poi comunicare tramite le code JMS alla ne-t box. Quest'ultimo viene avvisato dal framework di Spring che la coda **SMSin** (ossia la coda dei messaggi ricevuti) risulta non vuota e quindi esegue tutte le sue rispettive operazioni. Per quanto riguarda i messaggi del report inviati, essi servono a ne-t box solo per il controllo di avvenuta

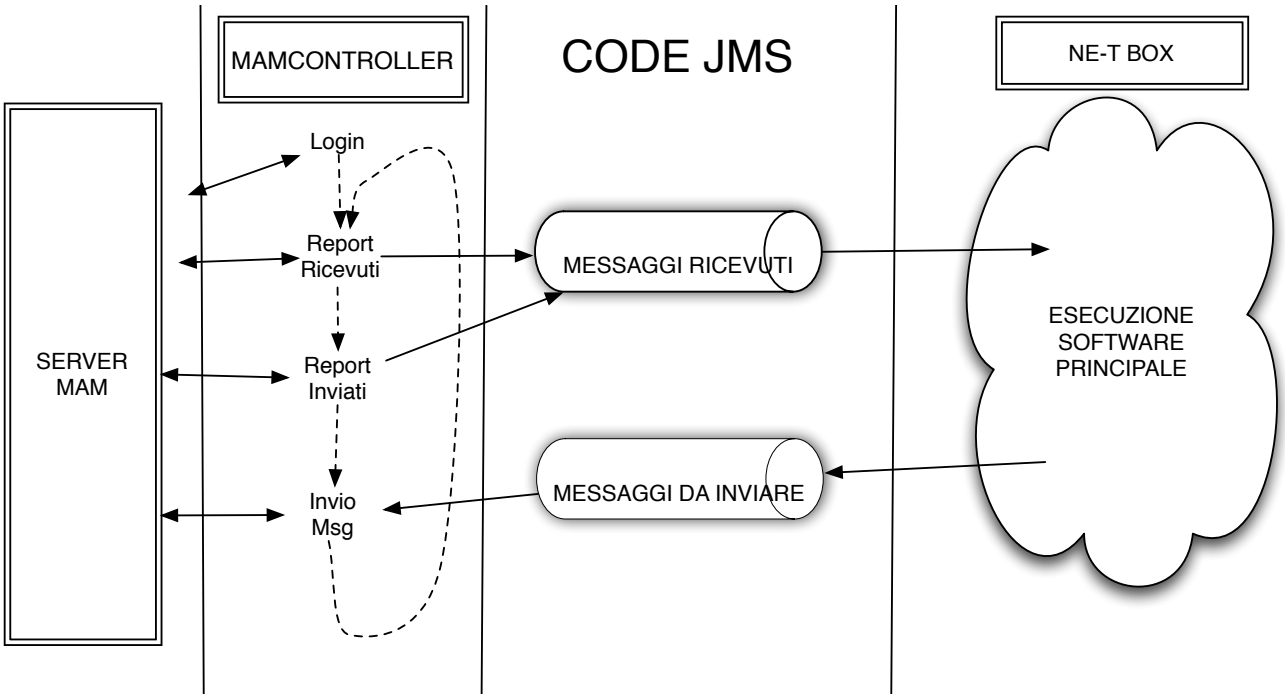

Figura 3.1: *ciclo di esecuzione*

ricezione del biglietto elettronico da parte del cliente.
Successivamente il MAMController controlla la coda **SMSout** (ossia la coda dei messaggi da inviare) se risulta non vuota. Nel caso affermativo, si occupa dell'invio dei messaggi, per poi continuare ad eseguire le istruzioni sopra descritte.

Possiamo osservare come questo tipo di esecuzione soddisfa ottimamente il problema di concorrenza di connessione in quanto solo un Thread, in maniera sequenziale e lineare, utilizza il socket di connessione. Inoltre questo tipo di approccio comporta il fatto che i *MAMController* e *ne-t box* risultano sempre independenti tra di loro.
Nel caso in cui il *MAMController* sia impegnato per molto tempo a ricevere molti messaggi tramite il Report, la *ne-t Application* non deve aspettare la fine della ricezione in quanto, di volta in volta, ogni messaggio ricevuto dal MAMController viene spedito alla coda, per essere usato prontamente da ne-t box.
Viceversa, nel caso in cui la ne-t box debba inviare molti messaggi, MAMController non deve aspettare tutto il ciclo di operazioni che deve compiere ne-t Application per tutti i messaggi (comunicazione dati con Movimbox, gestione degli errori, ecc); ma può spedire degli SMS ogni qualvolta risultano pronti nella coda.
Nel ciclo infinito di esecuzione viene inoltre aggiunto un *Thread.sleep* per non effettuare un polling continuo di *Request* al server MAM garantendo così un ottimo comportamento di comunicazione.

Capitolo 4

Creazione di interfaccia web di gestione di invio SMS e di campagne pubblicitarie

4.1 Schema principale del progetto

Il progetto dovrà utilizzare diversi linguaggi di programmazione e diverse strutture per soddisfare i requisiti:

- Dovrà infatti interfacciarsi con il server MAM della Vodafone utilizzando la classe java sviluppata nel capitolo 3.
- Interfacciarsi con un Database per utilizzare e salvare i dati del progetto.
- Avere un'interfaccia web per essere utilizzato dall'utente finale.
- Essere multi-utente, quindi aver la possibilità di essere utilizzato **in contemporanea** da più utenti che utilizzeranno **solo** i rispettivi dati.

È importante utilizzare un database in quanto abbiamo bisogno di salvare diversi tipi di dati in maniera persistente. I tipi di dati da salvare sono:

- i dati delle aziende che usufruiscono di questo servizio.
- i contatti a cui spedire gli sms.
- tutte le campagne pubblicitarie e gli elenchi dei contatti.
- i report di messaggi inviati e ricevuti.

Notiamo quindi che il nostro progetto java risulta più ampio e complesso rispetto al progetto precendente.
La classe *MAMController* ci risulta tuttavia sempre utile per realizzare il nostro scopo.
In particolare la struttura del nostro progetto è divisa in tre classi principali:

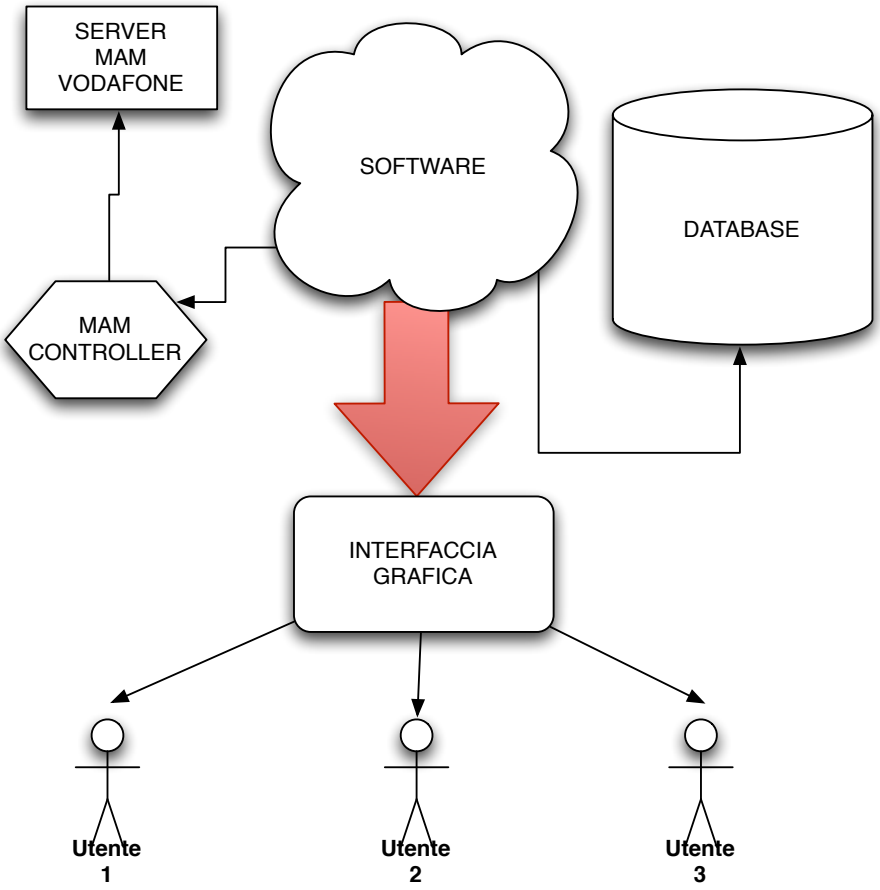

Figura 4.1: *schema del progetto*

- *MAMController*: classe già trattata con qualche modifica.
- *Apache*: classe java che comprende tutti i metodi chiamati dal server web. In essa vengono racchiusi tutte le chiamate di MAMController, gestisce il problema di concorrenza e richiama inoltre i metodi di DbSQL.
- *DbSQL*: gestisce tutta la comunicazione con il Database, racchiude tutti i metodi che eseguono query di *insert*, *select* e *delete*.

4.1.1 modifiche alla classe MAMController

Abbiamo modificato leggermente il codice della *MAMController* in quanto certe tipi di operazioni risultavano incompatibili con il nostro progetto.
Innanzitutto abbiamo eliminato il costruttore e il suo ciclo infinito di istruzioni, in quanto non ci interessa che la nostra piattaforma sia collegata in continuazione al server MAM. Essa infatti dovrà eseguire un login e logout ogni qualvolta l'utente vorrà spedire e/o ricevere dei messaggi. Inoltre l'utilizzo del Socket di connessione sarà interamente gestito dalla classe Apache, per cui

il metodo *login* ritornerà una variabile di tipo Socket e il metodo *logout* avrà come parametro esterno il Socket di connessione.

```
/**
         * metodo di login via telnet. Esso crea un nuovo Socket e restituisce lo
         * stesso socket appena creato
         *
         * @param String host indirizzo IP
         * @param int port porta di comunicazione
         * @param String user username del Client MAM
         * @param String pwd password del Client MAM
         * @return Socket connessione della MAM aperta.
         * @throws Exception
         */
        public Socket login(String host, int port, String user, String pwd)
                        throws Exception;

        /**
         * Logout della connessione con la MAM
         *
         * @param clientSocket
         *            Socket di connessione
         * @throws Exception
         */
        public void logout(Socket clientSocket) throws Exception;
```

Di conseguenza tutti gli altri metodi avranno in aggiunta come parametro il Socket aperto dal metodo *login*.
Sarà abolita inoltre la comunicazione via JMS, utilizzando come *return* nei metodi Report una lista di tipo *SMSReceived*, la quale verrà utilizzata da Apache per salvare i dati dei Report nel database.
La gestione di concorrenza verrá quindi spostata nella classe *Apache* rendendola un vero e proprio perno di comunicazione tra il serverMAM e il database.

4.1.2 flusso temporale delle operazioni

Osservando la figura 4.2, ogni qualvolta che si vuole mandare un SMS o eseguire un Report, si esegue un ciclo di istruzioni *Login/Request/Logout* in quanto non ha alcun senso mantenere aperta la connesione al server MAM durante i periodi di innattivitá della piattaforma. Si evita così il problema di caduta connessione a causa del timeout (cap 2.4.1), avendo un buon comportamento con server MAM in quanto viene presa la banda di connessione solo per lo stretto necessario.

4.1.3 classe Apache

I metodi principali dalla classe Apache sono i seguenti:

- **InvioMsg**: spedisce un determinato messaggio restituendo una stringa formattata in HTML da stampare nella pagina JSP corrispondente. In essa è contenuto il responso dell'invio.

