SOMMARIO

1 Introduzione 5
2 Obiettivi 6
3 Descrizione generale dell'architettura 7
4 Principi di utilizzo 9
4.1 Programmatore 9
4.2 Utilizzatore 10
5 Scelte progettuali 13
5.1 Struttura generale 13
5.1.1 Struttura a cartelle 13
5.2 Microsoft Bot Framework 13
5.3 MoCA Client 14
5.4 Luis 15
5.4.1 Estensibilità con Microsoft Cognitive Service 16
5.5 Conversazioni 17
5.6 BOT – MoCA Client Principale 19
5.7 Database MySQL 20
6 Microservizi 21
7 Logging e Monitoraggio 22
8 Protocolli 24
8.1 Protocollo HTTP 24
8.1.1 Messaggio di richiesta 24
8.1.2 Messaggio di risposta 25
8.1.3 TLS e crittografia 28
8.1.4 Formato JSON 28
8.1.5 Direct Line API 29
8.1.6 Autenticazione 30
8.1.7 Avviare una conversazione 30
8.1.8 Mandare un messaggio nella conversazione 31
8.1.9 Mandare un allegato multimediale alla conversazione 31
8.1.10 Ricevere messaggi dal bot 32
8.1.11 Chiudere una conversazione 32
8.1.12 Modificare un messaggio o cancellarlo 32
8.2 MoCA Protocol 33
8.2.1 Header 33

8.2.2 Body 33
9 MoCA SDK 35
9.1 Integrazione con MoCA Client (base della SDK) 35
9.2 Descrizione MoCA SDK (versione C) 35
9.3 Esempio interfacciamento MoCA Client 36
10 Descrizione software 37
10.1 IDE e strumenti di sviluppo 37
10.2 Git e supporto al codice 39
10.3 Sorgente MoCA Client 39
10.3.1 mocamain.c 39
10.3.2 Fontautil 43
10.3.3 fontaMoCAProtClient 45
10.3.4 fontahttp 45
10.3.5 Fontabotconnector 48
10.4 Sorgente BOT Core 50
10.4.1 Dialogs 51
10.4.2 Functions 54
10.4.3 MoCA 54
10.4.4 Sentence 56
10.5 MySQL REST API 56
11 Conclusioni 58
12 Sviluppi futuri 58

INDICE DELLE FIGURE

FIGURA 3.1 SCHEMA A BLOCCHI DEL SISTEMA GENERALE 7
FIGURA 3.2 SCHEMA A LIVELLI DEL SISTEMA GENERALE 8
FIGURA 4.1 SCHERMATA DI AVVIO DEL CLIENT MOCA 9
FIGURA 4.2 SCHEMA A BLOCCHI DELL'INTERAZIONE UTENTE-BOT-LUIS 10
FIGURA 4.3 SCHEMA A BLOCCHI DELLA MODIFICA DA PARTE DELL'UTENTE DELLO USER ID 10
FIGURA 4.4 SCHEMA A BLOCCHI DELL'AVVIO DEL SERVIZIO DA PARTE DELL'UTENTE 11
FIGURA 4.5 SCHEMA A BLOCCHI DELL'INTERAZIONE CON IL SERVIZIO 12
FIGURA 5.1 SCHEMA A BLOCCHI DI INTERAZIONE TRA UTENTE-LUIS 15
FIGURA 5.2 MESSAGGIO CON CORRETTA ASSEGNAZIONE DELL'INTENT 16
FIGURA 5.3 MESSAGGIO CON ASSEGNAZIONE ERRATA DELL'INTENT 16
FIGURA 5.4 SCHEMA A BLOCCHI DEL FLUSSO DELLE CONVERSAZIONI 17
FIGURA 5.5 DIAGRAMMA DI SEQUENZA DEI MESSAGGI DEL SISTEMA 18
FIGURA 5.6 DIAGRAMMA DEI FLUSSI DEI MESSAGGI ALL'INTERNO DEL BOT CORE 19
FIGURA 5.7 SCHEMA A BLOCCHI DELLE CONVERSAZIONI 19
FIGURA 5.8 RAPPRESENTAZIONE DEL DATABASE MYSQL 20
FIGURA 7.1 PANNELLO DI CONTROLLO DI MICROSOF AZURE 22
FIGURA 7.2 ALLOCAZIOE DI MEMORIA DEL BOT NEL TEMPO 23
FIGURA 7.3 TEMPO MEDIO DI RISPOSTA NEL TEMPO 23
FIGURA 8.1 MODELLI ISO/OSI E TCP/IP 28
FIGURA 8.2 SCHEMA A LIVELLI DEL SISTEMA GENERALE 29
FIGURA 8.3 IMBUSTAMENTO MULTIPLO PER IL MOCA PROTOCOL 33
FIGURA 9.1 SCHEMA A BLOCCHI DELLA COMUNICAZIONE TRA MOCA CLIENT E SERVIZIO 35
FIGURA 10.1 BOT FRAMEWORK CHANNEL EMULATOR CLIENT 38
FIGURA 10.2 ESEMPIO DI GESTIONE DELLE MODIFICHE CODICE 39
FIGURA 10.3 FLOWCHART DEL MAIN DEL MOCA CLIENT 42
FIGURA 10.4 FLOWCHART DI STARTSERVICE 49
FIGURA 10.5 FLUSSI DEI DIALOG NEL BOT CORE 51

RINGRAZIAMENTI

Vorrei ringraziare il mio relatore, il professor Pier Luca Montessoro, per gli ottimi consigli e la libertà con cui mi ha lasciato strutturare il sistema che andremo ad analizzare in questa tesi. Inoltre vorrei profondamente ringraziare Microsoft Italia che oltre ad accogliermi come Microsoft Student Partner Team Leader di Udine, mi ha fornito tutti gli strumenti e le conoscenze necessari alla progettazione ed allo sviluppo della tesi. In particolare vorrei ringraziare Jessica Tibaldi, Elena Quartini e Guenda Sciancalepore per avermi introdotto all'argomento dei BOT ed essere state disponibili a darmi una mano quando ne avevo bisogno.

1 INTRODUZIONE

Con la progressiva diffusione della rete ad alta velocità e di terminali con caratteristiche sempre migliori, pensiamo a smartphone, tablet o personal computer, nel giro di pochi anni la struttura dell'interfacciamento Uomo-Macchina ha seguito una serie di mutamenti sostanziali che hanno portato ad una moltitudine di possibilità in termini di interazione e condivisione delle informazioni. Pensiamo ad esempio ai primi calcolatori (dagli anni '70 in poi), che sfruttavano la riga di comando come principale metodo di intercomunicazione tra uomo e macchina. Successivamente con l'aumentare delle prestazioni elettroniche sono aumentate anche le possibilità in termini di interazione, come le prime interfacce grafiche. Negli ultimi 10 anni, con la nascita di dispositivi portatili con caratteristiche prestazionali equiparabili ai computer casalinghi, oltre alla nascita di nuovi mercati, come quello degli applicativi per smartphone, si sono diffuse le così dette "reti sociali", o semplicemente "social" (pensiamo a Facebook, Twitter, Youtube, Instagram, Telegram …). Tali reti non permettono soltanto una interazione con il terminale che si sta sfruttando ma mette in comunicazione i vari utenti sotto diverse forme, pensiamo alla condivisione di video o foto, commenti testuali o "reazionali". Tali reti sostanzialmente veicolano informazioni che possono essere sfruttante come leve di marketing o come possibilità di ulteriori connessioni sia Utente – Utente, sia Utente – Macchina. Proprio su quest'ultimo caso è basata questa tesi: con l'impiego delle reti sociali nel campo dell'interfacciamento ad un servizio, che sia virtuale o fisico, non sarà necessario più uno sviluppo grafico, ma la rete sociale sfrutterà la sua caratteristica universalità per fornire la stessa interfaccia a qualunque tipo di servizio ad essa collegato. In questo modo non sono più necessari applicativi custom installati sui terminali (PC o smartphone) per accedere ad un servizio. Fino adesso si sono definiti i "servizi" come unità astratta ma di fatto i servizi che possono essere collegati a tali reti sono diversi e i più disparati: un esempio è un servizio di ordinazione di cibo a domicilio piuttosto di uno che imposti la sveglia dello smartphone ad una determinata ora o la macchina del caffè che prepari una bevanda calda quando torno da casa. Gli assistenti personali in questo mondo si ritagliano uno spazio fondamentale: reti come Cortana[1] di Microsoft o Siri[2] di Apple sfruttano programmi, reti neurali e database per interpretare quello che un utente scrive o dice (riconoscimento volale), usano Internet per collegarsi con i servizi necessari a rielaborare tale informazione e forniscono una risposta rispetto alla persona fisica con cui stanno dialogando. Un programma che sfrutta questo tipo di approccio è definito BOT: *programma che accede alla rete attraverso lo stesso tipo di canali utilizzati dagli utenti umani (per esempio che accede alle pagine Web, invia messaggi in una chat, si muove nei videogiochi, e così via). Nei paesi anglosassoni, con "Bot" s'intende un programma autonomo che nei social network fa credere all'utente di comunicare con un'altra persona umana. Questi bot migliorano di anno in anno ed è sempre più difficile distinguere un bot da una persona umana*[3]. Ricapitolando: le reti sociali, oltre ad integrare una interfaccia comoda all'utente, ben studiata ed uguale per tutti, integrano un sistema di comunicazione che può essere sfruttato da un BOT, cioè un programma che da un lato si interfaccia con l'utente dialogandoci come se fosse un umano, dall'altra possa avviare servizi o algoritmi di cui un utente o un altro servizio ha bisogno. Non a caso ho citato il termine "servizio", con il quale intendo un programma o una suite di programmi atti a svolgere un compito o una serie di operazioni autonomamente o rispetto a degli stimoli che arrivano dall'esterno. Con questa tesi voglio sostenere un modello di sviluppo software che negli ultimi anni, grazie allo sviluppo delle infrastrutture di intercomunicazione, sta prendendo sempre più piede: il modello a microservizi o SOA (Service Oriented Architecture). Il modello SOA, che verrà discusso maggiormente più avanti nel corso di questa tesi, trova la sua caratteristica peculiare nella distribuzione delle funzioni e del carico di lavoro su più dispositivi. Lo svantaggio principale è un degrado generale della prestazione media rispetto ad un software monolitico, dovuta al fatto che per integrare tecnologie diverse, implementate su piattaforme diverse, è necessario attuare scelte (riguardo a protocolli di comunicazione o storicizzazione del dato) che rendono mediamente meno efficiente il sistema, inoltre i componenti che formano i vari servizi possono essere istanziati anche in zone geografiche molto distanti tra loro: questo fatto comporta sia problemi di latenza che di affidabilità. Questi limiti potrebbero risultare fatali per il corretto funzionamento del sistema nel caso in cui esso sia adibito ad applicazioni di elaborazione dati o comunque sistemi in cui è necessaria elevata efficienza. Considerato però il fatto che il fruitore finale del servizio è una persona fisica, i limiti di efficienza non sollevano una problematica sostanziale quanto quelli di scalabilità, manutenibilità ed

[1] https://www.microsoft.com/en-us/windows/cortana
[2] https://www.apple.com/ios/siri/
[3] https://it.wikipedia.org/wiki/Bot

estensibilità. Se pensiamo che il sistema che stiamo progettando abbia una valenza commerciale, i tre punti appena elencati sono fondamentali per la gestione di quello che si definisce “ciclo di vita del software”. Pensiamo per esempio ad un apparato a cui si collegano svariate centinaia di migliaia di persone attraverso i propri dispositivi mobili. Un sistema monolitico dovrebbe prevedere sottosistemi di smistamento del traffico, gestione delle risorse e tutta una serie di protezioni da attacchi informatici che oltre ad appesantire il sistema lo rendono anche molto costoso. I sistemi SOA per loro natura implementano crittografia, gestione delle connessioni e dei carichi, caratteristiche che non vanno sottovalutate in ambiti con forti escursioni di carico di lavoro.

2 OBIETTIVI

L’obiettivo della tesi è quello di progettare e realizzare un BOT che sfrutti le reti sociali per eseguire e/o interagire con i programmi realizzati dagli studenti durante il corso di Fondamenti di Programmazione tenuto dal professor Pier Luca Montessoro all’Università degli Studi di Udine. Di fatto si pone l’obiettivo di realizzare un sistema scalabile che permetta di estendere le proprie necessità in base alle funzionalità previste e realizzate dagli studenti ogni anno, nonostante esse siano differenti sia nella forma che nelle modalità di realizzazione. Si vuole porre particolare attenzione alle scelte progettuali e realizzative in modo che combacino con i principi dei sistemi SOA e che possano essere impiegate anche per sistemi differenti e legati maggiormente alla commercializzazione o in ambiti industriali.

3 DESCRIZIONE GENERALE DELL'ARCHITETTURA

Il sistema si basa su una struttura modulare collegata principalmente da API REST (Direct Line API[4]) che quindi sfrutta la rete Internet come elemento di interconnessione tra i vari sottosistemi. Ogni sottosistema lavora indipendentemente dagli altri, formando così una catena di servizi che possono essere allocati su macchine differenti. Tale approccio migliora la scalabilità del sistema stesso in quanto i programmatori dei singoli servizi non devono per forza progettare codici e applicazioni per una singola architettura, ma possono sfruttare le funzionalità e la potenza di calcolo delle architetture hardware di cui sono in possesso oppure hanno già conoscenza.

