Estensione di un servizo di messaggistica per telefonia mobile (per una società di agenti TuCSoN)

System Overview

di
Mattia Bargellini

CAPITOLO 1

1.1 Introduzione

Il seguente progetto intende estendere le funzionalità di un agente già esistente (agente `SMS_Sender`), il cui task è quello di offrire un servizio di invio di sms, introducendo alcune migliorie e qualche nuovo servizio. In particolare ci si propone di:

- Sviluppare un comportamento reattivo a fronte di una richiesta di un invio multiplo;
- Sviluppare un servizio a fronte di una richiesta di un invio differito

1.2 Overview del Sistema Esistente

In questa sezione vengono riportate brevemente gli aspetti principali emersi dallo sviluppo del servizio di messaggistica:

1.2.1 Analisi:

Seguendo la metodologia SODA, possiamo definire un modello di ruoli, di interazione e di risorse utilizzate:

- *Role model:* Il ruolo di `SMS_Sender`, è quello di inviare SMS a fronte di richieste da parte di agenti-utenti che vivono nella medesima società. In particolare possiamo individuare i seguenti scopi:
    1. inviare un SMS a fronte di una richiesta da parte di altri agenti;
    2. fornire alla società una informazione sullo stato del servizio.

- *Interaction model:* Le informazioni richieste da `SMS_Sender` alla società sono:
    1. il numero di cellulare del destinatario;
    2. il testo del messaggio.

Viceversa, le informazioni che l'agente fornisce alla società riguardano lo stato del servizio.

- *Resource model:* La risorsa sfruttata è rappresentata dall'idea di gateway - che può essere visto come l'insieme di parti sia hardware sia software - tramite cui è possibile spedire di fatto l'SMS.

Aspetti sociali e di coordinazione:
- Il ruolo che giocano gli altri agenti della società è quello di utenti del servizio.
- Dato che l'agente `SMS_Sender` fornisce l'informazione sullo stato del servizio, è necessario tenere traccia di chi vuole inviare cosa in modo che l'agente interessato possa sapere se il servizio è andato a buon fine o meno. Questo problema di coordinazione verrà caricato sull'infrastruttura di coordinazione.

1.2.2 Progetto:

Mappiamo il ruolo di `SMS_Sender` direttamente su una agent class con lo stesso nome.
Operazioni che l'agente deve eseguire per completare il proprio task:

1. Controllare il tuple centre eseguendo una `in()` per trovare tuple logiche che matchano con il seguente template:

   ```
   sms_to_send(Recipient, Text)
   ```

   dove `Recipient` è una stringa contenente il numero del destinatario nel formato internazionale e `Text` è una stringa contenente il testo del messaggio di una lunghezza massima di 160 caratteri;
2. Estrarre dalla tupla le informazioni da passare al gateway. Eventualmente controllare che la lunghezza massima del testo e il formato del numero del destinatario siano conformi alle specifiche del gateway;
3. Passare i dati al gateway[1] per l'invio del messaggio; poi interrogarlo sullo stato dell'operazione al fine di informare la società sullo stato del servizio tramite la tupla:

[1] Verranno usate una serie di API Java chiamate `jSMSEngine` (http://jsmsengine.sourceforge.net) , con distribuzione free progettata per inviare e ricevere SMS da e sul pc, usando un cellulare collegato con la porta COM del calcolatore. Tale libreria fa uso delle API Java COMM per la comunicazione col cellulare.

```
sms_service_status(Status)
```

dove `Status` rappresenta lo stato del servizio: “ok” se il servizio è andato in porto o “failed” se ha fallito.

Richieste multiple e concorrenti:
Richieste multiple e concorrenti si riassumono in un problema di coda e può essere risolto, demandando al centro di tuple il compito di fornire agli agenti che richiedono il servizio un ticket con un identificativo.
Il template per questo protocollo è:

```
ticket(Id)
```

La generazione automatica delle tuple che rappresentano il ticket e la relativa gestione (al fine di realizzare politiche di distinzione e accodamento delle richieste), viene fornita dal tuple centre opportunamente programmato:

```
reaction(in(ticket(Id)),(
      pre,
      in_r(ticket_counter(Id)),
      Id1 is Id + 1,
      out_r(ticket_counter(Id)),
      out_r(ticket(Id)) )).
```

Avendo introdotto questa politica di coda, il protocollo usato da `SMS_Sender` diventa:
`sms_to_send(Id, Recipient, Text)`

1.2.3 Nota sull’ Implementazione:

Il package cui appartengono le classi su cui viene mappato l’agente è `alice.tucsonx.sms` ed è costituito dalle classi:

- `SMS_Sender.java` che rappresenta di fatto l’agente;
- `StartService.java` che serve per lanciare il servizio.