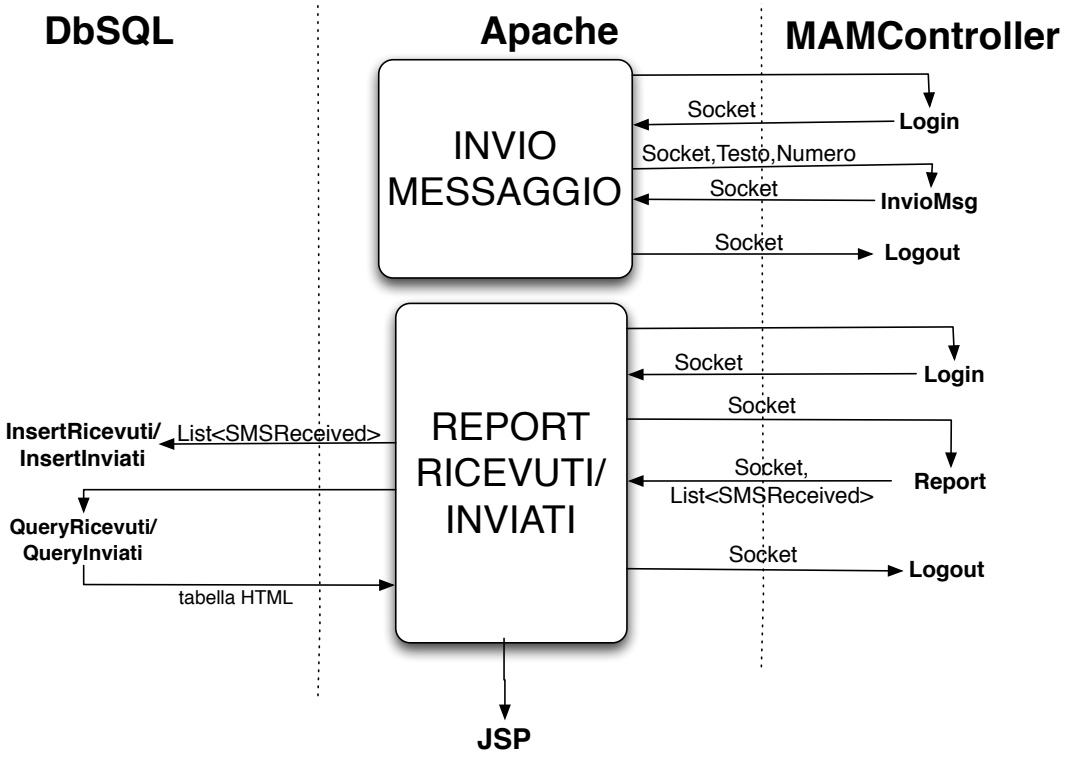

Figura 4.2: *flusso temporale di invio e report messaggi*

- **QueryRicevuti**: esegue il report dei messaggi ricevuti, li salva nel database nel caso ce ne siano. Successivamente esegue un query al database restituendo una tabella HTML in cui sono presenti tutti i messaggi ricevuti salvati nel Db.
- **QueryInviati**: esegue il report dei messaggi inviati, li salva nel database nel caso ce ne siano. Successivamente esegue un query al database restituendo una tabella HTML in cui sono presenti tutti i messaggi inviati salvati nel Db.
- **invioSMSdaElenco**: esegue una query al database per avere tutti i numeri di telefono dei contatti presenti nell'elenco. Successivamente invia il messaggio ad ogni contatto.
- **invioSMSCampagna**: esegue una query al database per avere i numeri di telefono dei contatti presenti nella campagna promozionale. Successivamente invia il messaggio ad ogni contatto. Una volta inviati tutti gli SMS, aggiorna il campo **OraDiInvio** e controlla se nella campagna é presente l'opzione di ripetizione. In quel caso aggiorna il campo di **OraPrevistaInvio** decrementando il numero di volte da ripetere.

4.1.4 Nuova gestione di concorrenza

Tutti i metodi della classe *Apache* sono **syncronized** e **static**. Il motivo risulta essere chiaro: in questo modo i metodi sono gestiti mediante il monitor di java e garantiscono la gestione di

concorrenza. Essendo i metodi anche statici, qualsiasi thread che richiama uno dei seguenti metodi, può eseguirlo se e solo se nessun altro thread sta eseguendo lo stesso metodo o altri della medesima classe.
Per la JVM il compito sarà quello di associare ad ogni oggetto un *lock* che un *thread* che attiva quando tenta di eseguire un metodo sincronizzato prima di eseguirne il codice. Se un *lock* è già posseduto da un altro *thread*, quello concorrente deve attendere sulla coda della regione (coda FIFO di *thread*).[4]

Ho pensato di spostare il controllo di mutua esclusione anzichè in *MAMController* come nel progetto precendente, nella classe *Apache* semplicemente perchè ora, i blocchi principali che devono essere gestiti in mutua esclusione non sono più i singoli metodi di *MAMController*, ma l'insieme delle istruzioni *Logini/Request/Logout*.
Qualora erroneamente avessimo lasciato la gestione di concorrenza come nel progetto precendente risulterebbe possibile che due *thread* (quindi due utenti diversi) eseguissero la login alla MAM uno dopo l'altro prima che il primo rilasci la connessione. Questo fatto comporterebbe un lancio di eccezioni da parte del secondo *thread* compromettendo la sua richiesta.

4.1.5 Classe DbSQL

Questa classe è già stata nominata in questo capitolo e ora vedremo quali metodi ne fanno parte e le loro specifiche funzioni. Essa è stata creata per eseguire istruzioni di login logout al server MySQL in cui è presente il nostro database e eseguire le principali query di cui abbiamo bisogno.
Innanzitutto per creare una connessione stabile con il nostro database è opportuno creare una nuova istanza di classe del nostro driver jdbc. In questo modo possiamo creare una nuova connessione di tipo *Connection* utilizzando il metodo *getConnection* avente come parametro la seguente stringa: `jdbc:mysql://SQLurl:SQLport/database?user=SQLuser&password=SQLpassword`.
La variabili **SQLurl**, **SQLport**, **database**, **SQLuser**, **SQLpassword** sono variabili statiche dichiarate all'interno della classe, al fine di poter essere modificate dal programmatore in modo semplice nel caso in cui debba cambiare qualche configurazione del server MySQL.
Queste istruzioni vengono svolte nel metodo *loginMySQL* il quale restituirà la variabile *Connection*.
Per eseguire il logout basta semplicemente chiudere la connessione utilizzando il metodo *close* presente della classe *Connection*.

Ogni metodo sotto elencato richiamerà quindi il metodo *loginMySQL*, eseguirà le sue istruzioni per poi eseguire il logout e interrompere la connessione con il server MySQL. In questo caso non abbiamo nessun problema di sessioni multiple in quanto il nostro server MySQL che utilizzeremo acccetterà più sessioni contemporaneamente, anche da parte dello stesso utente.

I metodi presenti nella classe *DbSQL* sono:

- **loginMySQL** e **logout**: già analizzati precedentemente.
- **InsertInviati**: inserisce i messaggi di tipo *SMSReceived* passati come parametro mediante una Lista nella tabella *Inviati* presente nel database.
- **InsertRicevuti**: inserisce i messaggi ricevuti nella stessa maniera del metodo precedente nella tabella *Ricevuti*.
- **QueryInviati**: esegue la query `SELECT * FROM Inviati`; ogni tupla viene inserita in una stringa contenente una tabella in linguaggio HTML. La stringa successivamente sarà la variabile di ritorno del metodo.
- **QueryRicevuti**: esegue la stessa procedura del metodo sopra elencato con l'unica differenza della query `SELECT * FROM Ricevuti`;
- **ResetReportInviati**: cancella ogni tupla presente nella tabella **Inviati** del database eseguendo la query `DELETE FROM Inviati`;
- **ResetReportRicevuti**: cancella ogni tupla presente nella tabella **Ricevuti** del database eseguendo la query `DELETE FROM Ricevuti`;
- **printHTMLCampagna**: restituisce una stringa in formato HTML in cui è presente una tabella nella quale sono presenti tutti i contatti di una determinata campagna passata per parametro.
  Per gestire il problema di multiutenza viene passato come parametro anche il nome dell'azienda che esegue questo comando, in modo da garantire la visualizzazione di una campagna creata dal giusto utente. In questo modo si risolve il problema di campagne omonime create da utenti diversi. La query SQL eseguita dal questo metodo è: `SELECT Nome, Cognome, Numero, DataNascita FROM Persone NATURAL JOIN Campagne_has_Persone NATURAL JOIN Campagne WHERE NomeCampagna = nomecampagna AND Azienda_Nome= NomeAzienda`;
- **printHTMLElenco**: esegue la stessa istruzione del metodo precendete lavorando però in questo caso con le tabella Elenchi_has_Persone. Query SQL: `SELECT Nome, Cognome, Numero, DataNascita FROM Persone NATURAL JOIN Elenchi_has_Persone NATURAL JOIN Persone WHERE NomeCampagna= nomecampagna AND Azienda_Nome= NomeAzienda`;
- **printReportCampagne**: restituisce in una tabella HTML tutte le campagne di un determianto utente. La query SQL in questo caso sarà: `SELECT Nomecampagna, Messaggio, OraInviata, OraPrevistaInvio, Stato, Ripetizione, RipValue, RipUnit, NumRip FROM Campagne WHERE Azienda_Nome=NomeAzienda`;
- **printCampagneModifica** e **printCampagneCancella**: questi due metodi si differiscono da **printReportCampagne** solo per il fatto che è possibile (tramite un parametro booleano) scegliere di aggiungere alla tabella HTML risultante un ulteriore campo input per scegliere

le varie tuple.
Il primo metodo utilizza campi di tipo *"radio"*, avendo quindi l'opportunità di scegliere solo una campagna alla volta (è possibile infatti modificare solo una campagna alla volta). Il secondo invece utilizza input di tipo *"checkbox"* con i quali è possibile selezionare più di una campagna.

- **insertSMS**: questo metodo viene utilizzato per riempire il campo **Messaggio** della tabella campagne con il testo da inviare a quella specifica campagna.
- **insertRipetizione**: viene utilizzato per aggiornare i campi di ripetizione quali OraPrevistaInvio, Stato, Ripetizione, RipValue, RipUnit, NumRip.
- **cancellaCampagna**: cancella una campagna dalla tabella **Campagne**
- **printUtenti**: restuisce una tabella HTML con i dati di tutti gli utenti registrati nel database. Questa tabella viene stampata nell'home page dell'amministratore in modo da avere sottomano tutti gli username e password degli utenti registrati.

Per capire al meglio le query eseguite dalla classe **DbSQL** è opportuno analizzare approfonditamente il prossimo capitolo, il quale descriverà nel dettaglio il database utilizzato.

Capitolo 5

Implementazione del Database

5.1 Analisi del database

Come studiato dal libro [8] è necessario effettuare un'analisi dei prerequisiti prima di costruire il database e analizzare successivamente lo stesso.
Ogni azienda inserita in questo progetto ha la possibilità di gestire una o più campagne. Per ogni campagna è necessario raccogliere le informazioni di invio quali il testo del messaggio, data e ora di invio e eventuale ripetizione. In ogni campagna possono partecipare più contatti di cui sappiamo nome, cognome, numero e data di nascita. Questi contatti ovviamente possono essere presenti in più campagne in quanto la presenza di una campagna non esclude la partecipazione di un'altra.
Lo stesso principio vale per gli elenchi: ogni azienda può avere molteplici elenchi nei quali sono presenti più contatti telefonici.

5.1.1 Schema concettuale

Le entità principali della nostra piccola base di dati risultano **Aziende**, **Campagne**, **Persone**, e **Elenchi_has_Persone**.

Aziende Tiene conto di tutte le caratteristiche dell'azienda quali username e password per la login al nostro software, email per la conferma della registrazione, indirizzo (via, cap, città), telefono, fax (queste ultime facoltative).
Ogni inserimento di un'azienda a questa entità viene effettuata dal web admin.

Campagne Tiene conto di tutte le informazioni necessarie della campagna. Innanzitutto sarà qui che verrà salvato il testo dell'SMS in quanto è uguale per tutti i contatti che partecipano alla campagna. Inoltre verranno salvati la data e ora dell'invio della partenza se viene scelto l'invio

programmato della campagna. Oltre a ciò abbiamo reso disponibile l'invio con ripetizione: saranno salvati in questa entità l'intervallo di ripetizione e il numero di volte che si vuole inviare la nostra campagna promozionale.

Persone In questa entità vengono salvati tutti i contatti che saranno utilizzati da tutti gli utenti del servizio. Essendo un servizio di invio SMS è necessario tener conto solo del nome, cognome e numero di cellulare. È possibile inoltre salvare in maniera opzionale la data di nascita del contatto.

Elenchi_has_Persone Tiene conto delle elenchi di rubrica delle varie aziende. Ogni persona rappresentata nella tabella precendente può partecipare a più elenchi anche di aziende differenti.

Le relazioni che intercorrono tra le nostre entità vengono raffigurate dallo schema entità relazionale rappresentato nella figura 5.1.