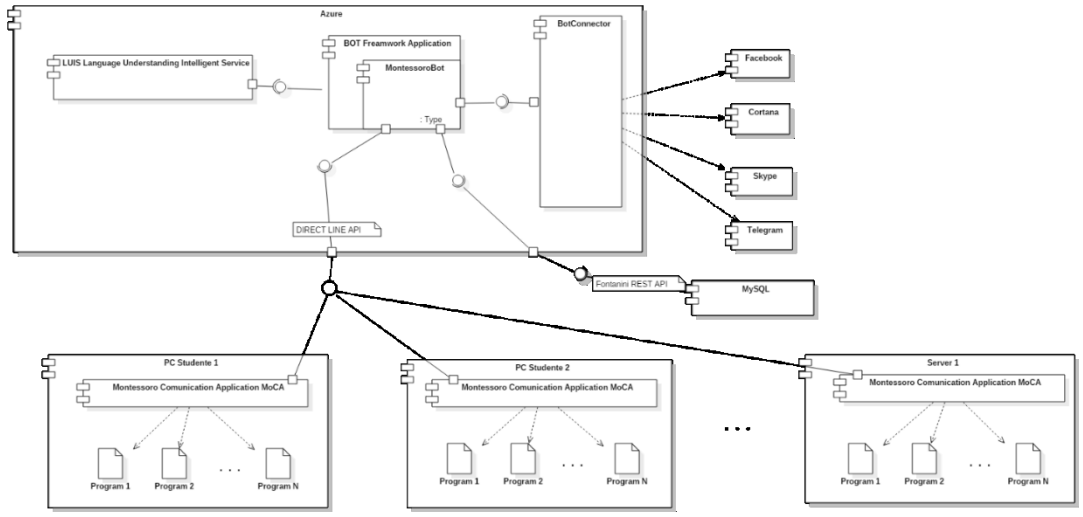


Figura 3.1 Schema a blocchi del sistema generale

Per sfruttare al meglio l'estensibilità e l'interconnessione con i servizi social (Facebook, Telegram tra i vari…) si è deciso di sfruttare il Microsoft Bot Framework[5] come base per la creazione del BOT atto a connettere i vari sottosistemi alle reti social. Tale framework permette di sfruttare la piattaforma Microsoft Azure[6] per istanziare il BOT, scritto in questo caso in linguaggio C# (disponibile anche in NODE.JS). Come si vede dalla seguente immagine, il sistema si struttura a strati, con i servizi che sono collegati al resto del sistema attraverso i MoCA client (Montessoro Connector Application), cioè dei programmi scritti in linguaggio C atti a gestire i servizi e comunicare con Microsoft Azure.

[4] https://docs.microsoft.com/en-us/bot-framework/rest-api/bot-framework-rest-direct-line-3-0-concepts
[5] https://dev.botframework.com/
[6] https://azure.microsoft.com/it-it/

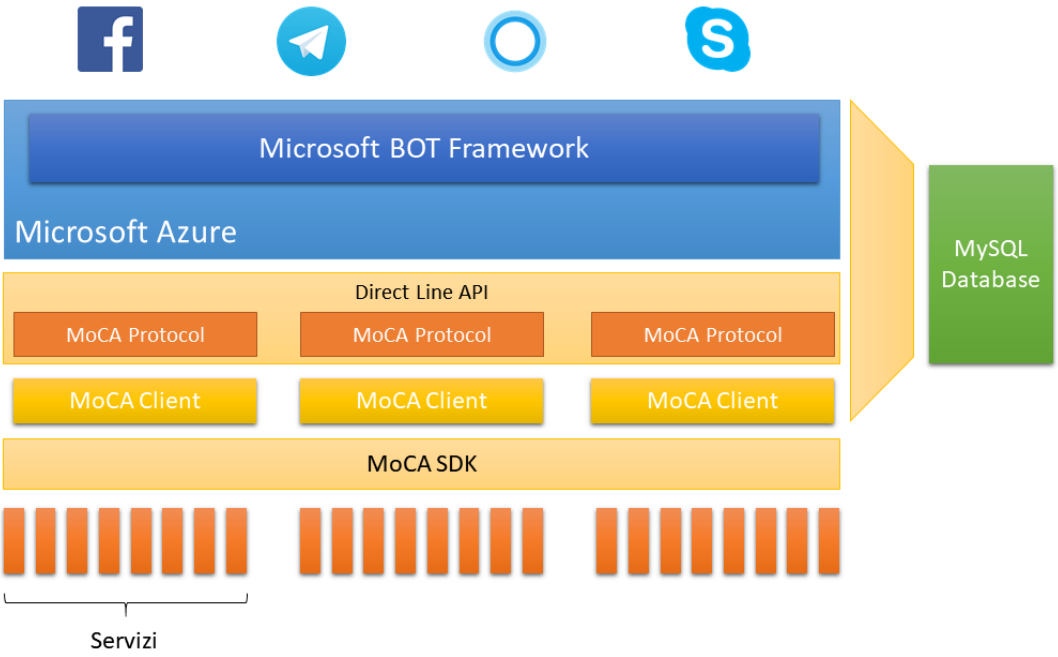


Figura 3.2 Schema a livelli del sistema generale

4 PRINCIPI DI UTILIZZO

Sono previste due modalità di utilizzo del pacchetto software:

- Programmatore, è l'utente che realizza, istanzia e gestisce i servizi aggiuntivi del bot i quali sono raggiungibili attraverso la conoscenza dell'UserID del programmatore (in genere viene utilizzata la matricola per identificare uno studente). Il programmatore fa uso della MoCA SDK per collegare i propri servizi al MoCA Client che a sua volta inoltra i messaggi al resto del sistema.
- Utilizzatore, il quale sfrutta solo i servizi esposti dal bot e i servizi messi a disposizione dagli utenti programmatori attraverso i Social Network a cui esso è collegato.

In ogni caso per testare i servizi esposti, entrambe le tipologie di utenti vengono viste come utilizzatori lato Social in quanto non c'è distinzione tra il programmatore del servizio e chi ne fa uso. D'altro canto chi fornisce il servizio (programmatore), possedendo lui l'elaboratore su cui è lanciato il MoCA client e di conseguenza tutti i servizi, può decidere quali e quanti servizi permettere che vengano avviati da lui stesso o da terzi.

4.1 Programmatore

Il programmatore, dopo aver compilato e salvato tutti i servizi come descritto nel paragrafo "struttura a cartelle" può avviare il client MoCA che automaticamente cercherà nelle sottocartelle i servizi, pubblicherà le informazioni necessarie alla connessione con il Bot Core all'interno del database MySQL, ed aprirà una conversazione con il bot attraverso le Direct Line API dove circoleranno tutti i messaggi del MoCA Protocol (descritto nella sezione Protocolli).

Nella seguente immagine è mostrata la schermata principale del client MoCA il quale:

- Ha trovato i servizi "ciao" e "sendandrecive"
- Ha eseguito un controllo sulle cartelle dei servizi per verificare che tutti i file necessari all'avvio del servizio esistessero e fossero utilizzabili
- Ha allocato la memoria necessaria per istanziare 50000 processi simultaneamente (se supportati dall'architettura)
- Ha aperto la connessione principale con il BOT core
- Ha creato il poller per la conversazione principale citata sopra e per le conversazioni collaterali
- Ha inviato le informazioni inerenti al client al database MySQL

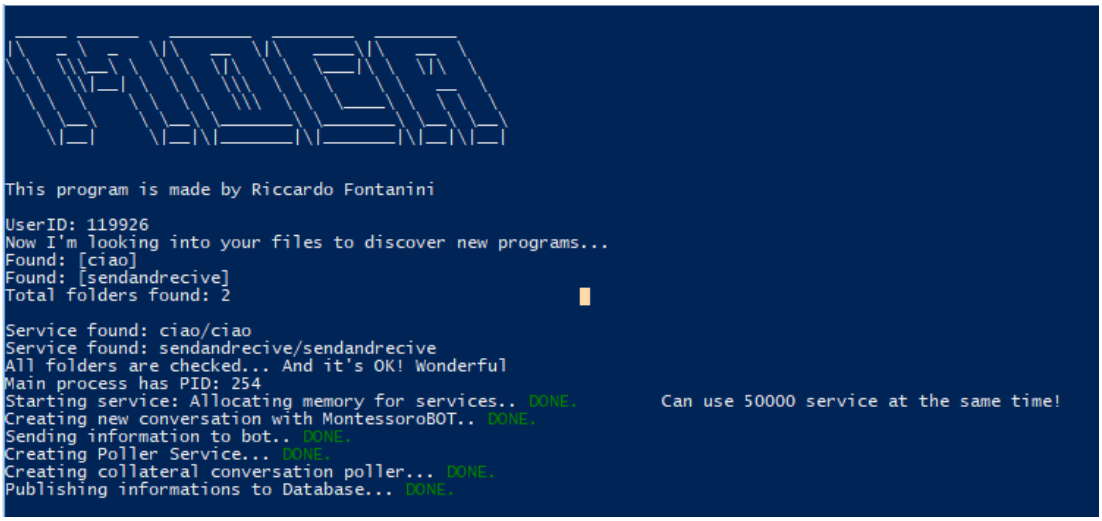


Figura 4.1 Schermata di avvio del Client MoCA

Le conversazioni principali sono una serie di messaggi http che servono per inviare parametri e comandi da un MoCA Client verso il BOT Core. Le conversazioni collaterali sono create al momento dell'avvio di un servizio e all'interno di esse vengono scambiati solamente messaggi tra utente e servizio. Queste due tipologie di comunicazione verranno approfondite meglio nel paragrafo Protocolli.

4.2 Utilizzatore

L'utilizzatore può utilizzare i servizi intrinsechi del bot (come ad esempio la richiesta delle informazioni sul ricevimento studenti) e, dopo che è stato avviato un client MoCA con successo dal Programmatore, dei nuovi servizi esposti dal Programmatore. Per utilizzare i servizi intrinsechi basta scrivere al BOT attraverso le reti sociali come se si scrivesse ad un umano, tale caratteristica è possibile grazie al cognitive service LUIS (vedere paragrafo dedicato), il quale sarà in grado di interpretare le frasi digitate e fornire una risposta, eseguendo a discrezione degli algoritmi a supporto di tale risposta. Di seguito è illustrato il flusso dei messaggio di interazione uomo-BOT.

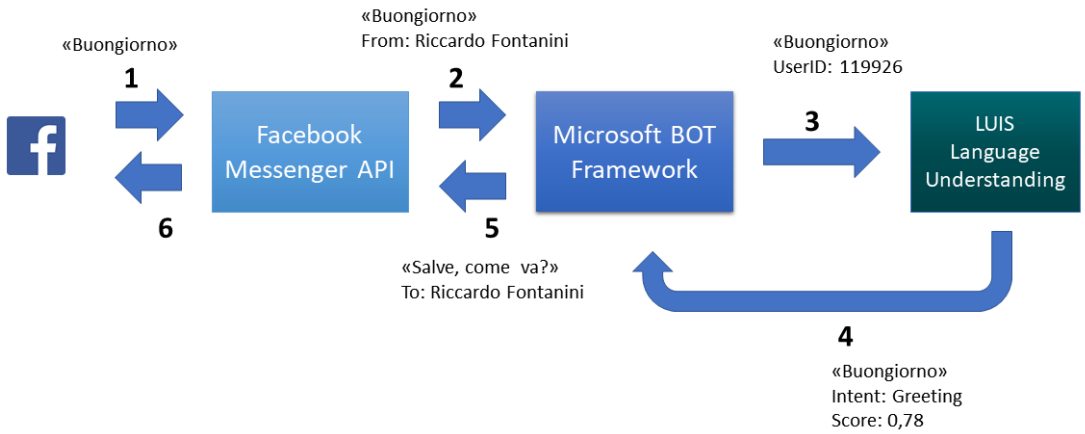


Figura 4.2 Schema a blocchi dell'interazione utente-bot-luis

Tale esempio verrà trattato nuovamente nel capitolo LUIS.

In seguito se un utente vuole avviare un servizio esposto da un specifico programmatore deve inviare al bot un messaggio per impostare l'User ID del programmatore sul proprio client: "imposta <UserID programmatore>". Ogni utente può avviare un solo servizio alla volta e può essere selezionato un solo UserID alla volta.

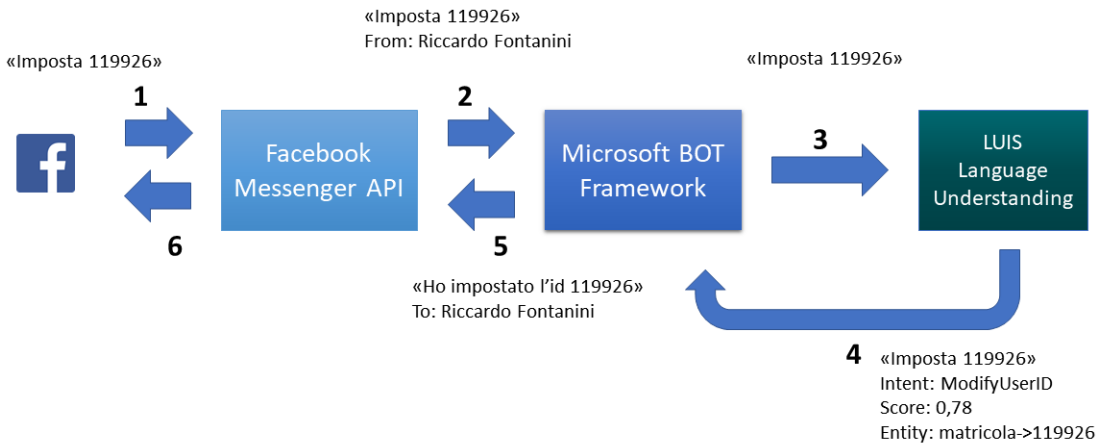


Figura 4.3 Schema a blocchi della modifica da parte dell'utente dello User ID

Per avviare un servizio tra quelli esposti dal client selezionato, basta lanciarlo con il comando "avvia <nome_servizio>".

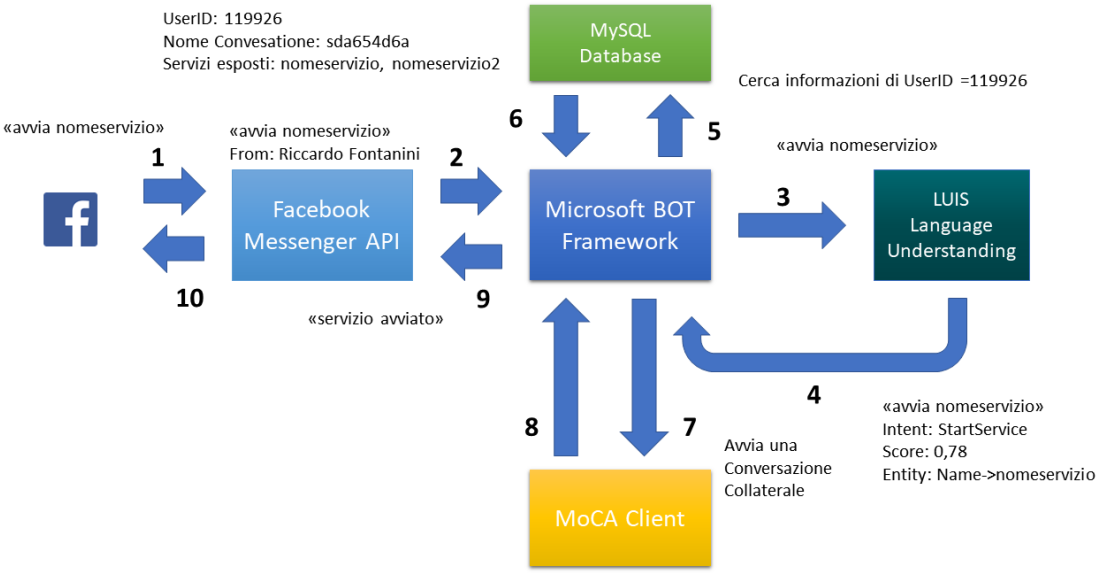


Figura 4.4 Schema a blocchi dell'avvio del servizio da parte dell'utente

1. Inoltro del messaggio di avvio del servizio
2. Arrivo del messaggio di avvio al BOT Core
3. Inoltro del messaggio a LUIS API per l'interpretazione
4. Messaggio riconosciuto ed associazione al Intent "StartService" con Entity che punta al nome del servizio che si vuole avviare
5. Ricerca informazioni sul database MySQL
6. Il primo record che viene trovato raccoglie le informazioni necessarie all'inoltro dei messaggi sulla corretta conversazione, inoltre elenca i servizi esposti
7. Comunicazione al moca client corretto dell'intenzione di avviare il servizio con nome "nomeservizio" e creazione della conversazione collaterale
8. Messaggio di notifica delle avvenute operazioni
9. Notifica all'utente che il servizio è stato avviato

Dopo l'avvio del servizio i messaggi provenienti dall'utente e dal servizio verranno re-indirizzati sulla conversazione collaterale aperta dal BOT Core in seguito alla precedente richiesta.