CAPITOLO 2

Di seguito si riportano le fasi di analisi di progetto e implementazione dei nuovi servizi e migliorie che si intendono apportare al sistema.

2.1 Analisi

1. Sviluppo di un comportamento reattivo a fronte di una richiesta di invio multiplo

Innanzitutto si definisce cosa si intende per **invio multiplo**: questo tipo di invio può essere visto sotto due accezioni: invio di un unico testo a più destinatari e/o invio a un solo destinatario di un testo la cui lunghezza supera i 160 caratteri.
Di conseguenza una singola richiesta diventa in realtà una serie di N richieste consecutive. Al fine di non andare ad agire su `SMS_Sender`, per non cambiarne il comportamento, si intende sviluppare un comportamento reattivo dell'infrastruttura che esegua questa "trasformazione".

Lo sviluppo di questo nuovo servizio fa nascere un problema di coordinazione; infatti la politica di coda introdotta da `SMS_Sender` deve essere mantenuta consistente e in fase di progetto verrà discussa più approfonditamente.

2. Sviluppo di un servizio a fronte di una richiesta di invio differito

Si definisce cosa si intende per **invio differito**: questo tipo di invio è l'invio a una data e/o ora prefissata.
Definiamo un modello di ruolo, di interazioni e di risorse utilizzate per l'agente `AlarmClock`[2]:

- *Role Model:* il ruolo di `AlarmClock` è quello di sveglia/allarme e il suo scopo è quello di emettere una segnalazione allo scoccare della cosiddetta ora X;

[2] Per comodità ci si riferirà ora sempre al servizio di sveglia come all'agente `AlarmClock`.

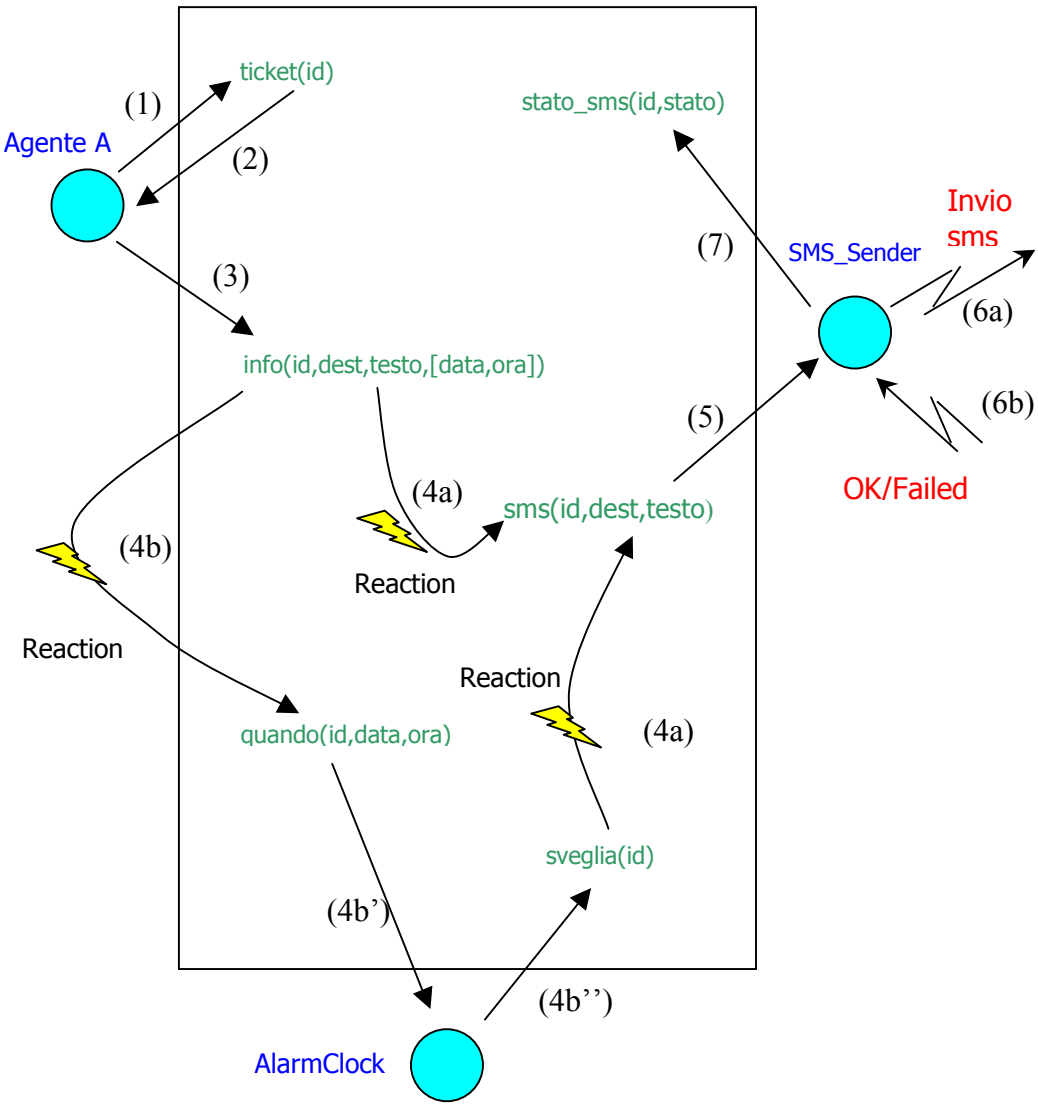
- *Interaction model*: le informazioni richieste dall'agente sono la data e l'ora a cui far scattare la segnalazione;