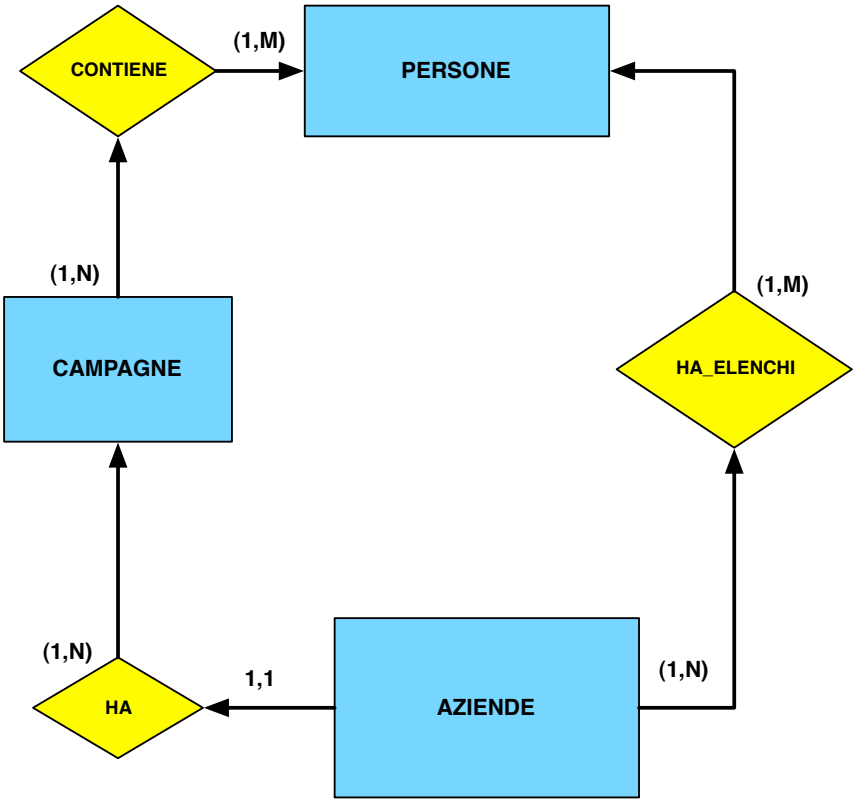

Figura 5.1: *schema ER del database*

Sono presenti tuttavia anche altre due tabelle che non hanno nessuna relazione con le altre tabelle: le tabelle **Inviati** e **Ricevuti**. In essi infatti sono presenti solo i messaggi dei due rispettivi report, non avendo quindi nessun nesso logico con eventuali campagne e/o elenchi di rubrica. Gli attributi di queste tabelle sono le informazioni presenti nell'oggeto *SMSReceived* creato appositamente (vedi capitolo 3.3.1).

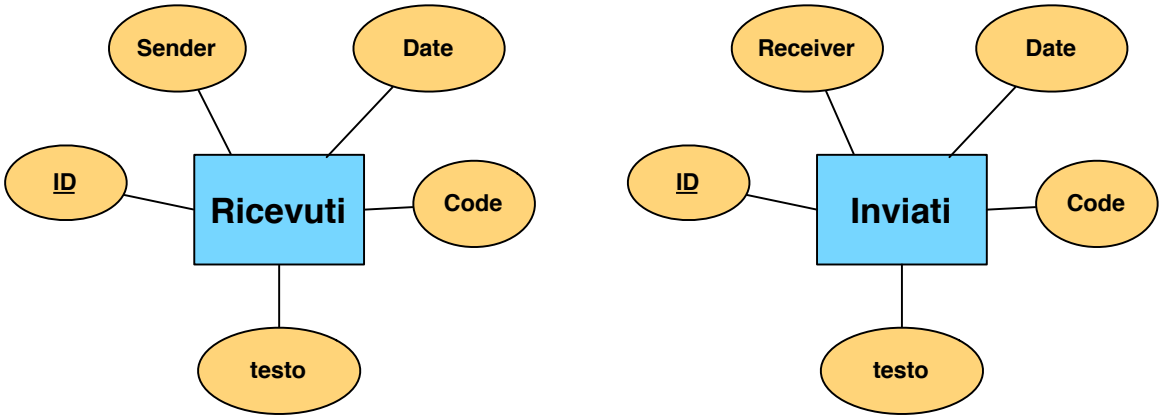

Figura 5.2: *tabelle Inviati e Ricevuti*

Grazie allo schema ER appena progettato, possiamo osservare che ogni azienda puó lavorare solo sulle campagne ed elenchi in relazione con il suo username, con l'impossibilità visualizzare e/o modificare quelli degli altri utenti. In questo modo é garantita la privacy dei contatti di ogni azienda nel rispetto dell'altra. Questo avviene perché le query principali generate dal server web per visualizzare i dati di ogni azienda (come possiamo osservare nel capitolo 4.1.5) sono del tipo:

```
SELECT * FROM Campagne WHERE Azienda_Nome="nomezienda";

SELECT * FROM Elenchi_has_Persone WHERE Azienda_Nome="nomeazienda";
```

Utilizzando sempre questa condizione (insieme ad altre più specifiche), abbiamo la certezza che il risultato sarà riferito solo all'utente in cui è stato effettuato l'accesso [7] [6]. Per quanto riguarda la struttura delle tabelle si può consultare l'appendice C.

5.1.2 Schema Logico Relazionale

Lo schema logico relazionale è ben rappresentato dalla figura 5.3 si può osservare come i vari attributi si relazionino tra le varie entitá diventando cosí chiavi primarie o chiavi esterne.

Tutti le chiavi esterne sono impostate con i parametri **ON DELETE** e **ON UPDATE** entrambi settati in modalità **CASCADE**. In questo modo, per esempio, qualsiasi modifica in ogni tupla di qualsiasi contatto telefonico, sarà aggiornata anche nella rispettiva tupla nelle tabelle *Elenchi_has_Persone* e *Campagne_has_Persone* qualora siano presenti.

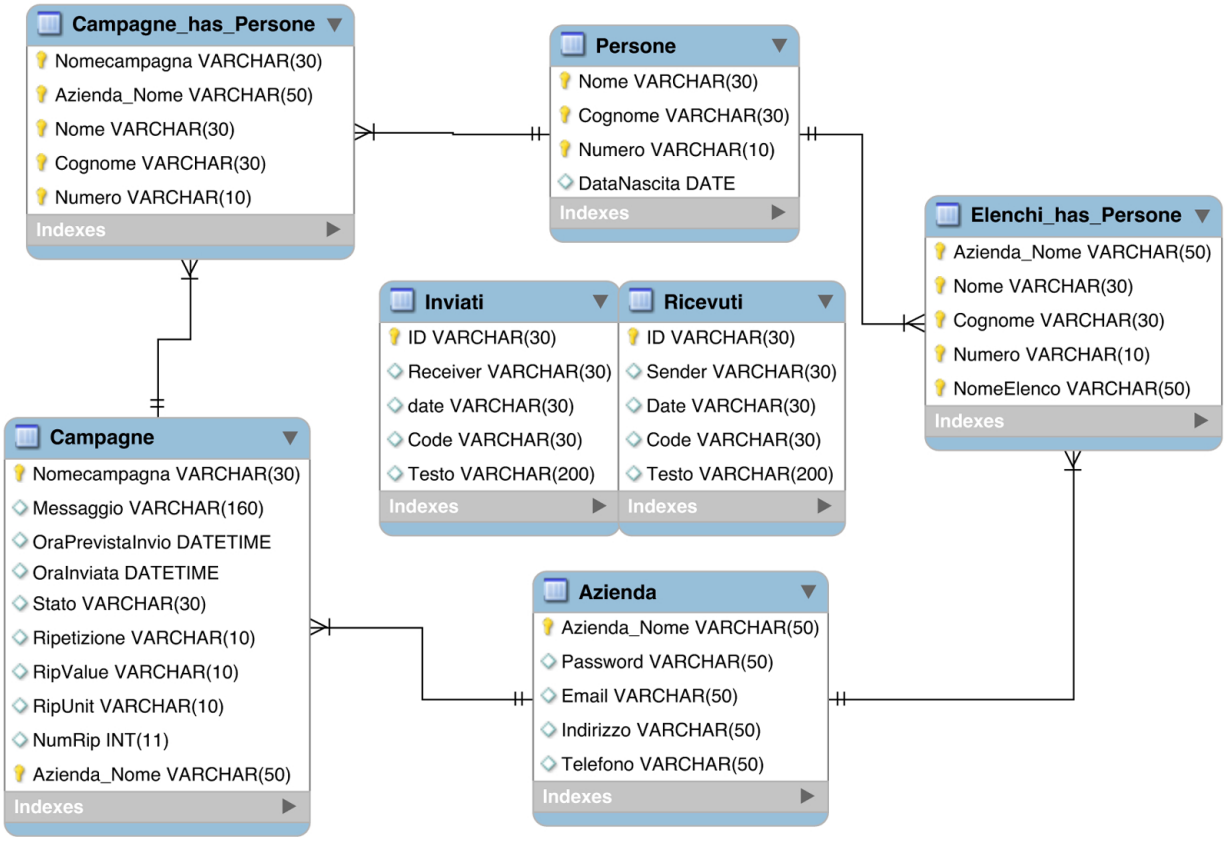

Figura 5.3: *Schema Logico Relazionale del database*

Capitolo 6

Interfaccia web

Per poter iniziare la nostra piattaforma web, dobbiamo scegliere quale tipo di linguaggio utilizzare per sviluppare al meglio le nostre idee i nostri scopi. Ovviamente l'HTML sarà il nostro compagno di viaggio per tutta la stesura del server web in quanto è il linguaggio principale di ogni sito e di ogni browser.
L'HTML infatti verrà utilizzato per indicare come i vari elementi vanno disposti in una pagina Web, per stampare del testo e /o tabelle all'interno della pagina, per utilizzare pulsanti, form, ecc.
Il codice HTML essenzialmente risulta il codice finale di una pagina web, inviato dal **server web** per poter esser letto dall'utente mediante un **browser**. Molto spesso però si ricorre nel creare delle pagine web **dinamiche**, interattive, grazie alla quale l'utente può compiere operazioni avanzate.
L'interattività di una pagine è ricavata dal fatto che ogni pagine dinamica è scritta con linguaggio HTML misto ad altri linguaggi (ASP, Perl, PHP). Queste pagine vengono poi compilate dal server web solo per poter trasmettere all'utente finale l'HTML essenziale per visualizzare la pagina secondo le sue richiste.

6.1 utilizzo del JSP

Un'altro linguaggio che permette la stesura di pagine dinamiche è il linguaggio **JSP** (*JAVA Server Page*)[2]. Si tratta una tecnologia multipiattaforma, che affianca le servlet Java, create per separare i contenuti dalla loro presentazione. JSP è una soluzione "embed": una pagina JSP infatti, è costituita da markup HTML frammentato da sezioni di codice Java. Si potranno quindi modificare le parti sviluppate in Java lasciando inalterata la struttura HTML o viceversa.

6.1.1 Principali Vantaggi

Le JSP si basano su tecnologia Java ereditandone i vantaggi garantiti dalla metodologia object oriented e dalla quasi totale portabilità multipiattaforma.

Quest'ultima caratteristica si rivela tanto più vantaggiosa quanto più ci fosse necessità di ambienti di produzione con sistemi operativi diversi tra loro (es Windows e Linux).
Essere object oriented significa anche avere nel DNA una predisposizione al riuso del codice: grazie ai Java Bean si possono includere porzioni di codice che semplificano l'implementazione di applicazioni anche complesse, anche senza approfondite conoscenze di Java, e ne rendono più semplice la modifica e la manutenzione.
La gestione delle sessioni introdotta con il JSP favorisce lo sviluppo di applicazioni di commercio elettronico fornendo uno strumento per memorizzare temporaneamente lo stato delle pagine fino alla chiusura del browser come vedremo nel capitolo 8.
L'ultimo principale vantaggio (ma non per questo il meno importante) consiste nel fatto che le pagine JSP vengono compilate in **lato server**. Ciò significa che l'utente finale è ignaro del codice Java presente nelle pagine web (e quindi tutto il meccanismo logico creato dal programmatore). Infatti l'utente finale, nel caso voglia osservare il codice sorgente di una pagina scritta in JSP, vedrà solo il codice HTML scelto come output dal servlet dopo aver compilato la pagina.