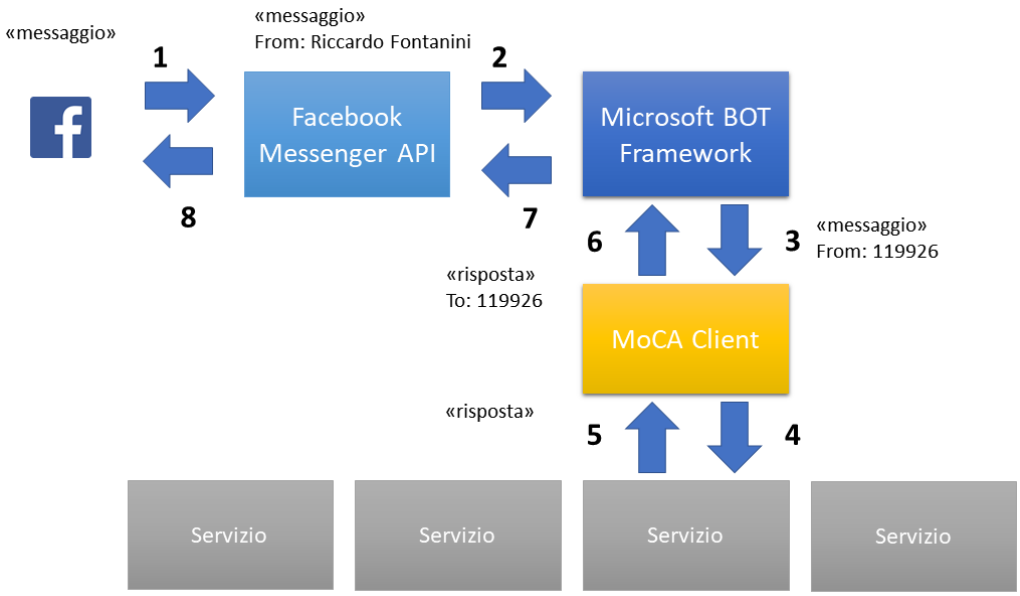


Figura 4.5 Schema a blocchi dell'interazione con il servizio

Una volta concluso il Task del servizio, il sistema arresterà automaticamente il processo e manderà una notifica all'utente utilizzatore che il servizio è stato arrestato. A quel punto l'utilizzatore può continuare con la navigazione tra i vari servizi intrinsechi del BOT o aprire un nuovo servizio come ha fatto precedentemente.

5 SCELTE PROGETTUALI

5.1 Struttura generale

La struttura basata su microservizi è frutto della volontà di poter estendere il comportamento e funzionalità del BOT direttamente dagli studenti e programmatori, indipendentemente da dove si trovano e dagli strumenti che utilizzano. Le nuove funzionalità hanno la necessità di essere assolutamente multipiattaforma e controllabili direttamente dagli stessi programmatori che in caso di malfunzionamenti, errori o comportamenti malevoli, risultassero isolati dal resto degli utenti e che solo messaggi di tipo testuale (eventualmente immagini o suoni) fossero inoltrati ai terminali dell'utente che fa richiesta del servizio. Un aspetto fondamentale è la sicurezza: infatti attraverso i MoCA client i programmi vengono eseguiti direttamente sui PC, schede emdedded o macchine virtuali dei programmatori, cioè di loro proprietà. La mancanza di un "servizio centrale" a cui affidare i propri algoritmi riduce sensibilmente la possibilità di inoltrare virus informatici che potrebbero danneggiare il sistema o esporlo a vulnerabilità. I client MoCA sono scritti in linguaggio C per sistemi operativi UNIX, scelta dettata per compatibilità ai sistemi impiegati durante il corso di studi: facilitare l'utilizzo del client agli studenti. Una ulteriore possibilità che è stata considerata, ma poi scartata, è JAVA. Questo ottimo linguaggio che per definizione è multipiattaforma sarebbe stata la scelta ideale per realizzare il MoCA client, ma necessitando di una installazione davvero onerosa che potrebbe portare gli studenti a desistere all'utilizzo del client, si è deciso quindi di svilupparlo con lo stesso linguaggio visto durante il corso. L'installazione e la modalità di estensione, in genere, sono i muri più alti con i quali un programmatore in erba deve cimentarsi avendo a che fare con un nuovo Framework. Il sistema è stato progettato in modo che sia altamente estensibile: non importa in che linguaggio verranno scritti i vari servizi, il sistema accetta ogni tipo di programma eseguibile che possa comunicare attraverso il buffering su un file, azione in cui lo studente/programmatore è aiutato dal MoCA SDK. Si è voluta creare una facilità di installazione del servizio che sia al contempo semplice per il programmatore ed efficace all'avvio del servizio: la struttura a cartelle spiega in maniera chiara come è organizzato il sistema.

5.1.1 Struttura a cartelle

Il MoCA Client, cioè il programma client che permette la comunicazione tra servizi e BOT Framework, è installato nella cartella che qui definiremo radice /. In tale radice sono presenti altre cartelle, tante quante i servizi esposti. Ogni cartella è denominata con il nome che vogliamo assegnare al servizio e contiene il relativo eseguibile. Per esempio, se voglio esporre un servizio al BOT con il nome "*nameservice*", nella cartella radice deve essere presente il file: */nameservice/nameservice*. Si nota che l'eseguibile deve avere lo stesso nome della cartella in cui è contenuto.

Inoltre la comunicazione tra client MoCA ed i servizi tra sfrutta il buffering su file. All'avvio del relativo servizio il client MoCA deve creare due file nella cartella */tmp/*. Siccome ad ogni servizio che viene avviato è associato un *conversationID,* cioè un identificativo univoco per ogni conversazione tra BOT e Client, tali file saranno formattati come: *conversationIDMOCAwrite.txt* e *conversationIDMOCAread.txt.*

5.2 Microsoft Bot Framework

Una delle motivazioni principali per cui si è scelto di utilizzare Microsoft Bot Framework è l'ottima estensibilità verso le reti sociali, cioè a differenza di Wit.ai[7] (proprietà di Facebook) e Api.ai[8] che rappresentano i framework più utilizzati dopo quello di Microsoft, Bot Framework ha il maggior numero di servizi collegabili, tra cui:

- Skype
- Facebook (messenger)
- Cortana
- Telegram
- Email

[7] https://wit.ai/
[8] https://api.ai/

- Bing

Inoltre tale Framework espone SDK in C# o NODE.JS. La scelta personale è ricaduta su C# coadiuvato a Visual Studio 2015 come strumento di sviluppo.

Tale framework mi ha aiutato nella progettazione del sistema e nella parte di codifica avendo già pronte delle interfacce verso le reti sociali. Inoltre grazie alla possibilità di studiare le API di collegamento attraverso la ottima documentazione online è stato possibile progettare i client MoCA in modo che sfruttassero appieno le Direct Line API, tale argomento verrà largamente discusso in una sezione dedicata.

5.3 MoCA Client

Avendo deciso di sfruttare un sistema a microservizi, il bot ha la naturale necessità di comunicare con il software prodotto dagli studenti/programmatori, la scelta da effettuare in questo caso è duplice:

- Realizzare un sistema di API a cui collegare ogni servizio attraverso connessioni internet, il collegamento e la programmazione alle api deve essere gestita da chi realizza il servizio allungando e complicando il livello realizzativo del servizio.
- Creare un client già predisposto alla comunicazione con il BOT, che tramite una SDK lancia i servizi necessari e comunica attraverso un protocollo semplice e ben strutturato.

Basandoci sul fatto che i servizi devono poter essere testati su qualunque tipo di macchina, dal server al PC casalingo, essere gestibili direttamente dall'utente che li ospita ed essere facilmente integrabili, quindi dare la possibilità di realizzare applicazioni anche molto semplici sfruttando una SDK dedicata, si è optato per la seconda opzione realizzando il MoCA Client (Montessoro Connector Application).

I principali vantaggi di questa tecnica sono:

- Possibilità di astrarre il modello di comunicazione rendendolo completamente indipendente dalla tipologia di rete a cui si devono appoggiare i servizi: sia terminali che sfruttano IP pubblici che collegati attraverso NAT possono condividere i servizi con tutti gli utenti che utilizzano il bot
- Instaurare una connessione protetta basata su TLS: i messaggi inviati dai client MoCA al BOT sono crittografati end-to-end tramite connessioni sicure
- I dati trasmessi sono controllati e limitati direttamente dal Client MoCA, quindi è limitato anche l'invio di dati o routine potenzialmente dannose per il BOT stesso
- Programmi e funzionalità realizzate dagli studenti/programmatori non devono essere caricati su server o macchina virtuale per essere avviati, ma sono testabili direttamente dal file compilato di progetto (se il progetto ha determinate caratteristiche) o dopo una copiatura nella cartella adibita a contenere l'eseguibile.
- Lancio dei servizi: l'avvio di un servizio avviene quando un utente ne fa richiesta. L'avvio di un servizio è completamente trasparente allo sviluppatore: viene comunicato il PID del servizio lanciato dal client MoCA al programmatore in modo tale che l'utente stesso possa arrestare il programma o monitorare in tempo reale l'utilizzo delle sue risorse.
- Gestione dei deadlock: se i servizi svolgono delle operazioni errate oppure entrano in deadlock, il client MoCA prevede dei tempi di timeout che permettono di arrestare un servizio in caso questo non risponda più a nessuno stimolo; inoltre lo comunica al programmatore/studente grazie ad un avviso in console.
- Possibilità di configurare facilmente il client tramite un file di configurazione o attraverso linea di comando.

Principali svantaggi:

- Maggiore tempo di sviluppo dell'applicazione MoCA Client.
- Limitazione delle piattaforme integrabili all'interno del sistema almeno in un primo momento di lancio dell'applicativo.
- Limitazione della variabilità dei dati inviati dal servizio al BOT.

5.4 Luis

Per far dialogare il BOT con gli utenti umani è stato scelto di utilizzare i Cognitive Service di Microsoft. In particolare il servizio LUIS[9] (Language Understanding Intelligent Service). Tale servizio fornito da Microsoft permette di associare parole e frasi a determinati Intent previsti in fase di strutturazione delle specifiche. Questi Intent definiscono il significato della frase che l'utente ha digitato al bot. All'interno di un Intent possono essere presenti una o più Entities cioè delle parole che acquistano un significato particolare all'interno della frase. Per ogni Intent sono state inserite delle frasi all'interno del portale dedicato che hanno il compito di istruire l'intelligenza artificiale sul significato che si vuole associarne.

Uno schema a blocchi del funzionamento del servizio LUIS è il seguente:

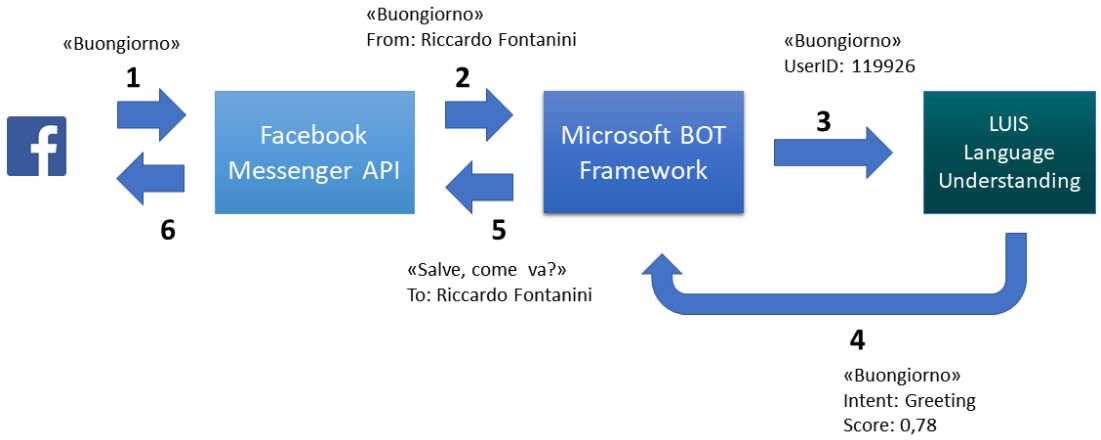


Figura 5.1 Schema a blocchi di interazione tra utente-LUIS

1. Il messaggio scritto dall'utente viene inoltrato attraverso le API del social Network
2. Dalle API il messaggio viene inviato al core del nostro BOT
3. Il core decide la tipologia di messaggio e viene inoltrato a LUIS per la decodifica e per l'associazione ad un Intent
4. Luis ritorna la frase associata all'Intent tramite lo score, percentuale di "sicurezza" con la quale il servizio associa la frase all'Intent, se sono presenti Entities vengono elencati
5. Viene generata una frase di risposta dal core del BOT ed inoltrata alle API del Social Network
6. La risposta viene recapitata all'utente

Gli Intent definiti per le funzionalità base di MontessoroBOT sono:

- Explanation – sono associate le varie richieste di funzionamento del bot o la richiesta di aiuto
- Fonta – definisce la richiesta di credits o informazioni generali dell'autore del programma
- Greeting – frasi di saluto o di cortesia
- ModifyUserID – frasi atte a modificare l'identificativo utente associato alla persona che sta scrivendo al bot (cioè la matricola per gli studenti)
- None – frase non compresa
- Reception – fa una ricerca sul sito dell'Università di Udine per sapere quando il professore ha ricevimento
- StartService – definisce le frasi per il lancio di un servizio associato all'UserID impostato
- Feel – frasi di cortesia
- StopDialog frasi atte alla chiusura forzata di una convesazione (per sviluppatore)

Per rendere meglio l'idea riporto l'esempio di un saluto.