- *Resource model*: le risorse sfruttate dall'agente sono quelle temporali fornite dall'infrastruttura in cui "vive" l'agente.

L'aggiunta di questo nuovo servizio a monte dell'invio di un sms crea alcuni problemi di coordinazione tra `SMS_Sender` e `AlarmClock`, e dunque tra i due servizi serve un qualche meccanismo di raccordo in modo che sia possibile accedere ad entrambi tramite l'immissione nell'infrastruttura di una sola informazione sottoforma di tupla logica. A tal proposito si pensa di sviluppare un ulteriore comportamento reattivo dell'infrastruttura, che tra l'altro rende lo sviluppo di `AlarmClock` il più possibile disaccoppiato da `SMS_Sender`. Questo fa si che il servizio di sveglia possa essere offerto anche ad altri agenti della società che se ne vogliano servire per scopi differenti dall'invio di un sms.

Esempio di funzionamento

Nella figura seguente si riporta lo schema di funzionamento del sistema. Inizialmente l'Agente A (utente del servizio) richiede un ticket, poi inserisce delle informazioni contenenti uno o più destinatari, un testo ed eventualmente una data e un'ora (l'opzionalità è indicata dalle parentesi []). L'infrastruttura reagisce con la reazione (4a) o (4b) dipendentemente dal tipo di informazione inserita.
Se data e ora non sono presenti, viene triggerata direttamente la reazione (4a) che implementa di fatto il primo punto del progetto e produce delle informazioni leggibili da `SMS_Sender`. Diversamente, se l'informazione contiene una data e un'ora, viene scatenata la reazione (4b) che crea un input per `AlarmClock` il quale produrrà una sveglia all'ora X. La successiva reazione è ancora la (4a).
Comunque in entrambi i casi, `SMS_Sender` risponderà con lo stato del servizio.


ticket(id)
(1)
(2)
Agente A
(3)
info(id,dest,testo,[data,ora])
(4a)
Reaction
sms(id,dest,testo)
(4b)
Reaction
quando(id,data,ora)
Reaction
(4a)
sveglia(id)
(4b')
(4b'')
AlarmClock
stato_sms(id,stato)
(7)
SMS_Sender
Invio sms
(6a)
(6b)
OK/Failed
(5)

2.2 Progetto

Tecnologie usate

- Infrastruttura TuCSoN:
- Agenti implementati in Java;
- Reazioni in linguaggio ReSpecT.

1. Comportamento reattivo a fronte di una richiesta di invio multiplo

Nel progettare questa estensione al servizio, il primo scopo è quello di non voler modificare il protocollo di `SMS_Sender` visto nel Capitolo 1:

`sms_to_send(Id, Recipient, Text)` (1)

Perciò verrà costruita una reazione che produca tuple del formato (1) a fronte di un evento del tipo `out(sms (Id, Recipient, Text))` dove `sms (Id, Recipient, Text)` (2) è un nuovo formato di tupla che gli utenti dovranno usare per ottenere questo servizio; in particolare, `Recipient` è una lista di destinatari (ad esempio ["+393331112224","+394445556667"][3]) e `Text` è un testo tra virgolette "".
La reazione nel suo complesso deve controllare la quantità di destinatari e la lunghezza del testo e produrre il numero adeguato di tuple per l'agente `SMS_Sender`:

```
reaction(out(sms(Id,Recipient,Text)),(
      in_r(sms(Id,Recipient,Text)),
      spawn(checker,java('alice.tucsonx.sms.Checker'),[sms(Id,Reci
pient,Text)]) )).
reaction(out_r(smsx(Id,Recipient,Text)),(
      in_r(smsx(Id,Recipient,Text)),
      spawn(checker,java('alice.tucsonx.sms.Checker'),[smsx(Id,Rec
ipient,Text)]) )).
```

[3] Per semplicità la lunghezza del numero di cellulare verrà impostato con una dimensione di 13 elementi: 3 elementi per il prefisso internazionale, 3 per il prefisso del gestore e 7 del numero restante.