6.1.2 Ciclo di vita della richiesta

Una Java Server Page può essere invocata utilizzando due diversi metodi:

- la richiesta può venire effettuata direttamente ad una pagina JSP, che grazie alle risorse messe a disposizione lato server, è in grado di elaborare i dati di ingresso per ottenere e restituire l'output voluto.

- la richiesta può essere filtrata da una servlet che, dopo l'elaborazione dei dati, incapsula adeguatamente i risultati e richiama la pagina JSP, che produrrà l'output.

6.1.3 Le servlet

Come abbiamo accennato le JSP affiancano le servlet che comunque giocano un ruolo fondamentale nella creazioni di applicazioni web.
Una servlet è una particolare applicazione che estende le funzionalità dell'application server. Ad esempio si possono creare servlet che gestiscano alcuni flussi di richieste HTTP che necessitano di particolare elaborazioni.

Tipicamente una servlet è rappresentata da una classe che implementa l'interfaccia Servlet o HttpServlet.
Le servlet inoltre possono essere utilizzate per costruire pagine HTML da inviare direttamente al client. A differenza di uno script CGI le servlet vengono caricate una sola volta, al momento della prima richiesta, e rimangono residenti in memoria, pronte per servire altre richieste fino a quando non vengono chiuse, con ovvi vantaggi di prestazioni (solo la prima richiesta risulta un pò più lenta nel caricamento, ma le successive vengono evase più velocemente).

6.1.3.1 Ciclo di vita di una servlet

Il ciclo di vita di una servlet si concentra su tre metodi fondamentali: **init()**, **service()** e **destroy()**.

- **init()**: Questo metodo viene chiamato una sola volta, subito dopo la sua istanziazione.
- **service()**: Questo metodo è incaricato di gestire le richieste effettuate dal client e ovviamente potrà iniziare il suo lavoro esclusivamente dopo la chiamata di *init()*.
- **destroy()**: questo metodo segna la chiusura della servlet, è qui che si effettuano eventuali salvataggi di informazioni utili ad un prossimo caricamento della servlet.

6.2 implementazione delle pagine JSP

Le pagine JSP devono essere salvate con l'estensione *.jsp*. Esse devono essere scritte il linguaggio HTML racchiudendo tra $<\%$ e $\%>$ le varie istruzioni in linguaggio java.
La prima volta che si effettua la richiesta del file, quest'ultimo viene compilato creando una servlet, che sarà archiviata in memoria (per servire le richieste successive); solo dopo questi passaggi viene elaborata la pagina HTML che viene mandata al browser.
Ad ogni richiesta successiva il server, dopo aver controllato eventuali modifiche nelle pagine jsp, richiama la servlet già compilata, altrimenti si occupa di eseguire nuovamente la compilazione e la memorizzare la nuova servlet.

Analizziamo per esempio l'homepage della nostra applicazione web senza soffermarci troppo su cosa effettivamente esegue:

Listing 6.1: *homepage index.jsp*

```
1
<%String NomeAzienda="";%>
3 <%@ include file="header.html" %>
<div id="content">
5
<%@ page import="mioApache.*"%>
7 <%@ page import="java.net.*;" %>
<%
9 if(NomeAzienda.compareTo("admin")==0){
        DbSQL SQL=new DbSQL();
11      String utenti=SQL.printUtenti();
        %>
13 <h1>LISTA UTENTI</h1>
<%
15      out.print(utenti);
        }
17      else{
        int connessione= Apache.primoLogin();
19
        if(connessione==-1){
21      //la connessione e' andata a buon fine

23 %>
<h1>ATTENZIONE!! SERVER WEB NON CONNESSO ALLA MAM DI VODAFONE!</h1>
```

```
25 <%
                    }
27                  else{
          //la connessione NON e' andata a buon fine
29
   %>
31 <h1>BENVENUTO ALLA CONSOLE DELLA MAM DI VODAFONE!!</h1>
   <%
33
                              }
35                  }
   %>
37 </div>


39
   <%@ include file="footer.html" %>
```

In questa pagina inizialmente viene inizializzata una stringa per poi includere successivamente una pagina HTML denominata ***header.html*** (in essa saranno presenti il tag dell'*header* e del *navigation* tralasciando quello del *content*). Più avanti la pagina includerà dei pachetti java per effettuare delle istruzioni scritte in linguaggio java. Se prestiamo attenzione alla seconda condizione **if** (*if(connessione==-1)*), verrà visualizzato della pagina finale un tag di tipo $< h1 >$ differente a seconda del risultato della condizione.

Tralasciando la parte di codice presente nelle pagine *header.html* e *footer.html*, il codice sorgente nella pagina web che viene visualizzato dall'utente sarà:

Listing 6.2: *codice sorgente della pagina index.jsp*

```
...
2
<h1>BENVENUTO ALLA CONSOLE DELLA MAM DI VODAFONE!!</h1>
4
...
```

Le principali direttive dunque sono:

- `<%@ include file="nomefile"%>`

  Include nella pagina web un altro file

- `<%@ page import="nomepacchetto"%>`

  Importa un pacchetto java da poter utilizzare all'interno della pagina jsp.

- `out.print(stringa)`

6.3 Sitemap del progetto

Il nostro progetto è quindi gestito tutto in JSP, dato che gran parte dell'architettura software è stata scritta in java. Nella figura 6.1 possiamo osservare come si dispone la nostra interfaccia web definendo così tutte le possibile operazioni che l'utente finale può fare.

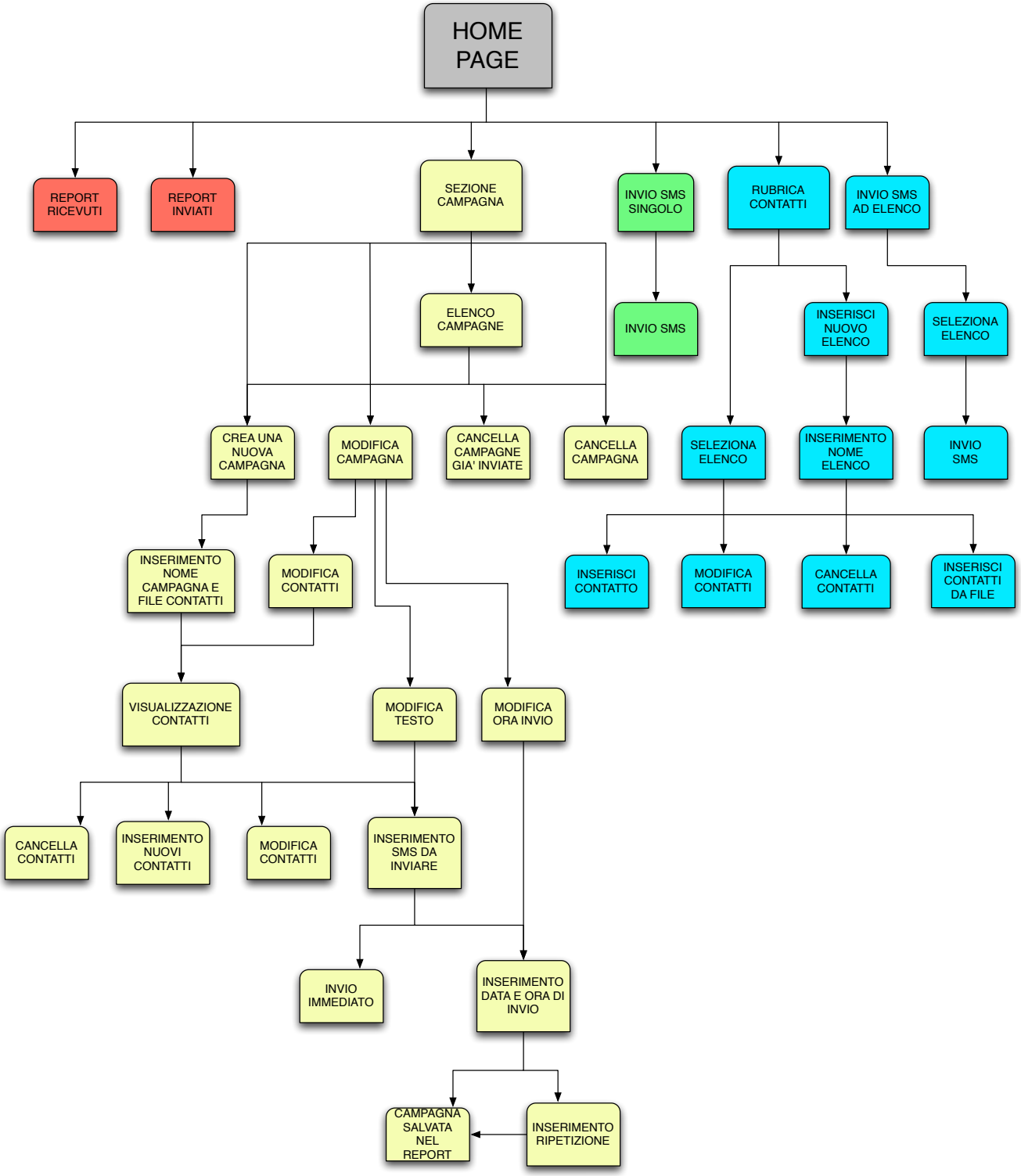

Figura 6.1: *Sitemap del progetto*

Risulta evidente che le principali sezioni rispecchiano molto i tre scopi principali del progetto rendendolo così molto pratico e alla portata di tutti.
Si può notare ulteriormente come sia possibile, sia nella sezione **campagna** e sia in quella **elenchi**, inserire contatti telefonici tramite file. Il formato del file può essere sia in *xls* che in *csv*, rendendo possibile un inserimento facile e veloce anche chi non dispone di prodotti a pagamento come Microsoft Office®.
Nel file devono essere presenti quattro colone, di cui l'ultima facoltativa, le quali rappresentano rispettivamente **Nome**, **Cognome**, **Numero** di telefono, **Data di nascita**. L'ultimo attributo deve essere in formato *aaaa-mm-gg*. Esso è stato inserito semplicemente per poter mandare al cliente gli auguri di buon compleanno; più avanti spiegheremo come.

La sezione **campagna** risulta molto più articolata in quanto offre la possibilità di stabilire l'orario di invio degli SMS in una data futura con l'aggiunta eventuale di una possibile ripetizione di invio della campagna. Questo è stato sviluppato soprattuto per invio di SMS con scopo promozionale in quanto è possibile pianificare un vero e proprio marketing pubblicitario via SMS.
Per l'utente invece che vuole spedire un SMS ad un solo numero in maniera facile e veloce su ogni pagina web, nella colonna navigazione è presente un riquadro dove è possibile inserire il numero di cellulare del destinatario e il testo del messaggio inviando un SMS in maniera immediata.

Per rendere il messaggio più personalizzato possibile, nel caso in cui all'interno del messaggio si inseriscano i tag $< nome >$ e $< cognome >$ durante la fase di invio del messaggio, la servlet di java li sostituirà con il rispettivo nome e cognome salvati nel database associati al quel numero. Ovviamente è possibile utilizzare i tag sia contemporaneamente sia solo uno dei due.

Nelle tre sezioni di invio degli SMS (una presente nella sezione campagna, una nella sezione *Rubrica contatti* e una per l'invio di un singolo SMS) viene richiamati diversi metodi della classe *Apache* (rispettivamente *invioSMSCampagna*, *invioSMSdaElenco*). La differenza consiste semplicemente nell'eseguire differenti query al database per selezionare i contatti destinatari relativi alla campagna e/o all'elenco stabilito.
All'interno dei seguenti metodi verrà richiamato il metodo di invio SMS *invioMsg* tante volte quanti sono i contatti all'interno della campagna e/o dell'elenco.
Il metodo di invio di un singolo messaggio richiamerà immeditamente il metodo *invioMsg*, in quanto non deve eseguire nessuna query al database avendo già il numero del destinatario e il testo, dato che sono stati inseriti dall'utente finale per mezzo della nostra interfaccia grafica.
I messaggi con l'**invio programmato** verranno inviati per mezzo di una servlet, spiegata nel dettaglio nel capitolo 6.4.
Il report dei messaggi inviati e ricevuti sarà ovviamente identico per ogni utente e saranno presenti indistintamente tutti messaggi inviati e ricevuti da ogni utente. Risulta impossibile sapere per chi è destinato un messaggio ricevuto dato che gli utenti sono associati allo stesso Server MAM e quindi allo stesso numero breve.
Viceversa, per il fatto della personalizzazione dell'SMS, è impossibile stabilire l'autore dei messaggi inviati.