[9] https://www.luis.ai

Intent Greeting, frasi associate:

- Buona giornata
- Salve
- Ciao

Cioè quando un utente scrive al bot un saluto come “Salve” o “Buongiorno”, i caratteri digitati vengono inviati attraverso REST API all’interfaccia LUIS. Essa grazie alla rete neurale di cui è fornito, associa una probabilità alla frase in ingresso di appartenere ad un determinato Intent. Per esempio la frase “buongiorno”, come si vede dalla figura seguente, secondo i parametri inseriti ha il 78% di probabilità di essere un saluto, il 2% di essere una richiesta di credits e così via.

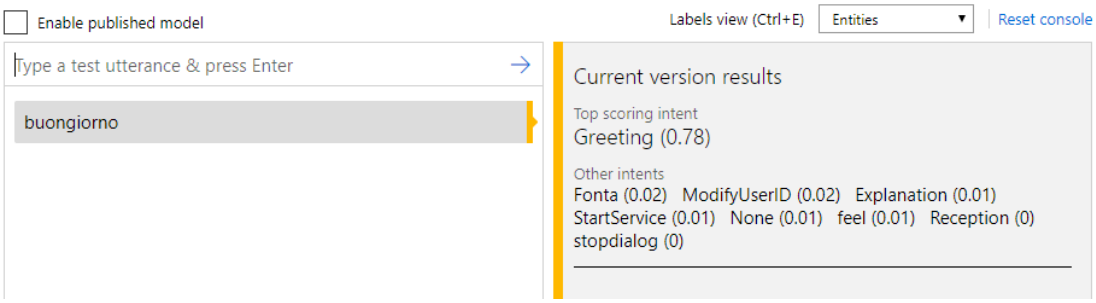


Figura 5.2 Messaggio con corretta assegnazione dell'Intent

Ovviamente più intent definiamo, maggiori possono essere le funzioni implementate dal BOT. Ci sono però casi in cui la rete neurale fallisce o non riesce a discernere efficacemente una frase, per esempio possono presentarsi errori di sovrastima, come riporto nell’immagine sottostante.

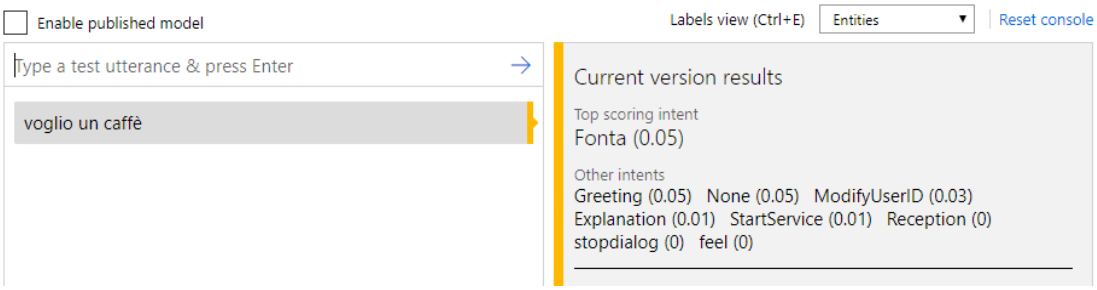


Figura 5.3 Messaggio con assegnazione errata dell'Intent

In questo caso infatti la frase “voglio un caffè” è stata associata ad una richiesta di credits, invece avrebbe dovuto rientrare nella categoria delle funzioni non definite None. Si nota che però è stato assegnato uno score molto basso a tale frase: pari al 5% di probabilità che si tratti di una richiesta di credits. Grazie allo score ed un controllo sugli Intent in uscita da Luis è possibile proteggere le proprie applicazioni da errori di sovrastima. Per l’implementazione in codice delle nozioni appena elencate si rimanda alla relativa sezione di descrizione degli algoritmi. Per l’inserimento dei vari parametri e il processo di training della rete neurale si è sfruttato il portale di Luis[10] che permette di sfruttare a pieno le potenzialità di questo servizio.

5.4.1 Estensibilità con Microsoft Cognitive Service

LUIS è solo una delle svariate applicazioni che il pacchetto di Microsoft “Cognitive Services” mette a disposizione. Tali servizi sono accumunati dal fatto che sfruttano l’intelligenza artificiale per compiere determinati compiti. I servizi sono molteplici e voglio citarne alcuni:

- API visione artificiale, estrae informazioni di utilità pratica da una immagine

[10] http://luis.ai

- API Traduzione vocale Microsoft Translator, cioè permette di eseguire traduzioni vocali in tempo reale attraverso chiamate API
- LUIS, permette di comprendere il significato di frasi o parole in un determinato contesto
- API analisi del testo, valuta i sentimenti e gli argomenti per identificare con precisione le esigenze degli utenti
- API controllo ortografico di BING, rileva e corregge gli errori ortografici
- API consigli, prevede e consiglia gli articoli più appetibili

Di questi servizi solo LUIS è sfruttato a pieno nell'ambito di questa tesi, ma sono tutte facilmente integrabili per estendere le funzionalità del sistema. Tutti questi servizi lavorano attraverso REST API e possono essere sfruttati tramite Microsoft Azure, la piattaforma Cloud di Microsoft.

5.5 Conversazioni

In questa sezione analizzeremo come comunica il sistema a livello "concettuale" senza analizzare i protocolli utilizzati ma considerando solamente l'informazione trasmessa, tale scelta permette di focalizzarsi maggiormente sulla logica che sta dietro ad ogni comunicazione senza considerarne i tecnicismi. "Come" viene trasmessa l'informazione viene demandato alla sezione protocolli.

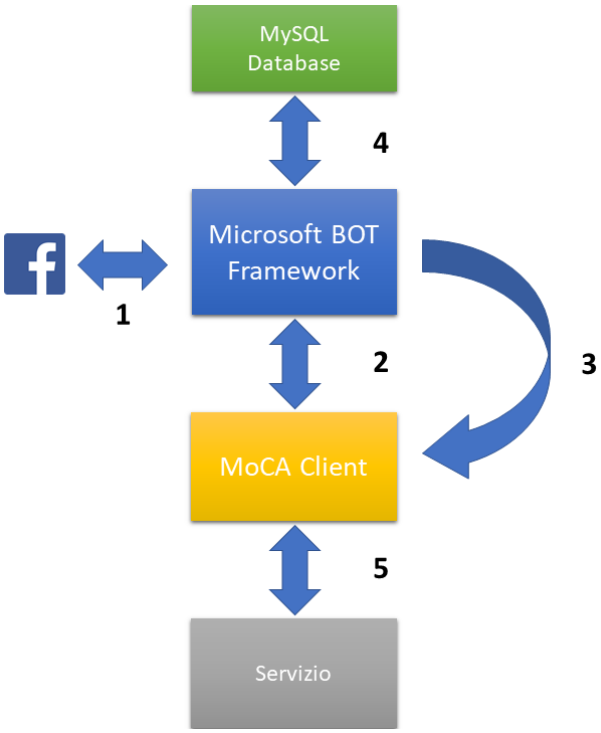


Figura 5.4 Schema a blocchi del flusso delle conversazioni

Il progetto prevede cinque tipologie di conversazioni differenti:

1. Utente – Social Network – BOT, che viene gestita tramite REST API ma non è di particolare interesse in questo capitolo siccome non trasporta informazioni di particolare rilevanza
2. BOT – MoCA Client (conversazione principale), trasporta le informazioni di connessione e di avvio dei servizi
3. BOT – MoCA Client (conversazione ausiliaria), trasporta tutti i messaggi che dall'utente devono propagarsi al servizio e viceversa
4. MoCA Client – MySQL Database, rappresenta la pubblicazione delle informazioni e dei servizi di ogni MoCA Client (analizzato nella sezione Database MySQL)
5. MoCA Client – Servizio conversazione tra un servizio avviato ed il MoCA Client

Con il seguente schema invece vogliamo visualizzare la disposizione temporale delle informazioni comunicate dal BOT ai vari componenti del sistema.

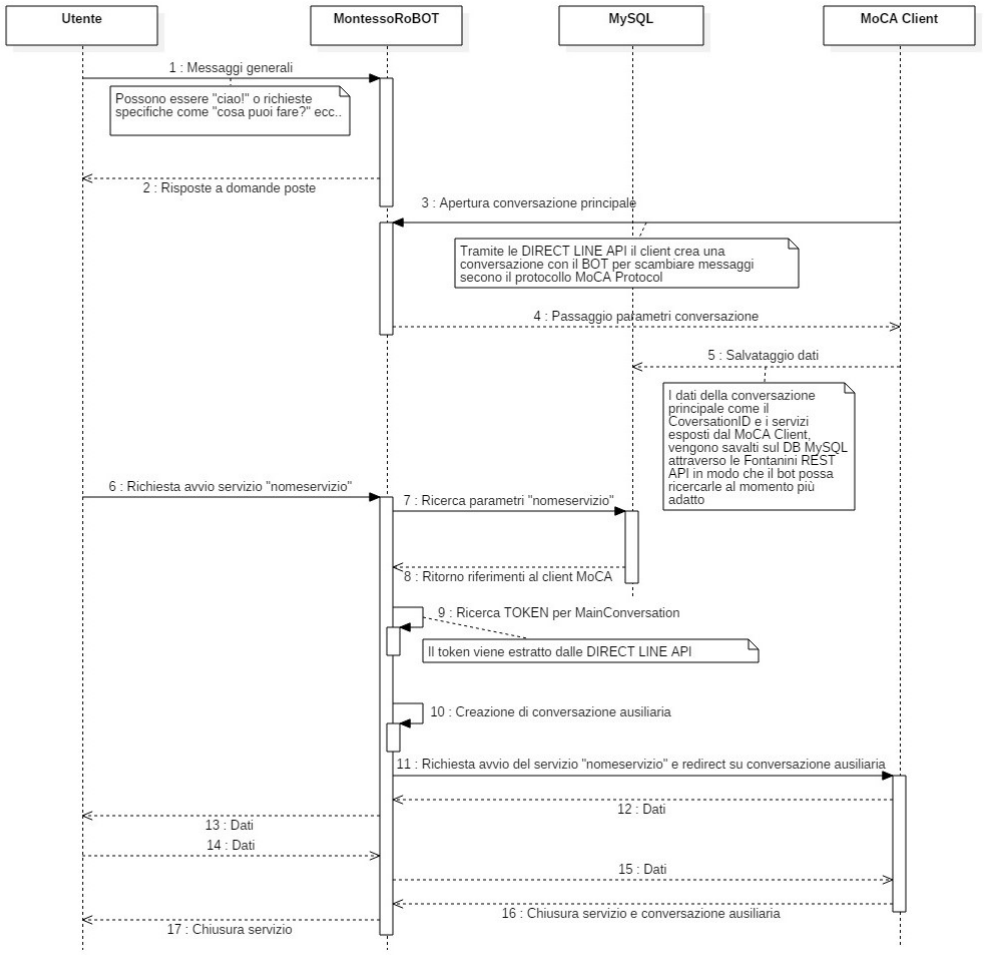


Figura 5.5 Diagramma di sequenza dei messaggi del sistema

All'interno del BOT core vengono smistate le conversazioni rispetto al contenuto del messaggio stesso e della dello stato in cui si trova il BOT:

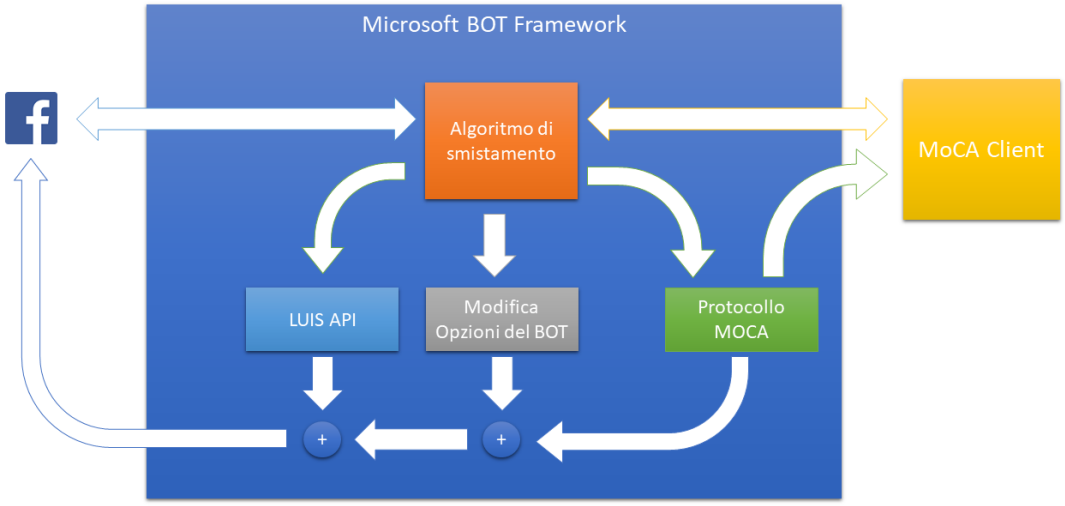


Figura 5.6 Diagramma dei flussi dei messaggi all'interno del BOT Core

Come si vede dall'immagine, tutti i messaggi passano per un algoritmo di smistamento che discerne tra le varie tipologie di messaggi:

- Messaggi da inviare alla decodifica tramite intelligenza artificiale (Servizio LUIS)
- Messaggi per la modifica delle opzioni del bot (riservato allo sviluppatore)
- Messaggi del protocollo MOCA, questi messaggi non vengono condivisi sui social network ma al contrario sono necessari per la comunicazione M2M, cioè tra BOT core e MoCA client.

5.6 BOT – MoCA Client Principale

In tale conversazione viaggiano le informazioni per sincronizzare il Client MoCA con BOT core. Le modalità e tipologie di messaggi sono definite dal protocollo MOCA descritto nell'apposita sezione. A livello concettuale questa è la comunicazione in cui il BOT core comunica al MoCA Client quale servizio avviare e quando terminarlo. Inoltre al suo interno vengono inviate le richieste di apertura di una conversazione collaterale.