In realtà tale reazione non fa altro che eliminare dal nodo le tuple immesse e passare i parametri ad un agente che viene creato *ad hoc* (agente `Checker` contenuto nel package `alice.tucsonx.sms`), il cui compito è quello di eseguire il vero controllo per l'invio multiplo e scrivere tuple nel formato per `SMS_Sender`.
Inoltre viene progettata una altra reazione che permette di leggere tutte le tuple eventualmente inserite prima dell'avvio di `SMS_Sender`:

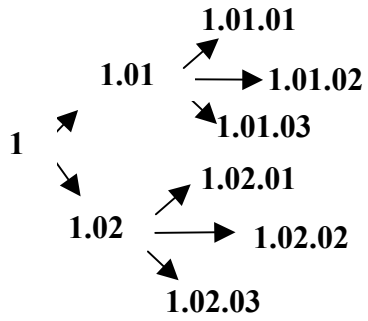
```
reaction(out(reset_sms),(
      in_r(reset_sms),
      out_r(reset_loop_sms) )).
reaction(out_r(reset_loop_sms),(
      in_r(sms(Id,Rec,Text)),
      out_r(smsx(Id,Rec,Text)),
      out_r(reset_loop_sms) )).
reaction(out_r(reset_loop_sms),(
      in_r(reset_loop_sms) )).
```

Comportamento di `Checker`:
Innanzitutto, se il numero del destinatario contiene caratteri diversi dai digit scrive la tupla `sms_service_status(Id, "null service")`: il servizio generato è un *fake* in quanto non prodotto da `SMS_Sender`, che così non viene appesantito di ulteriore lavoro. Diversamente, se il numero è corretto, `Checker` controlla la quantità dei destinatari e la lunghezza del testo:

1. (2) contiene N destinatari e un testo minore di 160 caratteri: verranno prodotte N tuple nel formato (1), contenenti lo stesso testo;
2. (2) contiene un unico destinatario e un testo maggiore di 160 caratteri: verranno prodotte M tuple nel formato (1) (dove M = lunghezza testo/160) ognuna contenente il destinatario e M-esimo blocco di testo;
3. (2) contiene N destinatari e un testo maggiore di 160 caratteri: verranno prodotte N*M tuple nel formato (1), ognuna contenente N-esimo destinatario e M-esimo blocco di testo.

Come già emerso in fase di analisi, la creazione di una certa quantità di tuple a partire da una sola crea un problema di consistenza nel campo `Id` che ora deve essere strutturato in modo da poter risalire per ogni tupla prodotta a quella originaria. In particolare si intende adottare la struttura `Id_originario.xx.yy` dove `xx` e `yy` sono numeri progressivi tra 0 e 99: `xx` rappresenta lo splitting nel

senso della quantità di destinatari, `yy` lo splitting nel senso della lunghezza del testo. Ad esempio, supponendo Id=1, N=2 e M=3 si ha:



2. Servizio a fronte di una richiesta di invio differito

Il ruolo di `AlarmClock` viene mappato su una agent class con lo stesso nome. Operazioni che esegue per completare il proprio task:

1. Fa una `in`() per recuperare le informazioni dal centro di tuple tramite il protocollo `when(Id,Date,Time)`, dove `Date` è la data di invio nel formato "dd.mm.yyyy" (es: "30.01.2005") e `Time` è l'orario nel formato "hh.mm" 24 ore (es: "17.05");
2. Ricava il timestamp dalla tupla letta e lo confronta con quello corrente di sistema:
   - se il timestamp è minore di quello corrente inserisce la tupla `error(Id,Msg)`;
   - se sono uguali inserisce `wake_up(Id)`, in modo da "svegliare" la tupla coinvolta;
   - se il timestamp è maggiore di quello corrente, memorizza id e timestamp letti. A questo punto `AlarmClock` inizia ad eseguire delle `inp`() in modo da essere libero di controllare ripetutamente il timestamp corrente con quelli memorizzati. Appena uno di questi è uguale a quello corrente viene eseguita una `out(wake_up(Id))`;
   - ora, se vi sono ancora timestamp in memoria, `AlarmClock` ripete il procedimento del punto precedente, altrimenti esegue una `in()`, ripartendo così dal punto 1.