6.4 Servlet in Background

Utilizzando il maccanismo *servlet-server web* abbiamo creato un servlet che viene eseguita durante l'attivazione del web server utilizzato. Lo scopo di questa servlet consiste nell'invio delle campagne con invio programmato nella data e ora prestabilita.
Questo servlet tuttavia non svolge alcuna istruzione o query al nostro database; essa si impegna a lanciare un thread denominato *MainThread* il quale possiede un ciclo while infinito, per cui rimane sempre attivo. Il thread *MainThread* si occupa di verificare nella tabella campagne ogni 20 secondi se sono presenti tuple con lo stato *"da inviare"* e con una data antecendente a quella in cui il thread esamina i dati.
Nel caso in cui ce ne siano esso si occupa a inviarle ed ad aggiornare lo stato e la data di invio della campagna inviata. Inoltre verifica se le campagne appena inviate devono essere essere spedite ulteriolmente altre volte a causa della presenza di dati nei campi di *ripetizione*. In quel caso aggiorna la data prevista di invio lasciando lo stato della campagna in modalità *"da inviare"*, in modo che, alla data prevista del nuovo invio, lo stesso thread effettuerà la spedizione.

Per poter avviare una servlet all'avvio del server web basta aggiungere i seguenti tag al file di configurazione *web.xml* presente nella cartella *Webcontent/WEB-INF*:

```
...
 <servlet>
    <servlet-name>ServletInitializer</servlet-name>
    <servlet-class>mioApache.ServletInitializer</servlet-class>
    <load-on-startup>1</load-on-startup>
 </servlet>
 ...
```

In questo modo eseguiremo la servlet di nome *ServletInizializer* la quale eseguira le seguenti istruzioni:

```
public class ServletInitializer extends HttpServlet
{
  public void init() throws ServletException
    {
          /// Automatically java script can run here
          System.out.println("***********");
          System.out.println("*** Servlet
          Initialized successfully ***..");
          System.out.println("**********");
    }

    public void service(HttpServletRequest request,
     HttpServletResponse response)
    throws ServletException, IOException{
```

```
MainThread background= new MainThread();
background.start();
      }
}
```

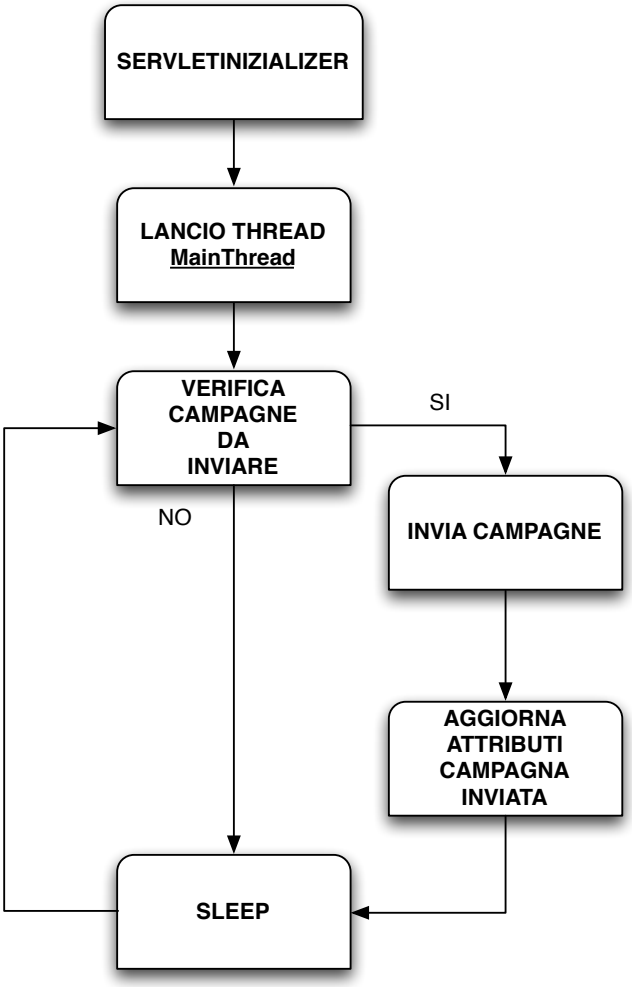

Figura 6.2: *flusso di esecuzione della servlet e del thread MainThread*

Capitolo 7

Controlli Javascript

Javascript è un linguaggio di scripting orientato ad oggetti comunemente usato nei siti web. Come può sembrare dal nome è possibile pensare che il linguaggio di Javascript sia identico a quello di Java. In realtà non è proprio così: ha una sintassi molto somigliante (derivata in entrambi i casi dal linguaggio C); le loro semantiche sono piuttosto diverse, e in particolare i loro *object model* non hanno relazione e sono ampiamente incompatibili.

La carattestica principale di Javascript è che il codice non viene compilato ma interpretato. Viene così eseguito dal lato client (a differenza del codice JSP) nel quale l'interprete è il browser web che si sta utilizzando. Il vantaggio di questo approcio è che, anche con la presenza di script particolarmente complessi, il server non viene sovraccaricato a causa delle richieste dei client. Uno svantaggio viene evidenzato nel caso di script che presentino un sorgente particolarmente grande; il tempo dello scaricamento può diventare abbastanza lungo.

7.1 utilizzo nei form

Nel nostro progetto il linguaggio Javascript viene utilizzato solo per verificare i campi input di un form prima di eseguire un post alla pagina successiva evitando di generare errori. Essenzialmente i controlli via javascript sono:

- controllo che gli input text non siano vuoti
- controllo delle date
- funzione calendario

I primi due controlli devono essere eseguiti quando l'utente esegue il post, quindi quando l'utente seleziona l'input *submit*.
Se il form è definito in questo modo:

```
<form name="nomeform" method="post" action="paginadidestinazione">
<input type="text" name="text1">" />
```

```
<input type="text" name="text2" /><BR>
<input type="submit" value="avanti"/>

</form>
```

Bisognerà aggiungere nel tag *input submit* il campo **onclick** con la funzione di javascript da eseguire.

```
...
<input type="submit" value="avanti" onclick="return controllo()"/>
...
```

La funzione javascript *controllo()* eseguirà le sue istruzioni per poi restituire *true* se tutta va per il verso giusto, altrimenti restituirà un *false*.
Di solito, nel caso in cui i controlli non venissero rispettati, l'utente vedrà un *alert* comparire nel browser in cui spiegherà l'errore e il motivo del blocco. Le funzioni di Javascript saranno differenti a seconda di cosa si vuole controllare.

7.1.1 Controllo input text

È opportuno sempre controllare che i campi di input non siano vuoti in quanto, nelle pagine richiamate dal form, saranno sempre utilizzati: si rischierebbe che una variabile venisse definita **null** lanciando quindi delle eccezioni a causa di un'eventuale utilizzo delle stesse. Un'esempio è la funzione **compareTo** di java: essa lancia un errore se una delle due stringhe da confrontare risulta *null*.
In javascript, per controllare se una variabile passata attraverso un post risulta *null*, basta semplicemente controllare se il campo *length* o il campo *value* della variabile sia diverso da 0.

Supponiamo per esempio di essere nella pagina adibita alla creazione di una nuova campagna. In questa pagina sono presenti due input: uno di tipo **text** in cui si inserisce il nome della nuova campagna; e uno di tipo **file** in cui si inserisce la path locale del file nel quale sono presenti i contatti che vogliamo inserire nella campagna. Il tag form in cui sono racchiusi i nostri input viene chiamato *"nomecampagna"* invece l'input file è denominato *"fileuploaded"*. La nostra funzione javascript sarà così composta:

```
function controllocampi1(){
var nomec=document.campagna.nomecampagna.value;
if (nomec==0){
alert("Inserire il Nome Campagna");
document.campagna.nomecampagna.focus();
return false;
}
else{
```

```
if(document.campagna.fileuploaded.value==0){
alert("Inserire il file per l'upload");
return false;
}
else return true;

}
}
```

Semplicemente essa è composta da due blocchi *if* nei quali si verificano, uno dopo l'altro, se due input passati non sono nulli.
La sintassi di un valore di un input è *document.nomeform.nomeinput.value*.
La funzione **alert** restituisce una finestra di popup che vedrà l'utente sul suo browser web in cui è scritto il messaggio d'errore.

7.1.2 funzione calendario

Quando si inserisce o si modifica la data di nascita in un contatto sono presenti degli input di testo con **id="DPC_calendar1b_YYYY-MM-DD"**. Questo id si associa al Javascript importato dalla pagine jsp di nome *datepickercontrol* il quale fa comparire all'utente la finestra calendario per selezionare al meglio la data. In questo modo l'utente non potrà sbagliare il formato della data e avrà un aiuto grafico per selezionare la data corretta.

7.1.3 controllo delle date

Risulta opportuno controllare se le date inserire dall'utente, oltre ad avere il formato corretto, siano anche corrette concettualmente. Ad esempio risulta impossibile che una data di nascita di un contatto sia posteriore alla data attuale, o viceversa non si può programmare l'invio di una campagna con data e ora passate. Ecco quindi che anche in questo caso, l'utilizzo delle funzioni javascript possono aiutare a non inserire nel nostro software dati concettualmente sbagliati.
In questo caso la funzione di javascript risulta un pò più complessa, anche se è sempre formata da semplici condizioni di *if*.

Analizziamo per esempio la funzione che controlla la data di invio di una campagna promozionale.

```
1         function controlloprog(){
                var controllo= false;
3               var radiobutton = false;
                var bottone = document.prog.prog;
5               for(var i=0; i<bottone.length; i++) {
                  if(bottone[i].checked) {
7                   radiobutton = true;
                  }
9                 if(bottone[1].checked)
                          controllo=true;
11              }
```

```
                if(!radiobutton) {
                  alert("Deve essere selezionata almeno un\'modalita' di "
                   +"programmazione.");
                  return false;
                }

                if(controllo){
                var scheduled_date=document.prog.calendar1b.value+" "+
                +document.prog.ora.value+":"+document.prog.minuti.value;
                var date = new Date();
                var my_date=date.getFullYear()+"-";
                var giorno=date.getDate();
                var mese=date.getMonth()+1;
                var ora=date.getHours();
                if(ora<10)
                        ora="0"+ora;
                var minuti=date.getMinutes();
                if(minuti<10)
                        minuti="0"+minuti;
                if(giorno<10)
                        giorno="0"+giorno;
                if(mese<10)
                        mese="0"+mese;
                var my_time=ora+":"+minuti;
                my_date=my_date+mese+"-"+giorno+" "+my_time;
                var ciao="data attuale: "+my_date+"\ndata programmata: "
                +scheduled_date+"\n";

                if(scheduled_date<my_date){
                        alert(ciao+"ERRORE!\nData antecedente a quella attuale!");
                return false;
                }
                else{ //alert(ciao+"data ok");
                                                        return true;
                                }
        }
        }
```

Per confrontare la data e l'ora inserita con quella attuale la funzione *controlloprog* utilizza i metodi della classe Date per utilizzare la data e l'ora in cui viene eseguito il post. Una piccola curiosità consiste anche nel fatto che questa classe conta i mesi da 0 a 11; per cui siamo costretti a incrementerare di una unità il valore del metodo *getMonth()* per confrontare con il mese inserito (di valore da 1 a 12).

Capitolo 8

Progetto multi utente

Per rendere il progetto accessibile a molte aziende è oppotuno renderlo in modalità multi-utente, ossia creare una parte dell'interfaccia web accessibile solo per chi possiede delle credenziali.
Con l'esecuzione di un login è possibile stabilire l'identità dell'utente e garantire l'accesso ai servizi della nostra piattaforma.

Bisogna quindi creare una pagina di *login*, *logout*, ed eventualmente una pagina di modifica dei dati.

Esistono differenti modi per effettuare l'accesso ad un sito web in modalità protetta:

- utilizzando i **cookies**, file salvati in locale nel client dove il browser può salvare e prendere varie informazioni, quali ad esempio l'username e la password.

- Utilizzando l'oggetto implicito **Session**: la nostra scelta.

8.1 Utilizzo delle sessioni

L'oggetto implicito **Session** gestisce appunto le informazioni a livello di sessione, relative ad un singolo utente a partire da suo ingresso alla sua uscita. Esso appunto risolve il problema di mantenere informazioni legate all'utente lungo tutto il tempo di visita del sito.
È possibile quindi creare applicazioni che riconoscono l'utente nelle varie pagine del sito, che tengono traccia delle sue scelte e dei suoi dati. È importante sapere che le sessioni, a differenza dei *cookies*, vengono memorizzate sul server e non devono però essere abilitati per poter memorizzare il così detto **SessionID** che consente di riconoscere il browser e quindi l'utente nelle fasi successive.
I dati di sessione sono quindi riferiti e riservati ad un utente a cui viene creata un'istanza dell'oggetto *Session* e non possono essere utilizzati da sessioni di altri utenti.