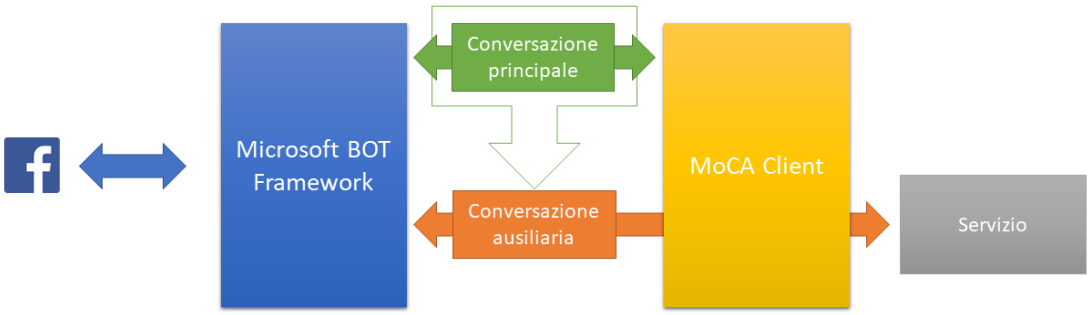


Figura 5.7 Schema a blocchi delle conversazioni

Nello schema si vede che grazie ai messaggi scambiati dalla conversazione principale viene aperta una conversazione ausiliaria (chiamata anche "collaterale") che attraversa il MoCA Client per arrivare al servizio e comunicare direttamente con esso.

5.7 Database MySQL

Per pubblicare le informazioni inerenti ai servizi associati ad un UserID (Matricola per gli studenti), il MoCA client inoltra verso le API del database MySQL una chiamata GET contente i parametri:

- Function: funzione della richiesta, se "insert" allora i dati degli altri parametri verranno inseriti nel database, se "get" allora verrà selezionato un record relativo allo UserID specificato
- UserID: id o matricola della persona che pubblica i servizi
- ConversationID: id della conversazione dove pubblicare i messaggi per comunicare con quello specifico client
- Services: lista dei servizi associati al client

La chiamata completa è:

```
http://riccardofonta.altervista.org/montessorobot/?Function=<funzione>&UserID=<UserID>&MoCAConversationID=<mocaconversation>&Services=<servizio1>,<servizio2>,<servizio3>
```

Un esempio di pubblicazione dei dati è:

```
http://riccardofonta.altervista.org/montessorobot/?Function=insert&UserID=119926&MoCAConversationID=sc15sdc443zdcsdw4&Services=ciao,sendandreceive
```

Un esempio di selezione dal database:

```
http://riccardofonta.altervista.org/montessorobot/?Function=get&UserID=11992
```

In uscita avremo una risposta http formattata in codice JSON con tutte le informazioni richieste:

```
{
  "UserID" : "<userID>",
  "MoCAConversationID" : "<ConversationID>",
  "Services" : ["<servizio1>", "<servizio2>", "<servizio3>"]
}
```

Per esempio:

```
{"UserID":"119926","MoCAConversationID":"sc15sdc443zdcsdw4","Services":["ciao"]}
```

Tale API è stata scritta in codice PHP (descrizione in seguito) e sfrutta una sola tabella in un database MySQL formattata in questo modo:

#	Nome	Tipo	Codifica caratteri	Attributi	Null	Predefinito	Extra	Azione
1	id	int(10)		UNSIGNED	No	*Nessuno*	AUTO_INCREMENT	Modifica Elimina Primaria Unica Indice Più
2	UserID	varchar(50)	latin1_swedish_ci		No	*Nessuno*		Modifica Elimina Primaria Unica Indice Più
3	MoCAConversationID	varchar(100)	latin1_swedish_ci		No	*Nessuno*		Modifica Elimina Primaria Unica Indice Più
4	Services	text	latin1_swedish_ci		Sì	*NULL*		Modifica Elimina Primaria Unica Indice Più

Figura 5.8 Rappresentazione del database MySQL

- ID è un intero, senza segno, non nullo, auto-incrementate, di chiave primaria che identifica univocamente il record selezionato
- UserID è un VARCHAR di 50 caratteri, non nullo che identifica l'ID dell'utente relativo a tale record
- MoCAConversationID è un VARCHAR di 100 caratteri, non nullo che identifica l'id della conversazione associata allo UserID che espone i vari servizi
- Services è un TEXT in cui sono contenuti i vari servizi.

6 MICROSERVIZI

Come descritto nell'articolo di Martin Fowler "Microservices"[11] lo stile architetturale a microservizi è un approccio particolare nello sviluppo di una applicazione. In generale lo sviluppo a sotto problemi non è affrontato come nel caso del software monolitico, in cui si tende a descrivere il problema, schematizzarlo, sintetizzarlo e codificarlo, racchiudendolo in una porzione di codice separata dal resto, ma in generale si tende a separare sia logicamente che fisicamente i vari sotto problemi: "*In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.*"

Ogni software di medie e grandi dimensioni generalmente è frutto del lavoro di più di una persona, cioè di un team di sviluppo, che è smembrato in aree geografiche diverse: tale situazione può non essere semplice da gestire. I microservizi vengono incontro a questa situazione sempre più diffusa in ambito industriale separando non solo il lavoro da svolgere ma anche le problematiche relative a tempi, infrastrutture e conoscenze. Nell'ambito di questa tesi i microservizi rivestono un ruolo fondamentale: essendo realizzati da studenti o programmatori indipendenti che non devono possedere nessun privilegio infrastrutturale gli uni rispetto agli altri, ogni servizio deve estendere il sistema senza appesantirlo o renderlo vulnerabile. Al contrario di un software monolitco, infatti, l'aumento del carico su un singolo servizio non appesantisce il sistema nella sua totalità: gli altri servizi che idealmente risiedono su macchine differenti non subiscono ripercussioni. Per esempio prendiamo un sistema formato da due microservizi, il servizio A che risiede in un server farm e il servizio B istanziato sul PC del programmatore, quindi si suppone in un ambito casalingo. Benché risiedano in due posizioni geografiche, connettive e infrastrutturali differenti, entrambi i servizi potenzialmente sono alla portata di tutti gli utenti abilitati all'utilizzo di quel servizio. Supponiamo che ad una determinata ora svariate centinaia di migliaia di richieste vengano inviate al servizio A (quello nel server farm). Tale servizio si blocca perché non è stata considerata in fase di progetto la possibilità di un tale numero di richieste, mentre il servizio che risiede nella rete casalinga del programmatore svolge ancora il suo servizio senza risentire degli eventi del servizio A.

Oltre a questo esempio banale, i microservizi ricoprono una importante utilità nel campo della diffusione di software libero. Pensiamo ad un apparato software, estensibile da liberi programmatori, a cui chiunque può accedere liberamente e chiunque per un motivo o per l'altro può inserire una funzionalità attraverso il linguaggio di programmazione che più aggrada o che permette di fornire le migliori prestazioni al servizio stesso.

Un altro utile aspetto dei microservizi è la facilità di passare dalla modalità development a quella di production. All'interno del ciclo di vita di un software monolitico, questo passaggio è critico in quanto devono essere testate tutte le componenti del software insieme con il rischio di sovraccaricare l'infrastruttura o che il software stesso non risulti efficace (o efficiente). D'altro canto, grazie alla pianificazione a microservizi, ogni componente viene testato singolarmente e se i parametri di efficienza ed efficacia sono rispettati per quel determinato microservizio allora viene direttamente inserito in production senza che gli altri servizi siano necessariamente pronti. Ovviamente l'efficienza generale del sistema ne potrebbe risentire, rispetto ad un software monolitico in cui tutte le librerie e algoritmi risiedono nello stesso luogo.

Un ulteriore punto a favore del sistema a microservizi rispetto ad uno monolitico è la manutenibilità: ogni servizio può essere reso pubblico su un server o su qualunque altro dispositivo, questa loro elasticità permette lo spostamento momentaneo di un determinato servizio da un elaboratore ad un altro senza dover interrompere la fornitura del servizio stesso. Tale caratteristica è molto apprezzata in quanto l'utente che sfrutta il servizio non si accorge del cambio di architettura che risulta quindi completamente trasparente all'utente finale. Per una azienda che distribuisce software pubblico, a cui potenzialmente ogni persona può accedere, è fondamentale tale caratteristica: pensiamo ad esempio che per risolvere un errore nel codice di un software monolitico, i sistemisti devono spegnere il servizio sfruttato magari da centinaia di migliaia di persone, fare l'upload del nuovo sistema e rimetterlo online. Il tempo necessario all'iter completo per compiersi porterebbe ad un ritorno negativo dal punto di vista sia di immagine che economico.

[11] https://martinfowler.com/articles/microservices.html

7 LOGGING E MONITORAGGIO

In tutta la vita di un software c'è la volontà da parte di chi struttura e gestisce il sistema di conoscere come esso risponde ad un determinato stimolo o come si evolve nel tempo, in particolare quando non tutte le sue parti sono sotto lo stretto controllo del progettista, come nel sistema progettato in questa tesi. Due sistemi fondamentali che sono stati impiegati in questo progetto sono il logging ed il device monitoring.

- Logging, si occupa di descrivere cosa fa un certo componente con un linguaggio comprensibile al progettista o all'utilizzatore. In genere la descrizione di un problema o semplicemente di una determinata azione viene scritta su un file (appunto file log che indica data e ora) o riprodotta a video, in modo che il sistemista comprenda il problema e operi per risolverlo. La tecnica di logging fornisce una informazione generalmente qualitativa del problema. Infatti può risultare inefficace in una visione totalitaria del sistema.
- Monitoraggio, con questa tecnica si registrano determinati parametri per un determinato lasso di tempo come ad esempio i dati in ingresso/uscita ad un server o la memoria allocata da un certo servizio. Chi gestisce il sistema grazie a queste informazioni ha una visione di insieme che permette di identificare i momenti critici o di stabilire quali contromisure prendere in previsione futura. Tale tecnica fornisce una informazione quantitativa del problema: perde senso se l'intervallo di tempo in cui la si analizza è troppo largo o troppo stretto. Pensiamo di voler osservare l'occupazione di memoria di un servizio: se prendiamo come riferimento un anno di dati, il grafico risultante potrebbe darci informazioni errate in quanto il tempo di utilizzo di un servizio in genere non supera l'ora e quindi perdere la visione del particolare. D'altro canto se osserviamo un periodo troppo breve in cui di fatto non è stato utilizzato nessun servizio, il risultato è comunque inefficace. Un ulteriore parametro da considerare è la granularità dei dati: cioè la quantità di dati da salvare ed il tempo di campionamento con cui acquisirli. Troppi dati potrebbero dare problemi in fase di elaborazione, mentre troppo pochi potrebbero perdere informazioni importanti come i picchi di utilizzo che rappresentano la vera minaccia per un sistema.

Utilizzando Microsoft Azure come base per il BOT core, il sistema integra già sistemi di monitoraggio che permettono di capire l'utilizzo di memoria o i dati in/out dal server su cui è installata la nostra applicazione. Fondamentale per il sistemista è la conoscenza e l'utilizzo della Dashboard dedicata che Azure mette a disposizione online, quindi accessibile da qualsiasi terminale/dispositivo. Con tale strumento è possibile visualizzare i dati in ingresso/uscita e fare un confronto:

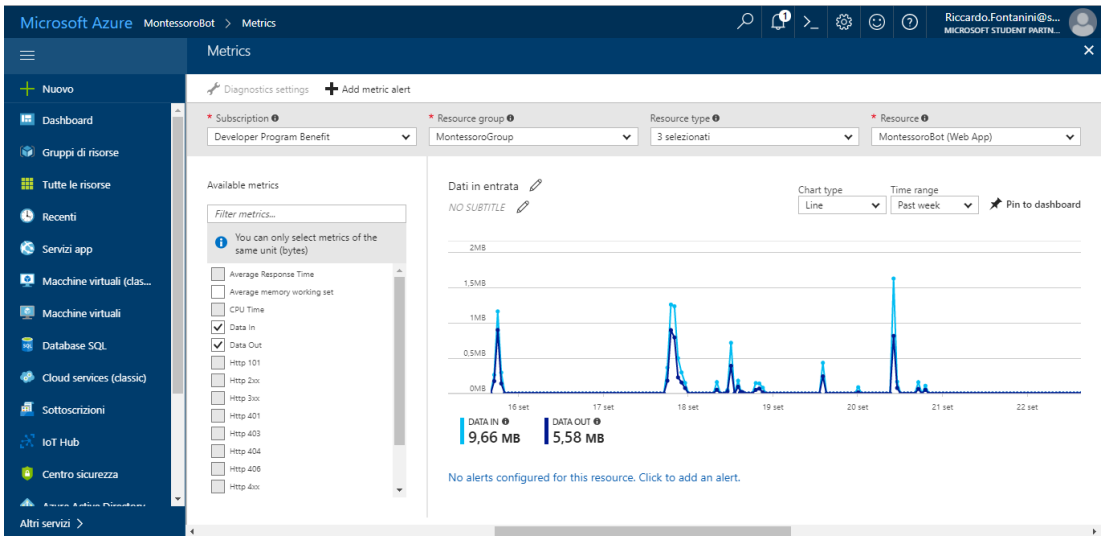


Figura 7.1 Pannello di controllo di Microsof Azure

Oppure l'allocazione di memoria del servizio:

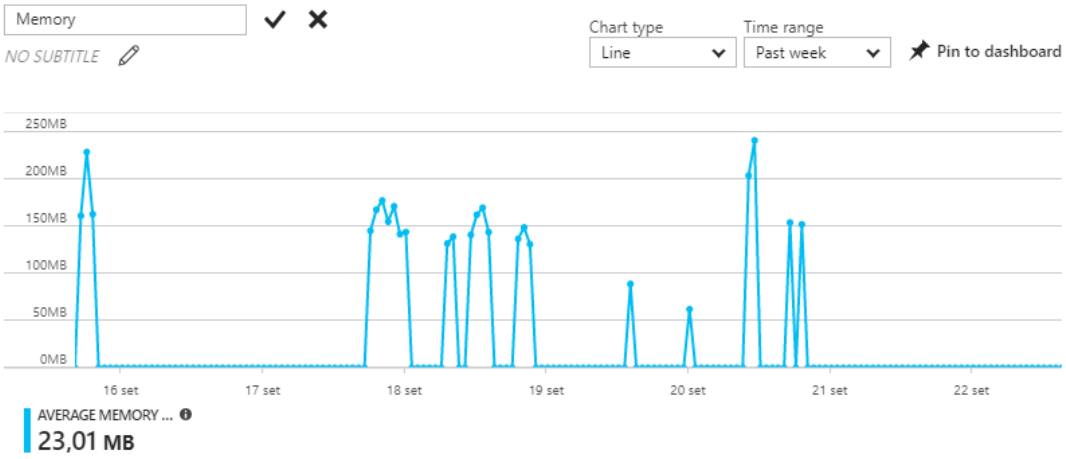


Figura 7.2 Allocazioe di memoria del BOT nel tempo

Oppure il tempo medio di risposta:

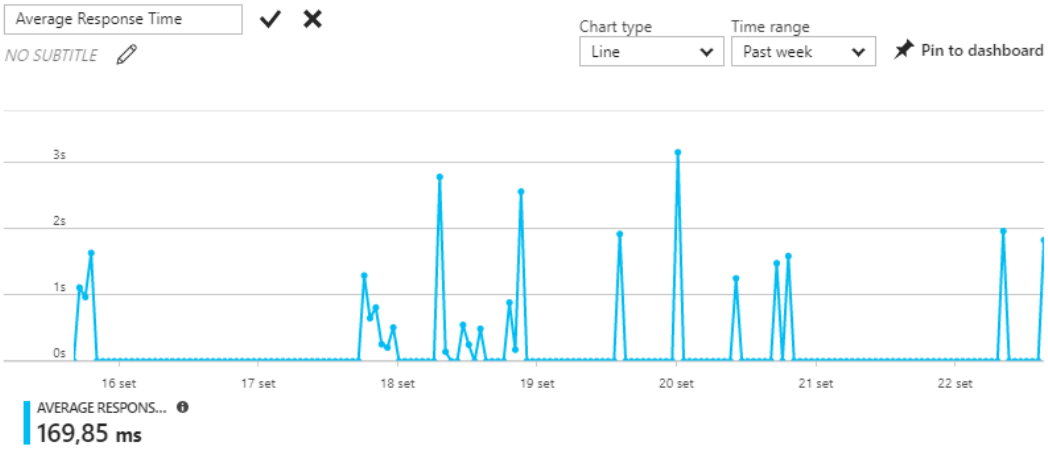


Figura 7.3 Tempo medio di risposta nel tempo

Tali dati possono anche essere messi a confronto e inseriti degli allert a soglia: se qualcuno dei parametri supera una certa soglia allora viene spedito un messaggio al sistemista che avverte di una anomalia in uno o più parametri.

8 PROTOCOLLI

In questa sezione andremo ad analizzare i protocolli impiegati dal sistema per comunicare con l'utente utilizzatore o programmatore ed i protocolli utilizzati per interconnettere tutti i servizi. In questa sezione non verranno analizzati tutti i livelli del modello ISO/OSI (o TCP/IP) in quanto il software sviluppato impiega principalmente il protocollo HTTP(s) su connessione TCP/IP, per scambiare messaggi o file. Il funzionamento dei livelli sottostanti al livello applicativo è demandato a letture più specifiche, in questa sede analizzeremo solamente protocolli di livello applicativo in quanto sono stati oggetto di analisi e conseguente rielaborazione per meglio adattarsi al progetto.

8.1 Protocollo HTTP

La maggior parte dei messaggi scambiati dal sistema sfrutta come base HTTPs, in questa sezione andremo a descrivere il funzionamento base del protocollo e di come viene assicurata la sicurezza del messaggio stesso.

Non c'è migliore modo per definire tale protocollo che citare la RFC[12] (Request For Comments) dell'organo IETF (Internet Engineering Task Force) dedicata: *"The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers."*

Il protocollo http è un protocollo di tipo client/server: il client esegue una richiesta al server e questo gli restituisce una risposta; vi sono quindi due tipi di messaggi http: uno di richiesta e uno di risposta.

8.1.1 Messaggio di richiesta

Quattro parti formano il messaggio di richiesta:

1. Riga di richiesta
2. Intestazione (header)
3. Riga vuota
4. Corpo del messaggio (body)

La riga di richiesta serve per identificare con precisione la risorsa e il tipo di richiesta che si fa al server, per questa è formata da un metodo, un URI (Uniform Resource Identifier) e la versione del protocollo che si sta impiegando. I metodi impiegati per la versione 1.1 di http possono essere:

- GET
- POST
- HEAD
- PUT
- DELETE
- TRACE
- OPTIONS
- CONNECT

Nel nostro caso i metodi che verranno utilizzati maggiormente sono GET e POST. Il primo permette di richiedere una risorsa formattando la richiesta come URI, mentre il metodo POST viene utilizzato, oltre che per richiedere informazioni, anche per inviarne al server salvando nella sezione body i parametri necessari.

All'interno dell'intestazione possono essere presenti svariati campi che hanno le funzioni più disparate, ne cito alcuni che sono per questo progetto di vitale importanza, demando a documenti specifici la descrizione di ogni campo intestazione previsto da IETF:

- Host, definisce il nome del server a cui ci stiamo collegando

[12] https://www.ietf.org/rfc/rfc2616.txt

- User-Agent, definisce il tipo di client che stiamo utilizzano
- Authorization, contiene le credenziali per autenticare il client con il server
- Accept-Charset, definisce il tipo di set di caratteri da impiegare
- Content-Lenght, definisce la lunghezza del body
- Content-Type, definisce il tipo di codifica del body

Il campo authorization ha una particolare sintassi che definisce prima il tipo di credenziali e poi il token delle credenziali:

```
Authorization: <type> <credentials>
```

I tipi definiti da IANA[13] (Internet Assigned Numbers Authority) sono:

- Basic
- Bearer
- Digiest
- Mutual
- OAuth
- SCRAM-SHA-1
- SCREAM-SHA-256
- HOBA
- Negotiate

Il tipo di autorizzazione che utilizzeremo è Bearer, essenzialmente definisce un token segreto che permette l'autenticazione del client, se il token è errato viene restituito un messaggio di errore di autenticazione. Rimando alla RFC[14] relativa per maggiori informazioni.

Il campo Content-Type nella maggior parte dei messaggi scambiati dal sistema sarà di tipo application/json:

```
Content-Type: application/json;
```

La maggior parte dei body conterrà informazioni formattate in JSON (per maggiori informazioni vedere sezione dedicata).

8.1.2 Messaggio di risposta

Il messaggio di risposta è formato da quattro parti fondamentali:

1. Riga di stato
2. Intestazione (header)
3. Riga vuota
4. Corpo (body)

La riga di stato fornisce una informazione fondamentale: il codice di stato http. Fondamentalmente è un codice a 3 cifre formattato in questo modo:

- 1xx: Informazione, è un messaggio informativo
- 2xx: Successo, la richiesta è stata soddisfatta
- 3xx: Re indirizzamento, non c'è risposta immediata, ma la richiesta è sensata e viene detto come ottenere la risposta
- 4xx: Errore del Client, richiesta errata
- 5xx: Errore del Server, la richiesta non può essere soddisfatta per un problema interno del server

[13] http://www.iana.org/assignments/http-authschemes/http-authschemes.xhtml
[14] https://tools.ietf.org/html/rfc6750

I codici più rilevanti per il nostro progetto sono:

- 200 OK. Il server ha fornito correttamente il contenuto nella sezione body
- 400 Bad Request. La risorsa richiesta non è comprensibile al server
- 500 Internal Server Error. Il server non è in grado di rispondere alla richiesta per un suo problema interno.

I campi intestazione più rilevanti per le nostre applicazioni, oltre che quelli descritti nella sezione di richiesta sono:

- Cache-Control, definisce il comportamento di caching
- Pragma, come Cache-Control ma utilizzato per retro-compatibilità con versioni precedenti al http 1.1
- Expires, data con la quale la risposta è considerata deprecata
- Server, definisce la tipologia di server a cui ci siamo collegati
- Strict-Transport-Security, forza l'uso di HTTPs invece che HTTP
- Date, data e ora della richiesta

Alcuni esempi di richiesta/risposta della nostra applicazione (in formato raw non crittografato) sono:

Richiesta di creazione di una conversazione (con token modificato)

```
POST http://directline.botframework.com/v3/directline/conversations
Authorization: Bearer Ay_aHU8nWyksdad.sdas6a6s4da4c48.sda4d6c3sac4.sdad
Content-Type: application/json
```

Risposta

```
201 Created
Cache-Control:  no-cache
Pragma:  no-cache
Content-Length:  566
Content-Type:  application/json; charset=utf-8
Expires:  -1
Server:  Microsoft-IIS/8.0
X-Powered-By:  ASP.NET
Strict-Transport-Security:  max-age=31536000
Date:  Tue, 29 Aug 2017 22:10:10 GMT

{
  "conversationId": "5sOn2tRHZP2DTY5xc3n2Om",
  "token": "Ay_aHU8nWyk.dAA._Xu0.mdbF2NdflMav6213s5cnMfgSEftdE",
  "expires_in": 1800,
  "streamUrl":
"wss://directline.botframework.com/v3/directline/conversations/5sOn2tRHZP2DT
Y5xc3n2Om/stream?watermark=-&t=Ay_aHUdcs31.dAA.c5",
  "referenceGrammarId": "3cc4bd36-8112-8850-b569-c75194b357d6"
}
```

Richiesta dei messaggi della conversazione

```
GET
https://directline.botframework.com/v3/directline/conversations/2H3xdfWmqkuBJ4lJT9Ka2s/activities
Authorization: Bearer Ay_aHU8nWyk.cwA.xv0.3Q-n11juq0vr4h9f4AZ89B4idQCHJ-bVJF-cA0S3kTI
```

Risposta affermativa con nessun messaggio nel body

```
200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Length:  144
Content-Type:  application/json; charset=utf-8
Content-Encoding:  gzip
Expires:  -1
Vary:  Accept-Encoding
Server:  Microsoft-IIS/8.0
X-Powered-By:  ASP.NET
Strict-Transport-Security:  max-age=31536000
Date:  Tue, 29 Aug 2017 22:05:30 GMT

{
  "activities": []
}
```

Richiesta dei messaggi di una determinata conversazione

```
POST
https://directline.botframework.com/v3/directline/conversations/5sOn2tRHZP2DTY5xc3n2Om/activities
Authorization: Bearer Ay_aHU8nWyk.cwA.xv0.3Q-n11juq0vr4h9f4AZ89B4idQCHJ-bVJF-cA0S3kTI
Content-Type: application/json
```

Errore del server

```
500 Internal Server Error
Cache-Control:  no-cache
Pragma:  no-cache
Content-Length:  51
Content-Type:  application/json; charset=utf-8
Expires:  -1
Server:  Microsoft-IIS/8.0
X-Powered-By:  ASP.NET
Strict-Transport-Security:  max-age=31536000
Date:  Tue, 29 Aug 2017 22:10:33 GMT

{
```

```
  "error": {
    "code": "ServiceError"
  }
}
```

8.1.3 TLS e crittografia

Ogni messaggio inviato da e verso il sistema che stiamo esaminando deve essere crittografato. La crittografia che generalmente viene applicata è quella relativa alla tecnologia HTTP**s**. Abbiamo parlato precedentemente del protocollo http e delle sue caratteristiche: nel modello TCP/IP tale protocollo occupa il livello applicativo. Quindi sembrerebbe che tra il protocollo http e il TCP non siano presenti altri protocolli.

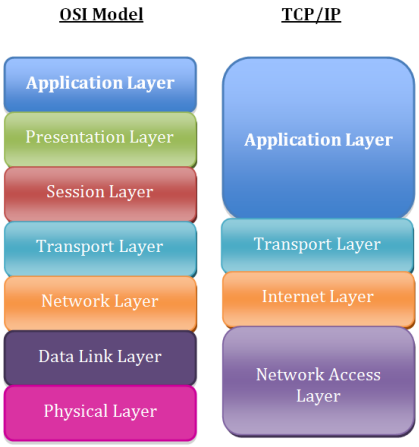


Figura 8.1 Modelli ISO/OSI e TCP/IP

In realtà esiste un protocollo "trasparente" all'utente tra http e TCP che serve a introdurre la crittografia di tutti i messaggi http spediti sulla connessione TCP precedentemente instaurata. L'https (HyperText Transfer Protocol over Secure Socket Layer) infatti sfrutta una connessione criptata SSL (secure Socket Layer, utilizzata fino a qualche anno fa) o TSL (Transport Layer Security correntemente utilizzato) per l'invio dei messaggi http. Ovviamente una trattazione completa della crittografia basata su queste tecnologie risulterebbe lunga e poco pertinente nell'ambito della tesi. In ogni caso voglio spendere alcune parole riguardo a tali tecnologie per convincere il lettore sull'efficacia del sistema di cui stiamo discutendo. Sostanzialmente, viene creato un canale di comunicazione criptato tra il client e il server attraverso uno scambio di certificati; una volta stabilito questo canale, al suo interno viene utilizzato il protocollo HTTP per la comunicazione. Strettamente parlando, HTTPS non è effettivamente considerato come un protocollo separato da HTTP, ma si riferisce appunto all'uso di quest'ultimo attraverso una connessione criptata SSL/TLS. Questo tipo di comunicazione garantisce che solamente il client e il server siano in grado di conoscere il contenuto della comunicazione.

Per maggiori informazioni riguardo al protocollo TLS si faccia riferimento al documento RFC 2246 o 5246[15]. L'utilizzo per un sistema a microservizi di una connessione crittografata è fondamentale in quanto il canale di comunicazione tra utente utilizzatore e servizio deve proteggere da attacchi esterni dati personali o parametri che circolano al loro interno.

8.1.4 Formato JSON

Il formato JSON è il formato principale con il quale sono formattati i body delle richieste o risposte http del sistema. JSON (JavaScript Object Notation) è un semplice formato per lo scambio di dati. Per le persone è facile da leggere e scrivere, mentre per le macchine risulta facile da generare e analizzarne la sintassi. È basato su un sottosistema del linguaggio di programmazione JavaScript. JSON è basato su due strutture:

- Un insieme di coppie chiave/valore

[15] https://tools.ietf.org/html/rfc5246

- Un elenco ordinato di valori

Maggiori dettagli si possono trovare nella pagina[16] dedicata, in ogni caso come XML viene spesso utilizzato per descrivere in modo semplice ed efficace dei dati provenienti da un server od un sistema informatico.