Come si nota, i protocolli usati da `AlarmClock` sono disaccopiati da quello di `SMS_Sender`: da un lato questo permette ad `AlarmClock` di fornire un servizio generico di sveglia, dall'altro rende necessaria la progettazione di una serie di reazioni che leghino il servizio di sveglia e quello di invio di sms al fine di creare un sistema che sia in grado di fornire il servizio di invio differito che si intendeva realizzare.Tali reazioni sono riportate di seguito:

1. La reazione seguente permette all'agente di leggere tutte le tuple eventualmente inserite prima del suo avvio o quando egli non è attivo sul nodo:

```
reaction(out(reset),(
      in_r(reset),
      out_r(reset_loop) )).
reaction(out_r(reset_loop),(
      in_r(sms(Id,Rec,Text,Data,Ora)),
      out_r(temp(Id,Rec,Text)),
      out_r(when(Id,Data,Ora)),
      out_r(reset_loop) )).
reaction(out_r(reset_loop),(
      in_r(reset_loop) )).
```

2. Questa reazione trasforma la tupla `sms(Id,Recipient,Text,Date,Time)` in una tupla temporanea `temp(Id,Recipient,Text)`, che tiene traccia delle informazioni sul destinatario e sul testo del messaggio e nella tupla `when(Id,Date,Time)` che viene letta da `AlarmClock`:

```
reaction(out(sms(Id,Recipient,Text,Date,Time)),(
      in_r(sms(Id,Recipient,Text,Date,Time)),
      out_r(temp(Id,Recipient,Text)),
      out_r(when(Id,Date,Time)) )).
```

3. Questa reazione serve per distruggere le tuple temporanee con id identico a quelle in cui si è verificato un errore:

```
reaction(out(error(Id,Msg)),(
      in_r(temp(Id,_,_)) )).
```

4 Quando `AlarmClock` genera il segnale di sveglia per un particolare identificativo, la reazione seguente elimina la tupla temporanea e immette nel

centro di tuple la tupla `sms(Id,Recipient,Text)` che triggera la reazione per l'invio multiplo:

```
reaction(out(wake_up(Id)),(
      in_r(temp(Id,Recipient,Text)),
      out_r(sms(Id,Recipient,Text)) )).
```

2.3 Implementazione

Il package cui appartengono le classi su cui vengono mappati gli agenti è `alice.tucsonx.sms` ed è costituito dalle classi:

- `SMS_Sender.java`: rappresenta l'agente incaricato di inviare gli sms;
- `StartService.java`: che serve per lanciare il servizio. Setta inoltre le reazioni per la politica di coda e per l'invio multiplo;
- `Checker.java`: è l'agente lanciato dalla reazione per l'invio multiplo settata da `StartService.java` ed esegue il controllo sulla quantità dei destinatari e la lunghezza del messaggio;
- `AlarmClock.java`: rappresenta l'agente incaricato di fare da "sveglia";
- `StartAlarm.java`: serve per lanciare il servizio di sveglia. Setta inoltre e reazioni per l'invio differito.

Conclusioni

In questo progetto sono stati sviluppati agenti e reazioni per fornire un servizio a una società. Per questo motivo l'intelligenza del sistema non risiede nei singoli agenti, i quali per altro eseguono operazioni piuttosto semplici e non necessitano di strategie, ma piuttosto l'intelligenza risiede nella cooperazione e coordinazione tra i due agenti che singolarmente fornirebbero un ben scarno servizio. Dunque, visto ai "morsetti", il sistema nel suo complesso è capace di rispondere e riconoscere le situazioni che gli si propongono e offrire un servizio piuttosto articolato.
Da notare come `SMS_Sender` sia puramente reattivo perché agisce solo quando è presente una informazione per lui, mentre `AlarmClock` è sia reattivo sia proattivo in quanto in alcune circostanze rimane in attesa di un input e in altre agisce per conto proprio.
Infine, la coordinazione tra gli agenti è stata possibile grazie ai centri di tuple messi a disposizione dall'infrastruttura TuCSoN, tramite cui si è potuto sviluppare `AlarmClock` indipendentemente dal servizio offerto da `SMS_Sender`. Questo ha permesso di non caricare sugli agenti tutte le regole di coordinazione e di rendere il loro sviluppo più semplice e solamente finalizzato al servizio.