8.1.1 Memorizzazione dei dati

Per memorizzare i dati all'interno dell'oggetto *session* è sufficiente utilizzare il metodo **setAttribute** specificando il nome dell'oggetto da memorizzare e una sua istanza. Per esempio, per memorizzare il nome dell'utente al suo ingresso alla pagina ed averlo a disposizione in seguito è sufficiente eseguire *session.setAttribute(nomeUtente, nome)* a patto che nome sia un oggetto di tipo stringa che contenga il nome dell'utente.
Per poter effettuare la memorizzazione dei dati è opportuno controllare prima se l'*username* e la *password* dell'utente siano corretti: per questo basta effettuare una verifica nel database all'interno della tabella *Azienda*.

La nostra pagine di login infatti, una volta verificata la veridicità dell'*username* e *password*, esegue queste instruzioni:

```
session.setAttribute("nomeUtente",strUser);
response.sendRedirect("../index.jsp");
```

dove *strUser* è l'username dell'utente.
Con il metodo *sendRedirect* dell'oggetto *response* si esegue in redirect alla pagina inserita come argomento (in questo caso l'homepage).

8.1.2 Leggere il contenuto di una variabile di sessione

La lettura di una variabile di sessione precedentemente memorizzata è possibile grazie al metodo **getAttribute** che ha come unico ingresso il nome della variabile di sessione con cui avevamo memorizzato il dato che ci interessa reperire. *Session.getAttribute("nomeUtente")* restituisce il nome dell'utente memorizzato come visto in precedenza; se non viene trovata nessuna corrispondenza con il parametro dato in ingresso, restituisce *null*.
In ogni pagina bisognerà quindi controllare se sia presente una variabile all'interno dell'attributo *nomeUtente* altrimenti si eseguirà un redirect alla pagina di login.

Nella pagina *header.html* (importata in ogni pagina) infatti sono presenti le seguenti istruzioni:

```
   NomeAzienda +=session.getAttribute("nomeUtente");
if(NomeAzienda.compareTo("null")==0){

session.removeAttribute("nomeUtente");
response.sendRedirect("Session/login.jsp");

}
```

8.1.3 Logout

Il logout dal nostro sito rimuove semplicemente l'attributo *nomeUtente* dall'oggetto Session utilizzando il metodo *removeAttribute* ed eseguendo successivamente un redirect all'homepage (che a sua volta eseguirà un redirect nella pagina di login):

```
session.removeAttribute("nomeUtente");
  response.sendRedirect("../index.jsp");
```

8.1.4 ulteriori metodi di Session

A seguito riportiamo altri metodi utili dell'oggetto *Session* che non abbiamo utilizzato:

- **getAttributeNames()**: restituisce un oggetto di tipo enumerativo di stringhe contenente i nomi di tutti gli oggetti memorizzati nella sessione corrente.
- **getCreationTime()**: restituisce il tempo (in millisecondi dalla mezzanotte del primo gennaio del 1970) di quando è stata creata la sessione.
- **getId()**: restituisce una stringa contenente il sessionID che come detto permette di identificare univocamente una sessione.
- **getLastAccesedTime()**: restituisce il tempo (in millisecondi dalla mezzanotte del primo gennaio del 1970) dall'ultima richiesta associata alla sessione corrente.
- **getMaxInactiveInterval()**: restituisce un valore intero che corrisponde all'intervallo massimo di tempo tra una richiesta dell'utente ad un'altra della stessa sessione.

Capitolo 9

Perfezionamento grafica con CSS

Dietro il semplice acronimo CSS (Cascading Style Sheets - Fogli di stile a cascata) si nasconde uno dei fondamentali linguaggi standard del W3C. La sua storia cammina su binari paralleli rispetto a quelli di HTML, di cui vuole essere l'ideale complemento. Da sempre infatti HTML dovrebbe essere visto semplicemente come un linguaggio strutturale, alieno da qualunque scopo attinente la presentazione di un documento. Per questo obiettivo, ovvero arricchire l'aspetto visuale ed estetico di una pagina, lo strumento designato sono appunto i CSS. Si vuole coì **separare il contenuto dalla presentazione**.[5][1]

9.1 Utilità del CSS

Chi conosce un minimo di HTML è pienamente consapevole dei limiti di questo linguaggio sul lato della pura presentazione. A dire il vero, non è giusto parlare di limiti: HTML non è nato per questo. È stato però piegato a fare cose che intrinsecamente non era in grado di fare.
Finalmente, ad esempio, si può dare al testo delle nostre pagine un aspetto da word-processor: non solo con il colore o i font che preferiamo, ma con un sistema di interlinea pratico e funzionale, con le decorazioni che desiderate, riuscendo a spaziare lettere e parole, impostando stili diversi per titoli e paragrafi, sfruttando i benefici dell'indentatura o della giustificazione.
Se si vuole cambiare in futuro l'immagine di sfondo, o semplicemente camabiare colore dei titoli o font del corpo sel sito, non dobbiamo più andare a modificare una per una 300 pagine in quanto i CSS sono separati dal documento: basterà quindi modificare solo il foglio di stile.

9.2 Layout

Il layout utilizzato per le nostra pagine web è quello *a due colonne*. La semplicità dei fogli di stile CSS è quella di caratterizzare la pagina web secondo la filosofia **table-less**.
Chi si è avvicinato da poco ai fogli di stile, e più in generale alla creazione dei siti web, troverà facile e logico l'uso dei css per la creazione dei layout a tal punto che le tabelle serviranno solo

per dati tabellari.

La struttura della nostra pagina web sarà divisa quindi da 4 strutture, entrambi definitivi dai tag *div* ma con diversi *id*. La sua definizione è **generic block-level element** ossia contenitore generico block level. Il fatto che sia un elemento block-level ci garantisce il fatto che possa contenere qualsiasi tipo di elemento html. Inoltre, la sua presentazione naturale (quindi senza fogli di stile) è totalmente neutra: infatti questo elemento si presenta di default senza margini, bordi o padding.

9.2.1 header

L'header si estende orizzontalmente per tutto lo spazio a disposizione del layout. Il nostro header riporta il nome della piatafforma con i loghi dell'azienda e dell'operatore telefonico utlizzato.

9.2.2 navigazione

La navigazione, anche detta menu è una sezione indispensabile di ogni sito, in quanto permette di accedere ai contenuti. È possibile quindi spostarci dalla sezione **campagne** o dalla sezione **elenchi**, visualizzare il report dei messaggi o aggiungere nuovi contatti.

9.2.3 sezione dei contenuti

In questo blocco saranno presenti i contenuti relativi alla pagina e alla sezione in cui siamo. Qui ci sarà il vero corpo della nostra piattaforma web.

9.2.4 footer

Il footer è generalmente una piccola sezione disposta a fondo pagina e contiene informazioni sullo sviluppatore del sito, sul copyright, i contatti di posta elettronica ed eventualmente indirizzo e numero di telefono se il sito riguarda un'azienda.

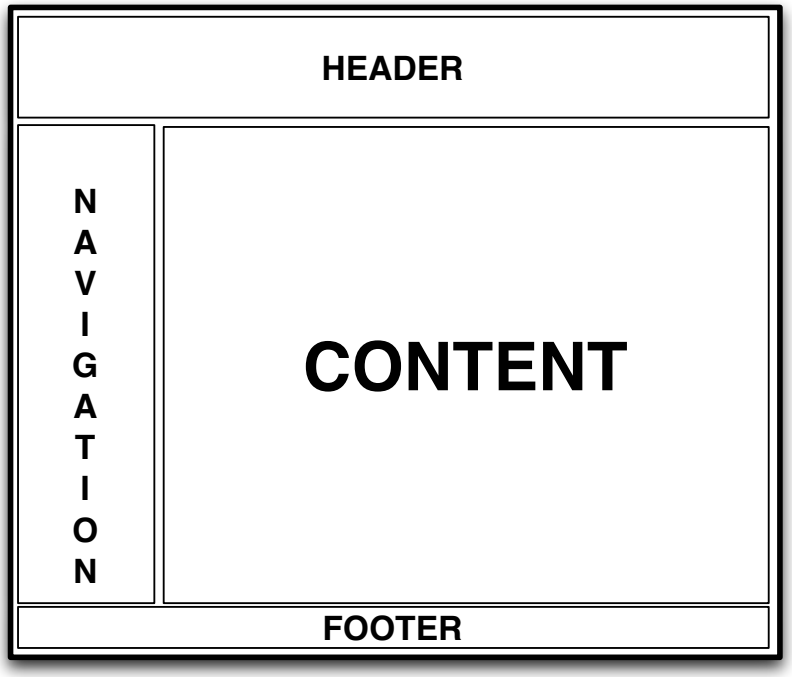

Figura 9.1: *Layout CSS utilizzato*

Figura 9.2: *Homepage della piattaforma web*

Appendici

Appendice A

Principali comandi della MAM Vodafone

Parametro	Valore	Descrizione
MsgId	numeber(15)	Identificatore del messaggio
Orig	*phone #* *dafult*	mittente del messaggio (default seleziona i messaggi inviati dal cliente stesso)
Type	 *sent* *receivednew* *all*(default)	Tipo del messaggio: -messaggi inviati -messaggi ricevuti -tutti i messaggi
FromDate	date	Data GMT inizio del periodo selezionato.
ToDate	date	Data GMT fine periodo selezionato.
Encoding	*7bit* 8bit 16bit	Codifica del Body.
AllFields	boolean	Se *true*, i report prodotti conterranno tutti i campi disponibili(indipendentemente dalle impostazione date nel comando si *Set*)

Tabella A.1: *Parametri di chiamata del comando Report*

Parametro	Valore	Descrizione
MsgId	numeber(15)	Identificativo del messaggio
Status	*tipo di stato*	Stato del messaggio (vedi cap.2.4.3)
Date	date	Data di trasmissione/ricezione del messaggio dal server MAM nel formato yyyy/mm/dd. hh:mm:ss
LastUpdate	date	Data dell'ultima variazione di stato avvenuta (nello stesso formato del parametro precedente)
Orig	*phone # dafult*	mittente del messaggio
Dest	*phone # dafult*	destinatario del messaggio
Prefix	*nonOPI*	prefisso applicato al numero breve del cliente. Questo parametro richiede l'abilitazione del cliente alla Vodafone
Ext	*number*	estensione applicata al numero breve del cliente; si riferisce al numero mittente per i messaggi MT, al destinatario per i messaggi MO. Anche questo parametro richiede l'abilitazione.
UserDataHeader	String	campo Userdata del messaggio rappresentato a 8 bit.
Body	*string*	Corpo del messaggio rappresentato in base alla codifica indicata da Encoding.
Encoding		modalitá di codifica.

Tabella A.2: *Parametri di ritorno del comando Report*

Parametro	Valore	Descrizione
Dest	phone	Numero del destinatario (numero di 12 cifre che incomincia con il prefisso nazionale (ad esempio 39) e il numero telefonico)
Prefix		Prefisso da aggiungere al numero breve usato come mittente (vedi cap.2.4.1)
Ext		Estensione da aggiungere al numero breve usato come mittente(vedi cap.2.4.1)
Orig	default o alias	Specifica il mittente del messaggio, puó essere il numero breve del client(default) o l'alias impostato tamire il comando Set (2.4.1)
UserdataHeader	string	Campo usato per l'invio di dati speciali come ad esempio loghi e suonerie, deve essere sempre rappresentato con codifica a 8 bit. La lunghezza massima é di 140 Byte(280 cifre hex).
Body	string	Corpo del messaggio formattato in base alla codifica presceta (di default é do 7 bit). La lunghezza massima é di 160 caratteri.
ValidyPeriod	number	Si puó definire il periodo di validitá del messaggio. Il valore é espresso in secondi ed é compreso fra 0 e 259200 secondi con qualche arrotondamento all'unitá superiore.
Class	number	Classe del messaggio. Indica come verrá trattato il messaggio dal ricevente: 0 Flash: visualizza immediatamente su display 1 ME specific: memorizza il messaggio nel telefono 2 SIM specifico: memorizza preferibilmente su SIM card 3 specifico TE: comunica con un terminale esterno collegato al ricevente (ad esempio un PC collegato a modem GSM)
Alert	boolean (default=true)	Modalitá di avviso ricezione messaggio su terminale ricevente.