Un esempio di dati formattati in JSON sono:

```
{
  "employees":
    [
      { "firstName":"John", "lastName":"Doe" },
      { "firstName":"Anna", "lastName":"Smith" },
      { "firstName":"Peter", "lastName":"Jones" }
    ]
}
```

Che rappresenta un array di coppie chiavi/valori contenti nomi e cognomi di una serie di impiegati. Ogni oggetto JSON è separato da parentesi graffe, una virgola per separarlo dal un oggetto successivo ed identificato da una stringa chiave.

8.1.5 Direct Line API

Direct Line API versione 3.0 sono delle REST API sviluppate da Microsoft per l'interconnessione tra un client custom e il Microsoft BOT Framework. Tali API in genere vengono utilizzate per collegare un client a cui gli utenti accedono per interagire con il bot. In questo caso però verranno sfruttate per realizzare un collegamento Machine to Machine senza sfruttare tecnologie differenti a quelle che Microsoft ha previsto per il collegamento verso servizi esterni. Infatti il MoCA Client è stato fornito di algoritmi per la connessione alle Direct Line API e sfruttando esse crea i canali comunicativi tra BOT Framework e servizi degli studenti/programmatori. Tale scelta è dettata dal fatto che tali API sfruttano una connessione sicura e crittografata, supportano allegati come video o foto e sono facilmente integrabili all'interno del BOT Core; la parte più impegnativa del lavoro in tal senso è dovuto al fatto che non esistono SDK scritte in C, compatibili quindi con i MoCA Client. Ovviamente tali funzioni sono state scritte ex novo partendo dalla documentazione online. Sono stati utilizzati tutti i criteri di sicurezza e crittografia richiesti.

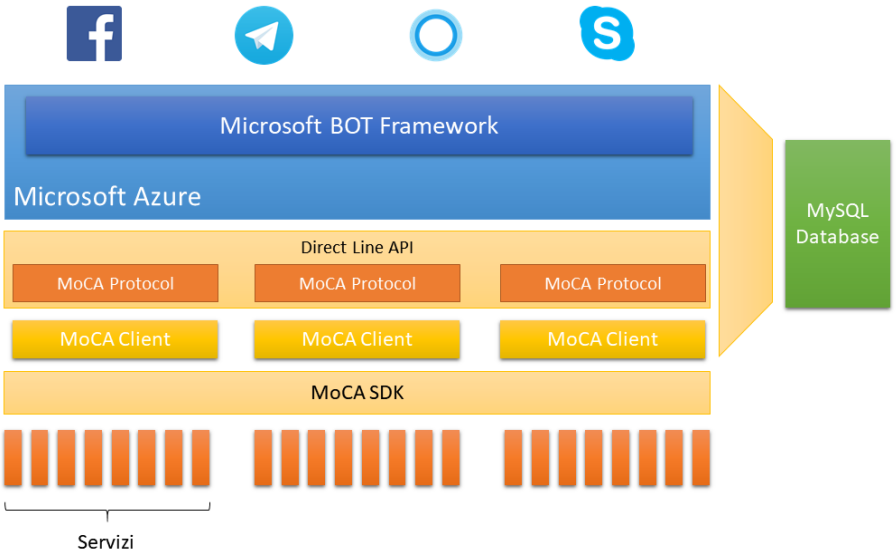


Figura 8.2 Schema a livelli del sistema generale

[16] http://www.json.org/json-it.html

8.1.6 Autenticazione

Per autenticarsi al servizio di Direct Line API è necessario ottenere un Secret di sicurezza che dovrà essere inserito in ogni programma che si vuole realizzare per connettersi alle funzioni base. Per ottenere tale Secret bisogna registrarsi sul portale di Azure e ottenere le credenziali per il portale di Microsft BOT framework[17]. Dopo aver ottenuto tale stringa, essa va inserita in ogni richiesta http nel campo authorization al fine di creare una conversazione, cioè un canale tra BOT e client:

```
Authorization: Bearer SECRET
```

I Secret sono dei segreti universali, cioè col tempo non scadono e hanno piena accessibilità ad ogni conversazione, al contrario dei token che invece servono per accedere ad un'unica conversazione. Gli algoritmi del MoCA Client utilizzano il secret per creare una conversazione mentre i token per accedere alle rispettive conversazioni. Siccome i token hanno una scadenza (espressa in secondi al momento della creazione della conversazione) hanno necessità di essere rigenerati, per fare cioè è necessario inviare una richiesta http di tipo POST formattata:

```
POST https://directline.botframework.com/v3/directline/tokens/refresh
Authorization: Bearer TOKEN_TO_BE_REFRESHED
```

Con body vuoto. Un token può essere rinnovato un numero infinito di volte.

Se la risposta è affermativa sarà del tipo:

```
HTTP/1.1 200 OK
…
{
  "conversationId": "abc123",
  "token": "RCurR_XV9ZA.cwA.BKA.y8qbbOFqbOF2vtCX7CZj0LdjAPGfiCpg4Fv0",
  "expires_in": 1800
}
```

8.1.7 Avviare una conversazione

La conversazione è un canale di comunicazione che viene instaurato tra Client e BOT. Entrambi possono inoltrare messaggi su una conversazione se essa è aperta. Per creare una nuova conversazione bisogna spedire un messaggio http con parametri:

```
POST https://directline.botframework.com/v3/directline/conversations
Authorization: Bearer SECRET_OR_TOKEN
```

Se la richiesta va a buon fine e la conversazione viene create ed il server può rispondere con un messaggio:

```
HTTP/1.1 201 Created
…
{
  "conversationId": "abc123",
  "token":
"RCurR_XV9ZA.cwA.BKA.iaJrFv0y8qbOF5xPGfiCpg4Fv0y8qqbOF5x8qbOF5xn",
  "expires_in": 1800,
  "streamUrl":
"https://directline.botframework.com/v3/directline/conversations/abc123/stre
am?t=RCurR_XV9ZA.cwA..."
}
```

[17] https://dev.botframework.com/

Si nota che viene fornito un conversationID e un token, che identificano rispettivamente l'identificativo della conversazione e il token con cui accedervi. Il campo expires_in indica il numero di secondi in cui il token viene considerato scaduto.

8.1.8 Mandare un messaggio nella conversazione

Prima di vedere la procedura di inoltro di un messaggio bisogna conoscere il concetto di Activity. Una Activity non è nient'altro che un oggetto JSON che possiede svariati campi, tra cui:

- Action, campo che definisce l'azione svolta o da svolgere dalla destinazione del messaggio
- Attachments, array di oggetti che descrivono gli allegati al messaggio che possono essere musica video o foto
- ChannelID, stringa che definisce univocamente il canale
- Conversation, oggetto che descrive la conversazione
- From, campo che descrive il client o il server da cui arriva il messaggio(mittente)
- Id, id del messaggio
- serviceUrl, URL che specifica la destinazione di canale
- text, contenuto testuale del messaggio
- type, definisce il tipo del messaggio

Per inviare un messaggio ad una determinata conversazione basta inviare una richiesta http formattata come segue:

```
POST
https://directline.botframework.com/v3/directline/conversations/abc123/activities
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.idjAPGfiCpg4Fv0
…
{
    "type": "message",
    "from": {
        "id": "user1"
    },
    "text": "hello"
}
```

Se l'invio del messaggio è andato a buon fine, un messaggio viene inoltrato dal server contenente il numero del messaggio della conversazione:

```
HTTP/1.1 200 OK
…
{
    "id": "0001"
}
```

8.1.9 Mandare un allegato multimediale alla conversazione

Spesso quando si comunica con un utente, all'interno dei messaggi bisogna allegare un file multimediale. Le DIRECT LINE API permetto di mandare un messaggio con contenuto un'immagine facendo una chiamata http POST con all'interno il contenuto del file. Per esempio se volessi invia un'immagine jpg:

```
POST
https://directline.botframework.com/v3/directline/conversations/{conversationID}/upload?userId={userID}
```

```
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qbOF5xnR2vtpg4Fv0
Content-Type: image/jpeg
Content-Disposition: name="file"; filename="badjokeeel.jpg"
…
[file content]
```

In questo porzione di richiesta `{userID}` rappresenta l'id utente a cui mandare l'allegato, e `{conversationID}` l'identificativo della conversazione a cui allegare il file

8.1.10 Ricevere messaggi dal bot

Ci sono due possibili modalità di ricezione dei messaggi dal bot, noi in questo ambito ne analizzeremo soltanto una, l'unica che è stata implementata all'interno del sistema MoCA Client.

Per effettuare la ricezione dei messaggi di una determinata conversazione basta inoltrare una richiesta http GET come segue. Si nota che il watermark rappresenta la soglia con cui selezionare determinati messaggi.

```
GET
https://directline.botframework.com/v3/directline/conversations/{conversationID}/activities?watermark={watermark}
Authorization: Bearer
RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qbOF5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
```

Con tale richiesta si vogliono ottenere tutti i messaggi dalla `{conversationID}` dal messaggio che ha come identificativo `{watermark}` in poi.

8.1.11 Chiudere una conversazione

Chiudere una conversazione è fondamentale quando non si vuole più utilizzare quel determinato servizio. Per terminare quindi una conversazione basta inviare un messaggio http POST, come se fosse un messaggio normalissimo, ma il contenuto deve possedere un campo `"type": "endOfConversation",` come segue:

```
POST
https://directline.botframework.com/v3/directline/conversations/abc123/activities
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qbjAPGfiCpg4Fv0
…
{
    "type": "endOfConversation",
    "from": {
        "id": "user1"
    }
}
```

Ovviamente trattandosi di un normalissimo messaggio, la risposta è analoga all'invio di un comune messaggio. L'unica differenza è che se viene chiusa la conversazione, non sarà più possibile pubblicare messaggi al suo interno.

8.1.12 Modificare un messaggio o cancellarlo

Le DIRECT LINE API, se il canale lo permette, danno la possibilità di modificare un messaggio o addirittura cancellarlo dalla conversazione.

Per modificarlo basta inviare un http PUT:

```
PUT /v3/conversations/{conversationId}/activities/{activityId}
```

O per cancellarlo basta un http DELETE:

```
DELETE /v3/conversations/{conversationId}/activities/{activityId}
```

8.2 MoCA Protocol

Serve per gestire la comunicazione tra MoCA Client e Server BOT. Tale protocollo viene impiegato nella conversazione principale, aperta dal MoCA Client, per accederne ai servizi. È un protocollo che si appoggia sui messaggi scambiati tra BOT e Client, i quali sfruttano le DIRECT LINE API, quindi l'header e il payload di tale protocollo sono inseriti all'interno del campo Text del JSON di ogni activity della conversazione principale.

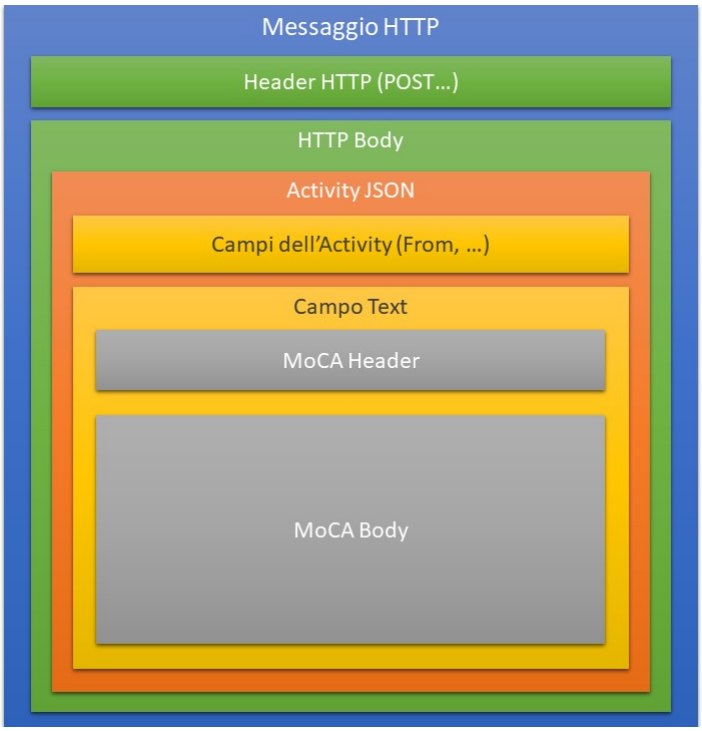


Figura 8.3 Imbustamento multiplo per il MoCA Protocol

Come si può vedere si sfrutta l'imbustamento multiplo per l'invio dei messaggi. Anche se può risultare ostico ad una prima occhiata, tale protocollo favorisce la leggibilità del messaggio anche ad occhio umano, fondamentale in fase di debugging, testing e manutenzione. La sicurezza del dato è garantita dal livello sottostante HTTPS che obbligatoriamente sfrutta la crittografia SSL/TLS.

8.2.1 Header

Tutti i messaggi del protocollo iniziano per * e sono seguiti da una parola che identifica la provenienza del messaggio:

- ***MOCACLIENT** identifica che il messaggio proviene dal client
- ***MOCABOT** identifica che il messaggio proviene dal BOT

Tale parola è seguita da un'altra che definisce l'operazione da fare, o che è stata fatta, dal rispettivo dispositivo. Tali parole sono intervallate da uno spazio.

8.2.2 Body

All'interno del body sono contenute le azioni da svolgere (o che sono state svolte) e i parametri da eseguire. Si definiscono per il Client MoCA:

- **STARTMAINCONVERSATION**, non è seguito da nessun altro parametro ed è il primo messaggio della conversazione in assoluto
- **CONNECTEDCOLLATERAL <idcollataeralconversation>**, notifica che il client si è connesso alla conversazione collaterale creata in precedenza dal BOT

- **ERRORCOLLATERAL <idcollataeralconversation>**, notifica che il client ha riscontrato un errore nella conversazione collaterale creata in precedenza dal BOT
- **DISCONNECTEDCOLLATERAL <idcollataeralconversation>**, notifica che il client si è disconnesso dalla conversazione collaterale creata in precedenza dal BOT

Es.

*MOCACLIENT CONNECTEDCOLLATERALCONVERSATION asnoanca41asc481cacsa6

Vuol dire che il client si è connesso con successo alla conversazione collaterale "asnoanca41asc481cacsa6".

Si definiscono per il BOT:

- **CREATEDCOLLATERAL <idcollateralconversation> <tokencollateralconversation> <nameservice>**, il bot vuole comunicare che ha creato una conversazione collaterale di ID "idcollateralconversation", token "tokencollateralconversation" per far in modo che il Client avvii il servizio "nameservice" e reindirizzi le informazioni di tale servizio su quella conversazione collaterale
- **DESTROYCOLLATERAL <idcollateralconversation>**, il bot non è più in ascolto di tale conversazione ed ha distrutto la conversazione

Es.