Tabella A.3: *Parametri del comando Send*

Appendice B

Codici Java

Listing B.1: *interfaccia MAMController.java*

```
package it.telerete.apsmobile.api.controller;

import it.telerete.apsmobile.api.model.SMSReceived;

import java.io.BufferedReader;
import java.net.Socket;
import java.util.List;

public interface MAMController {



        /**
         * Metodo che stampa a video la riposta del server MAM. Viene utilizzata nei
         * metodi di login, logout, ricezione e invio messaggio.
         *
         * @param inFromServer
         *            BufferedReader
         * @throws Exception
         */
        public  void stampaRisposta(BufferedReader inFromServer) throws Exception;

        /**
         * metodo che si logga tramite telnet alla MAM
         *
         * @param String
         *            l'host da effettuare la connessione telnet
         * @param int numero di porta
         * @param user
         *            username del telnet
         * @param pwd
         *            password del telnet
         * @return Socket socket di connesione
         * @throws Exception
         */
        public void login()     throws Exception;

        /**
         * Invio di un SMS tramite la MAM
         *
```

```
42          * @param numero
            *            Socket di connessione
44          * @param numero
            *            numero di cellulare
46          * @param Msg
            *            messaggio da inviare
48          *
            * @throws Exception
50          */
        public void invioMsg( String numero, String Msg)
52                      throws Exception;


54

        /**
56       * Logout della connessione con la MAM
         *
58       * @param clientSocket
         *            Socket di connessione
60       * @throws Exception
         */
62      public void logout() throws Exception;

64      /**
         * Report dei messaggi inviati con setcode=delivered
66       *
         * @param Socket
68       *            di connessione
         * @return Lista dei messaggi inviati di tipo SMSReceived
70       */
        public void reportMsgInviati()
72                      throws Exception;

74      /**
         * Report dei messaggi ricevuti con setcode=receivednew
76       *
         * @param Socket
78       *            di connessione
         * @return Lista dei messaggi ricevuti di tipo SMSReceived
80       */
        public void reportMsgRicevuti()
82                      throws Exception;

84      public boolean isConnected();

86  }
```

Listing B.2: *implementazione di MAMController.java*

```
    package it.telerete.apsmobile.api.controller;
2
    import java.io.*;
4   import java.net.*;
    import java.util.List;
6   import java.util.ArrayList;

8   import java.util.StringTokenizer;

10  import javax.annotation.PostConstruct;
    import javax.annotation.PreDestroy;
12
    import org.apache.log4j.Logger;
```

```
14 import org.springframework.beans.factory.annotation.Value;
   import org.springframework.stereotype.Service;
16
   import it.telerete.apsmobile.api.model.SMSReceived;
18
   @Service("mamController")
20 public class MAMControllerImpl implements MAMController {

22 private Logger logger = Logger.getLogger(this.getClass());

24 private static Socket clientSocket;

26 @Value("#{apsMobileProperties['MAM.host']}")
   private String MAMhost;
28
   @Value("#{apsMobileProperties['MAM.port']}")
30 private int MAMport;

32 @Value("#{apsMobileProperties['MAM.user']}")
   private String MAMuser;
34
   @Value("#{apsMobileProperties['MAM.password']}")
36 private String MAMpassword;

38



40
   /**
42  * Stampa a video le righe che manda il MAM
    */
44 public void stampaRisposta(BufferedReader inFromServer) throws Exception {

46
        String risposta = "";
48
        // Cicla fintanto che la MAM non fornisce una risposta
50      while (risposta.compareTo("") == 0) {
                risposta = inFromServer.readLine();
52              logger.info("Risposta␣MAM:␣"  + risposta);
        }
54
}
56
   /**
58  * Esegue il login
    */
60 @PostConstruct
   public void login()
62 throws Exception {

64
        logger.info("MAMhost:␣" + MAMhost);
66      logger.info("MAMport:␣" + MAMport);
        logger.info("MAMuser:␣" + MAMuser);
68      logger.info("MAMpassword:␣" + MAMpassword);

70      clientSocket = new Socket(MAMhost, MAMport);

72      // creo un OutputStream per la richiesta
        DataOutputStream outToServer = new DataOutputStream(
74              clientSocket.getOutputStream());
```

```
76          // creo un BufferReader per la risposta
            BufferedReader inFromServer = new BufferedReader(
78                                  new InputStreamReader(
                                    clientSocket.getInputStream()));
80
            stampaRisposta(inFromServer);
82          stampaRisposta(inFromServer);
            // aspetto la comparsa a schermo di "username:"
84          outToServer.writeBytes(MAMuser + "\n"); // invio username
            stampaRisposta(inFromServer);
86
            // aspetto la comparsa a schermo di "password:"
88          outToServer.writeBytes(MAMpassword + "\n"); // invio password
            stampaRisposta(inFromServer);
90          Thread.sleep(1000);

92          outToServer.writeBytes("Set\n");

94          outToServer.writeBytes("Mode=sync\n");

96          outToServer.writeBytes("Storage=true\n");

98          // da mail bisogna attendere tra i due invii qualche secondo
            Thread.sleep(5000);
100         outToServer.writeBytes("\n");
            Thread.sleep(5000);
102


104         String risposta = "";
            risposta = inFromServer.readLine();
106
            String msg = "";
108
            int k=0;
110         while (risposta.compareTo("OK") != 0) {

112                 logger.info("dentro␣ciclo␣" + k + "␣while␣risposta:" + risposta);
                    logger.info("dentro␣ciclo␣" + k + "␣msg:␣" + msg);
114
                    if (risposta.compareTo("") != 0)
116                         msg = msg + risposta + "\n";
                    risposta = inFromServer.readLine();
118
                    }
120
            logger.info("Risposta:␣" + risposta);
122
            logger.info("Username␣e␣Password␣corretti");
124
            logger.info("ClientSocket:␣" + clientSocket);
126
}
128
/**
130  * Esegue il logout
 */
132 @PreDestroy
public void logout() throws Exception {
134
            // creo un OutputStream per la richiesta
```

```
136         logger.info("Procedura␣logout");
138         if (clientSocket.isConnected()){
140                 logger.info("Eseguo␣Logout");
142                 DataOutputStream outToServer =
                                new DataOutputStream(
144                                         clientSocket.getOutputStream());
                    outToServer.writeBytes("Logout\n");
146                 //Thread.sleep(2000);
148                 outToServer.writeBytes("\n");
                    // stampaRisposta(inFromServer);
150
                    clientSocket.close(); // chiudo la socket
152
                    logger.info("Logout␣eseguito");
154
            } else {
156
                    logger.info("Logout␣non␣necessario");
158
            }
160
    }
162
    /**
164  * Invio messaggio
     */
166 public void invioMsg(String numero, String msg) throws Exception {
            try {
168                 logger.info("**********␣Invio␣Messaggio␣********");
170                 boolean newConn = false;
                    logger.info("clientSocket:␣" + clientSocket);
172                 if (clientSocket!=null){
                            logger.info("clientSocket:s");
174                         logger.info("Socket␣chiuso:␣" + clientSocket.isClosed() );
                            logger.info("Socket␣connesso:␣" + clientSocket.isConnected());
176                 }
178                 if ( (clientSocket==null)||(clientSocket.isClosed()) ){
                            logger.info("Connessione␣per␣invio");
180                         login();
                            newConn = true;
182                 }
184                 logger.info("Socket␣After␣chiuso:␣" + clientSocket.isClosed() );
                    logger.info("Socket␣After␣connesso:␣" + clientSocket.isConnected());
186
                    logger.info("ClientSocket:␣" + clientSocket);
188                 logger.info("Impostazione␣Invio␣messaggio␣a␣" + numero
                                        + "␣con␣testo␣\"" + msg + "\"");
190
                    // Creo un OutputStream per la richiesta
192                 DataOutputStream outToServer = new DataOutputStream(
                                        clientSocket.getOutputStream());
194
                    // Creo un BufferReader per la risposta
196                 BufferedReader inFromServer = new BufferedReader(
```

```
                                new InputStreamReader(
198                                     clientSocket.getInputStream()));

200
                // Eseguo i comandi necessari per mandare i messaggi
202
                // effettuo l'invio del messaggio
204             outToServer.writeBytes("Send\n");
                outToServer.writeBytes("Dest=" + numero + "\n");
206
                outToServer.writeBytes("Body=" + msg + "\n");
208

210             // da mail bisogna attendere tra i due invii qualche secondo
                Thread.sleep(500);
212             outToServer.writeBytes("\n");
                stampaRisposta(inFromServer);// MsgID= nnnnnnnnnn
214
                logger.info("Invio messaggio (MAM)  a Dest= " + numero + " Body= "
216                                     + msg + "\t\t");
                stampaRisposta(inFromServer);// OK
218
                if(newConn){
220                     logger.info("Logout invio newConn");
                        logout();
222             }

224             logger.info("Socket chiuso: " + clientSocket.isClosed() );
                logger.info("Socket connesso: " + clientSocket.isConnected() );
226
                logger.info("********** Messaggio Inviato (" + numero + " / "
228                                     + msg + ")*********\n\n\n");

230         } catch (Exception e) {
                logger.error("Errore invio messaggio: " + e.getLocalizedMessage());
232             throw e;
            }
234 }

236
/**
238  * Report dei messaggi ricevuti
 */
240 public void reportMsgRicevuti() throws Exception {

242             logger.info("********** Report Messaggi Ricevuti *********");
                boolean newConn = false;
244
                if (clientSocket!=null){
246                     logger.info("clientSocket:s");
                        logger.info("Socket chiuso: " + clientSocket.isClosed() );
248                     logger.info("Socket connesso: " + clientSocket.isConnected());
                }
250
                if ( (clientSocket!=null) && (clientSocket.isClosed()) ){
252                     logger.info("Connessione nuova");
                        login();
254                     newConn = true;
                }
256
                logger.info("Socket After Closed: " + clientSocket.isClosed() );
```

```
258                 logger.info("Socket␣After␣connesso:␣" + clientSocket.isConnected() );
                    logger.info("Socket␣:" + clientSocket);
260
                    SMSReceived a = new SMSReceived();
262
                    DataOutputStream outToServer = new DataOutputStream(
264                         clientSocket.getOutputStream());

266                 logger.info("After␣DataOutputStream␣outToServer");

268                 // creo un BufferReader per la risposta
                    BufferedReader inFromServer = new BufferedReader(
270                         new InputStreamReader(
                                    clientSocket.getInputStream()));
272

274                 logger.info("After␣BufferedReader␣inFromServer");

276
                    logger.info("Eseguo␣il␣report␣messaggi␣ricevuti");
278
                    // abilito il report dei messaggi in ricezione (Type=receivednew)
280                 outToServer.writeBytes("Report\n");

282                 //      Thread.sleep(5000);

284                 outToServer.writeBytes("Type=receivednew\n");

286                 //      Thread.sleep(5000);

288                 logger.info("After␣Report␣␣Type=receivednew");
                    // da mail bisogna attendere tra i due invii qualche secondo
290                 Thread.sleep(5000);
                    outToServer.writeBytes("\n");
292
                    logger.info("After␣Report␣outToServer.writeBytes");;
294                 String risposta = "";
                    String msg = "";
296
                    // Riceve il report dei messaggi ricevuti e salva tutto sulla
298                 // stringa msg (fintanto che non ottengo la stringa OK)
                    risposta = inFromServer.readLine();
300
                    //      stampaRisposta(inFromServer);
302
                    logger.debug("Risposta:␣" + risposta);
304
                    logger.info("After␣inFromServer.readLine()");
306
                    int k = 0;
308                 while (risposta.compareTo("OK") != 0) {

310                         logger.debug("dentro␣ciclo␣" + k + "␣while␣risposta:" +
                                                risposta);
312                         logger.debug("dentro␣ciclo␣" + k + "␣msg:␣" + msg);

314                         if (risposta.compareTo("") != 0)
                                    msg = msg + risposta + "\n";
316                         risposta = inFromServer.readLine();
                            k++;
318
```

```
                }
320
                logger.info("After␣riposta␣compare");
322
                logger.info("msg:␣" + msg);
324
                int i = 0;
326
                // Divide la stringa msg in righe e lavora su quelle
328             StringTokenizer st = new StringTokenizer(msg, "\n");
                logger.debug("String␣tokenizer␣count:␣" +st.countTokens());
330
332
                while (st.hasMoreTokens()) {
334                     logger.info("Ciclo␣i:␣" + i);
                        risposta = st.nextToken();
336
                if (i % 8 == 0) {
338                     //crea nuovo SMS
                        a = new SMSReceived();
340                     //inserisce ID
                        //"MsgID=xxxxxxxx"
342                     a.setSmsid(risposta.substring(6));
                        //definisce il tipo di messaggio (sempre SMS)
344                     a.setMsgtype("SMS");
                        logger.info("**␣SMSid:␣" +  risposta.substring(6) +
346                                                 "␣a.setMsgtype(SMS);");
                }
348
350             if (i % 8 == 3){
                        // inserisce il mittente
352                     //"Orig=nnnnnnnnnnnn"
                        a.setSender(risposta.substring(5));
354                     logger.info("**␣Sender␣:␣" + risposta.substring(5));
                }
356
                if (i % 8 == 4){
358                     //inserisce il code messaggio

360                     a.setCode(risposta.substring(7));
                        logger.debug("**␣Code:␣" + risposta.substring(7));
362             }

364             if (i % 8 == 5) {
                        //risposta:"Date=yyyy/mm/dd hh:mm:ss GMT"
366                     //prende l'ora e la incrementa di 1 (questione di fuso orario)
                        int ora = Integer.parseInt(risposta.substring(16, 18)) + 1;
368                     logger.debug("**␣Ora:␣" + ora);

370                     //inserisce la data di arrivo del messaggio
                        a.setDatetime(risposta.substring(5, 16) + ora
372                                                 + risposta.substring(18, 24));

374                     logger.info("**␣Datetime:␣" + risposta.substring(5, 16) + ora
                                                + risposta.substring(18, 24));
376             }

378             if (i % 8 == 6){
                        //risposta:"body=asdsaasasasdssddsa"
```

```
380                         a.setMsgtext(risposta.substring(5));
                            logger.info("** Msgtext: " + risposta.substring(5));
382                 }

384                 if (i % 8 == 7) {
                            logger.info("** Messaggio Ricevuto!!!");
386                 }//fine if (i % 9 == 8)
                    i++;
388         }// fine while
                    logger.info("** Fine ciclo while (st.hasMoreTokens())");
390                 if (i == 0) {
                            logger.debug("Report ricevuti vuoto\n");
392                 }

394                 if(newConn){
                            logger.info("Logout invio newConn");
396                         logout();
                    }
398
                    if (clientSocket!=null) {
400                         logger.info("Socket chiuso: " + clientSocket.isClosed() );
                            logger.info("Socket connesso: " + clientSocket.isConnected());
402                 }
                    logger.info("********** Fine Report Messaggi Ricevuti *********");
404         }

406         /**
             * Report dei messaggi inviati
408          */
            public void reportMsgInviati() throws Exception {
410
                    logger.info("********** Report Messaggi Inviati **********");
412                 boolean newConn = false;

414                 logger.info("Socket chiuso: " + clientSocket.isClosed() );
                    logger.info("Socket connesso: " + clientSocket.isConnected() );
416
                    if (clientSocket.isClosed()){
418                         logger.info("Connessione nuova");
                            login();
420                         newConn = true;
                    }
422
                    logger.info("Socket After Closed: " + clientSocket.isClosed() );
424                 logger.info("Socket After connesso: " + clientSocket.isConnected() );

426                 logger.info("Socket :" + clientSocket);
                    SMSReceived a = new SMSReceived();
428
                    DataOutputStream outToServer = new DataOutputStream(
430                                                    clientSocket.getOutputStream());

432                 logger.info("After DataOutputStream outToServer ");

434                 // creo un BufferReader per la risposta
                    BufferedReader inFromServer = new BufferedReader(
436                                         new InputStreamReader(
                                               clientSocket.getInputStream()));
438
                    logger.info("After BufferedReader inFromServer ");
440
```

```
                outToServer.writeBytes("Report\n");
442             outToServer.writeBytes("Type=sent\n");

444             logger.info("After␣Report␣Type=sent");

446             // da mail bisogna attendere tra i due invii qualche secondo
                Thread.sleep(500);
448             outToServer.writeBytes("\n");

450             logger.info("After␣outToServer.writeBytes;");

452             logger.info("Report␣Type=sent");
                String risposta = "";
454             String msg = "";
                int i = 0;
456
                // Riceve il report dei messaggi ricevuti e salva tutto sulla
458             // stringa msg (fintanto che non ottengo la stringa OK)
                risposta = inFromServer.readLine();
460
                logger.info("After␣inFromServer.readLine()");
462
                logger.info("Risposta:␣␣" + risposta);
464
                while (risposta.compareTo("OK") != 0) {
466                     if (risposta.compareTo("") != 0)
                                msg = msg + risposta + "\n";
468                     risposta = inFromServer.readLine();
                }
470             logger.info("After␣while");

472             // System.out.println("MSG\n " + msg);
                // divide la stringa msg in righe e lavora su quelle
474             StringTokenizer st = new StringTokenizer(msg, "\n");

476             logger.info("String␣tokenizer␣count:␣" +st.countTokens());

478             while (st.hasMoreTokens()) {

480             logger.info("Cicolo␣i:␣" + i);
                risposta = st.nextToken();
482
                logger.info("After:␣risposta␣=␣st.nextToken();");
484
                if (i % 9 == 0) {
486                     //crea nuovo SMS
                        a = new SMSReceived();
488                     //inserisce ID
                        //"MsgID=xxxxxxxx"
490                     a.setSmsid(risposta.substring(6));
                        //definisce il tipo di messaggio (sempre SMS)
492                     a.setMsgtype("SMS");
                }
494

496             if (i % 9 == 4)
                        //inserisce il mittente
498                     //"Orig=nnnnnnnnnnnn"
                        a.setSender(risposta.substring(5));
500
                if (i % 9 == 5)
```

```
502                         //inserisce il code messaggio
504                         a.setCode(risposta.substring(7));
506                 if (i % 9 == 6) {
                            //risposta:"Date=yyyy/mm/dd hh:mm:ss GMT"
508                         //prende l'ora e la incrementa di 1 (questione di fuso orario)
                            int ora = Integer.parseInt(risposta.substring(16, 18)) + 1;
510
                            //inserisce la data di arrivo del messaggio
512                         a.setDatetime(risposta.substring(5, 16) + ora
                                        + risposta.substring(18, 24));
514                 }
                            if (i % 9 == 7)
516                         //risposta:"body=asdsaasasasdssddsa"
                            a.setMsgtext(risposta.substring(5));
518
                            if (i % 9 == 8) {
520                         //va in coda
                    }//fine if (i % 9 == 8)
522                 i++;
            }// fine while
524
                    logger.info("Fine ciclo while (st.hasMoreTokens())");
526                 // System.out.println(messaggi);
528                 if (i == 0) {
                            logger.debug("Report inviati vuoto\n");
530                 }// FINE if (i == 0)
532                 if(newConn){
                            logger.info("Logout invio");
534                         logout();
                    }
536                 logger.info("Socket chiuso: " + clientSocket.isClosed() );
                    logger.info("Socket connesso: " + clientSocket.isConnected() );
538
                    logger.info("********** Fine Report Messaggi Inviati **********");
540         }
542
            public boolean isConnected(){
544                 return clientSocket.isConnected();
            }
546
548
}
```

Listing B.3: *interfaccia SMSReceived*

```
1  package mioApache;
   import java.util.Date;
3

5  public class SMSReceived {

7          public void SMSReceived(){}

9          private String smsid;
```

```
11          private String msgtype;

13          private String code;

15          private String datetime;

17          private String sender;

19          private String msgtext;

21          public String getSmsid() {
                    return smsid;
23          }

25          public String getMsgtype() {
                    return msgtype;
27          }

29          public String getCode() {
                    return code;
31          }

33          public String getDatetime() {
                    return datetime;
35          }

37          public String getSender() {
                    return sender;
39          }

41          public String getMsgtext() {
                    return msgtext;
43          }

45          public void setSmsid(String smsid) {
                    this.smsid = smsid;
47          }

49          public void setMsgtype(String msgtype) {
                    this.msgtype = msgtype;
51          }

53          public void setCode(String code) {
                    this.code = code;
55          }

57          public void setDatetime(String datetime) {
                    this.datetime = datetime;
59          }

61          public void setSender(String sender) {
                    this.sender = sender;
63          }

65          public void setMsgtext(String msgtext) {
                    this.msgtext = msgtext;
67          }
    }
```

Appendice C

Struttura Database

NOME ATTRIBUTO	TIPO	Descrizione
Azienda_Nome	varchar(50)	Username dell'utente
Password	varchar(50)	Password dell'utente
Email	varchar(50)	Email dell'utente (obbligatoria per ricevere username e password)
Indirizzo	varchar(50)	Indirizzo dell'azienda che utilizza il servizio
Telefono	varchar(50)	Numero di telefono

Tabella C.1: *Struttura tabella Azienda*

NOME ATTRIBUTO	TIPO	Descrizione
Nomecampagna	varchar(30)	Nome scelto per la campagna promozionale
Messaggio	varchar(1600)	Messaggio da inviare
OraPrevistaInvio	datetime	Data di previsione invio *
OraInviata	datetime	Ora di invio *
Stato	varchar(30)	Stato della campagna (da inviare, inviata) *
Ripetizione	varchar(10)	Indica se l'utente ha scelto la ripetizione (true o false) *
RipValue	varchar(10)	Valore dell'intervallo di ripetizione * **
RipUnit	varchar(10)	Unità di tempo dell'intervallo di ripetizione * **
NumRip	int(11)	Numero di volte di ripetizioni * **
Azienda_Nome	varchar(50)	Nome dell'azienda che gestisce la campagna

** attributi usati per l'invio dei messaggi con programmazione posticipata*
***attributi utilizzati con la scelta di ripetizione*

Tabella C.2: *Struttura tabella Campagne*

NOME ATTRIBUTO	TIPO	Descrizione
Nomecampagna	varchar(30)	Nome scelto per la campagna promozionale
Azienda_Nome	varchar(50)	Nome dell'azienda che gestisce la campagna
Nome	varchar(30)	Nome della persona che fa parte della campagna promozionale
Cognome	varchar(30)	Cognome della persona che fa parte della campagna promozionale
Numero	varchar(10)	Numero della persona che fa parte della campagna promozionale

Tabella C.3: *Struttura tabella Campagne_has_Persone*

NOME ATTRIBUTO	TIPO	Descrizione
NomeElenco	varchar(50)	Nome scelto per l'elenco telefonico
Azienda_Nome	varchar(50)	Nome dell'azienda che gestisce l'elenco
Nome	varchar(30)	Nome della persona che fa parte dell'elenco
Cognome	varchar(30)	Cognome della persona che fa parte dell'elenco
Numero	varchar(10)	Numero della persona che fa parte dell'elenco

Tabella C.4: *Struttura tabella Elenchi_has_Persone*

NOME ATTRIBUTO	TIPO	Descrizione
Nome	varchar(30)	Nome
Cognome	varchar(30)	Cognome
Numero	varchar(10)	Numero
DataNascita	date	Data di nascita

Tabella C.5: *Struttura tabella Persone*

Bibliografia

[1] Michael Browers. *Pro CSS and HTML Desing Patterns*. Appress, 2007.

[2] Robert J. Brunner. *JSP: pratical guide for programmers*. Morgan Kaufmann, 2003.

[3] P. Canesi. *MAM - messaggistica aziendale Mobile, Protocollo di comunicazione*. Vodafone, 2003.

[4] Moro Clemente, Filira. *Sistemi Operativi*. EDIZIONI LIBRERIA PROGETTO PADOVA, 2006.

[5] Simmon Collison. *Beginning CSS Web Development*. Appress, 2006.

[6] Joseph D.Gradecki Mark Mathhews, Jim Cole. *MySQL and Java Developer's Guide*. WILEY, 2003.

[7] Robin Nixon. *Learning PHP, MySQL, Javascript*. O'REILLY, 2009.

[8] Shamkant B. Navathe Ramez A. Elmasri. *Sistemi di basi di dati, fondamenti*. PEARSON, Addison Weskey, 2007.

[9] Craig Walls with Ryan Breidenbach. *Spring in Action, Second Edition*. MANNING, August, 2007.