*MOCABOT CREATEDCOLLATERAL asnoanca41asc481cacsa6 4z<6xs8xsa684asx4<zx.461354 servizio1

Vuole significare che il bot ha creato una conversazione e sta aspettando che il client avvii il servizio e reindirizzi tutte le informazioni del programma nella conversazione.

9 MoCA SDK

MoCA SDK (software development kit) fornisce gli strumenti per l'interazione tra un MoCA Client ed un servizio creato da un programmatore o uno studente. Per questa tesi è stato sviluppato solamente la versione "C" cioè si possono sfruttare gli algoritmi di MoCA SDK solo se si sviluppa in linguaggio C. In ogni caso, essendo estensibile a qualunque linguaggio di programmazione la possibilità di integrare un servizio, basta solamente riscrivere tale SDK nel linguaggio desiderato e poi ricompilarla. Di fatto questa SDK è solamente una facilitazione per gli sviluppatori: aiuta solamente a sviluppare il proprio programma più velocemente, ma di fatto con le nozioni che verranno accennate più avanti, sarà possibile modificare, rivedere e migliorare questa SDK o svilupparne una diversa versione che permetta di utilizzare diversamente l'integrazione con il MoCA Client.

In principio l'algoritmo che sta alla base del sistema di scambio messaggi era basato sul redirect dello standard input/output sfruttando le pipe, però a causa di problematiche di sincronizzazione, si è deciso di sfruttare i file come sistema di comunicazione.

9.1 Integrazione con MoCA Client (base della SDK)

Un programma (o servizio) per interagire con il MoCA Client deve poter leggere o scrivere su file, inoltre deve poter accettare un parametro in ingresso che rappresenta un identificativo dell'avvio del servizio. Tale identificativo è una stringa in cui è contenuto un valore univoco per ogni avvio del servizio, normalmente coincide con la conversationID collaterale avviata dal MoCA Client. Definiremo fino alla fine della sezione tale parametro come conversationID.

Una volta acquisito il conversationID il programma deve creare due file nella cartella **/tmp/**:

- **<conversationID>MOCAread.txt** file dove vengono scritti i messaggi inoltrati dal MoCA Client verso il servizio
- **<conversationID>MOCAwrite.txt** file dove vengono scritti i messaggi dal servizio verso il Client MoCA /utente

Ogni volta che viene letto un file, il suo contenuto deve essere cancellato e le risorse vengono rilasciate in attesa che vengano scritti altri messaggi.

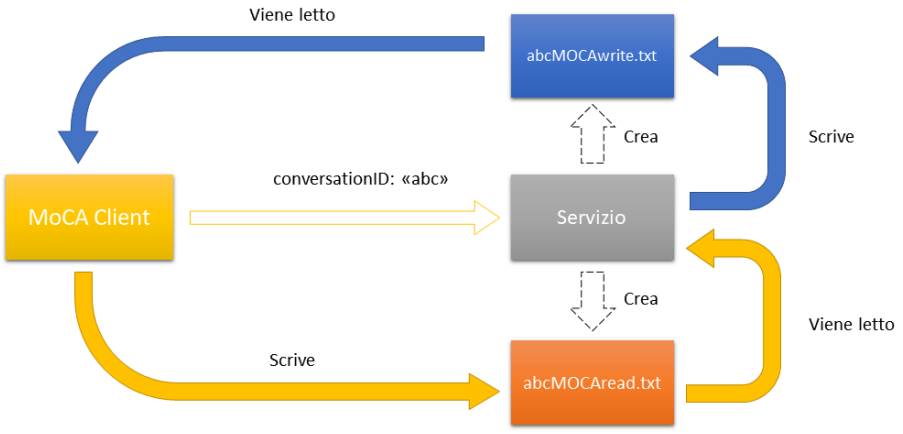


Figura 9.1 Schema a blocchi della comunicazione tra MoCA Client e servizio

9.2 Descrizione MoCA SDK (versione C)

Il MoCA SDK versione C è formato da due file: MOCAAPI.c e MOCAAPI.h.

Contiene varie costanti:

- _MOCAWRITEINBOT, in fase di scrittura imposta il file come destinazione
- _MOCAWRITECONSOLE, in fase di scrittura imposta la scrittura su standard output (printf)
- _MOCAREADFROMBOT, in fase di lettura imposta l'acquisizione da file

- _MOCAREADFROMCONSOLE, in fase di lettura imposta l'acquisizione da standard input (fgets)
- _MAXWRITEATTEMPTS, massimo numero di tentativi per la scrittura su file
- _USECONDSWRITING, microsecondi da aspettare se l'operazione di scrittura fallisce per riprovare al prossimo tentativo
- _MAXREADATTEMPTS, massimo numero di tentativi per la lettura di file
- _USECONDSREADING, microsecondi da aspettare se l'operazione di lettura fallisce per riprovare al prossimo tentativo

Contiene quattro metodi:

- **int mocaWrite(const char *buffer, int type)**, accetta in ingresso una stringa da scrivere, la destinazione di scrittura viene definita da type che può essere pari a _MOCAWRITEINBOT o _MOCAWRITECONSOLE, tale metodo serve per scrivere un messaggio al MoCA Client. In caso venga selezionato _MOCAWRITEINBOT, esegue una printf del buffer in ingresso
- **int mocaRead(char *buffer, int sizeBuffer, int type)**, accetta in ingresso un buffer, la lunghezza del buffer e viene definito da type se leggere su file o da standard input se type è rispettivamente _MOCAREADFROMBOT o _MOCAREADFROMCONSOLE. Nel caso venga selezionato _MOCAREADFROMCONSOLE, esegue un fgets dello standard input
- **void initMocaRW(char *fromMOCA)**, inizializza i parametri della libreria e crea i file nella cartella /tmp/
- **void mocaClose()**, chiude una conversazione.

Dal punto di vista implementativo la libreria non ha porzioni di codice particolarmente rilevante, comunque ogni funzione è stata commentata direttamente nel sorgente. Si vuole mostrare invece un esempio di come è semplice creare un servizio ed interfacciarlo al MoCA Client.

9.3 Esempio interfacciamento MoCA Client

Il seguente programma è scritto in linguaggio C e sfrutta le la MoCA SDK per l'interazione con il MoCA Client. Il funzionamento è molto semplice, manda un messaggio di testo al BOT ed aspetta che venga inviata una risposta, la quale verrà a sua volta inoltrata nuovamente al client.

```
#include <stdio.h>
#include <stdlib.h>
#include "MoCAAPI.h"
int main(int argc, char **argv)
{
    if(argc < 2)
        return -1;
    initMocaRW(argv[1]); //GET first parameter and setup communication
    mocaWrite("Messaggio di prova", _MOCAWRITEINBOT);//write message TO bot
    char buffer[500];
    mocaRead(buffer, 500, _MOCAREADFROMBOT); //write message TO bot
    mocaWrite(buffer, _MOCAWRITEINBOT); //retransmit message to bot
    mocaClose();//close communication
    return 0;
}
```

Il programma viene compilato con: `gcc -g -Wall -o sendandrecive sendandrecive.c MoCAAPI.c`

10 DESCRIZIONE SOFTWARE

In questo capitolo viene riportato il processo di sviluppo del sistema, in particolare vengono descritte le varie applicazioni e programmi realizzati. Non viene riportato tutto il codice, ma solamente alcuni script o funzioni di particolare interesse. Il restante codice è visionabile online[18] ed in fase di sviluppo è stato già sufficientemente commentato. Ci concentreremo principalmente sulla realizzazione del MoCA Client e del BOT, rispettivamente sviluppati in linguaggio C e C#.

10.1 IDE e strumenti di sviluppo

Per quanto riguarda gli strumenti di sviluppo si è deciso di sfruttare un sistema ibrido Microsoft – Linux. Ultimamente Microsoft ha aggiunto una funzionalità davvero interessante per gli sviluppatori software: il Windows Subsystem for Linux. Tale funzionalità permette di accede ad un sottosistema Linux direttamente dalla console di Microsoft Windows, proprio come se sfruttassimo una virtual machine. Tale sottosistema ha migliorato molto l'esperienza di sviluppo in quando il progetto prevede l'impiego del C# per realizzare il BOT core, che necessita di un OS Windows per debugging e sviluppo, mentre i Client MoCA devono essere compilati per sistemi Linux. Facendo coesistere tutti e due i sistemi sullo stesso OS la fase di sviluppo ha subito una notevole accelerazione anche perché accessibili entrambi da console dell'IDE di sviluppo.

Per la realizzazione del BOT Core si è impiegato Microsoft Visual Studio Community 2015[19], che oltre a supportare strumenti avanzati per il debugging e di auto-completamento del codice, è pienamente integrato a Microsoft Azure, caratteristica che ha semplificato di molto le operazioni di upload e testing del Core.

Per quanto riguarda il MoCA Client si è scelto di sfruttare le potenzialità del nuovo Microsoft Visual Studio Code. Tale IDE differisce da Visual Studio Community sotto diversi aspetti, ma forse i più importanti per quanto riguarda il progetto sono la leggerezza e l'elasticità. Tale software infatti permette di codificare in praticamente ogni linguaggio di programmazione esistente e tramite i comodissimi plug-in estenderne l'esperienza d'uso. Per quanto riguarda la compilazione del sorgente è stato utilizzato un makefile creato ad hoc per il progetto MoCA, scatenato attraverso un bash Ubuntu che sfrutta Windows Subsystem for Linux:

```
#Makefile previsto per il progetto MoCA
#written by Riccardo Fontanini
OBJECTS=
CC=gcc
CFLAGS= -g -Wall
FONTAMOCAPROTCLIENT = build/linux/fontamocaprotclient.o
FONTAUTILO= build/linux/fontautil.o
FONTACOMMUNICATIONO= build/linux/fontacommunication.o
FONTAHTTP = build/linux/fontahttp.o
FONTABOTCONNECTOR = build/linux/fontabotconnector.o
NAMEFILE= mocamain.c
PROGRAM_NAME=moca
LIBS = $(FONTAMOCAPROTCLIENT) $(FONTAUTILO) $(FONTABOTCONNECTOR)
$(FONTAHTTP) -lpthread -lcurl -ljson -std=gnu99
fontamocaprotclient:
    $(CC) $(CFLAGS) -o $(FONTAMOCAPROTCLIENT) -c libs/fontaMoCAProtClient.c
fontabotconnector:
```

[18] https://bitbucket.org/
[19] https://www.visualstudio.com/

```
	$(CC) $(CFLAGS) -o $(FONTABOTCONNECTOR) -c libs/fontabotconnector.c -
ljson -std=gnu99
fontahttp:
	$(CC) $(CFLAGS) -o $(FONTAHTTP) -c libs/fontahttp.c -lcurl -lpthread
fontacommunication:
	$(CC) $(CFLAGS) -o $(FONTACOMMUNICATIONO) -c libs/fontacommunication.c
fontautil:
	$(CC) $(CFLAGS) -o $(FONTAUTILO) -c libs/fontautil.c -lpthread
send:
	gcc -g -Wall -o build/linux/sendandrecive/sendandrecive
build/linux/sendandrecive/sendandrecive.c
build/linux/sendandrecive/MoCAAPI.c

$(PROGRAM_NAME): fontautil fontahttp fontabotconnector fontamocaprotclient
	@mkdir -p build/linux
	$(CC) $(CFLAGS) $(NAMEFILE) -o build/linux/$(PROGRAM_NAME) $(OBJECTS)
$(LIBS)
	@echo " "
	@echo "Compilazione completata! il file si trova nella cartella
build/linux!"
	@echo " "
```

Per il debugging dei messaggi in ingresso ed in uscita dal client con cui un utente utilizzatore si interfaccia al BOT è stato utilizzato il Bot Framework Channel Emulator. Tale emulatore è sostanzialmente un client sviluppato da Microsoft che permette di collegarsi ad un servizio anche in fase di debugging. Inoltre permette di analizzare il JSON dei vari messaggi: funzionalità molto utile per scoprire inesattezze o errori nel codice.

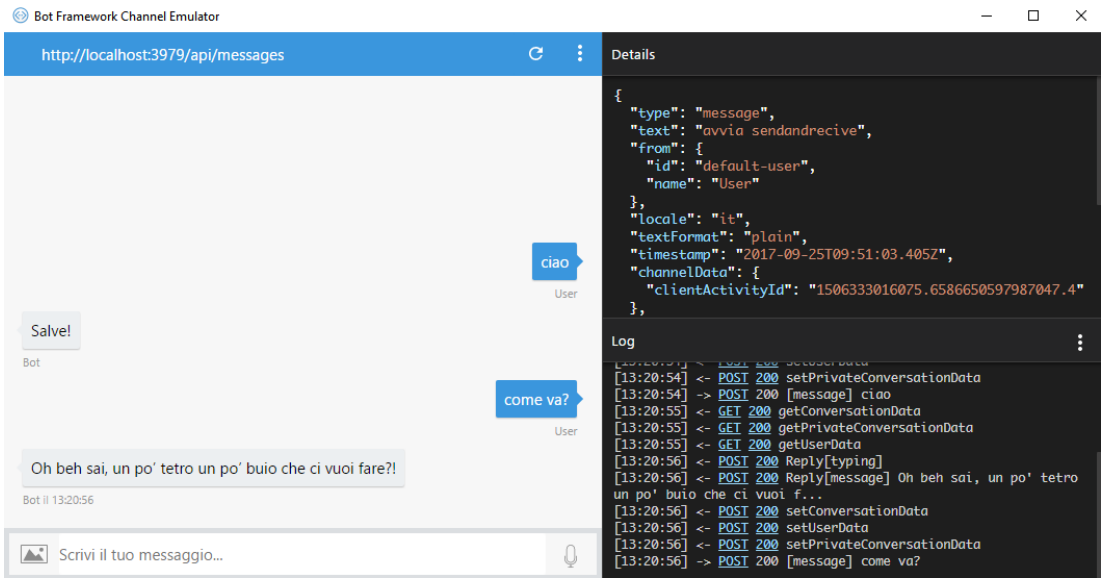


Figura 10.1 Bot Framework Channel Emulator Client

10.2 Git e supporto al codice

Per questo progetto si è voluto sfruttare un software di controllo versione distribuito chiamato git. Tale software permette di creare automaticamente delle copie di backup del codice ogni volta che si applicano delle modifiche e creare dei commit, cioè degli stadi di sviluppo con dei commenti dello stesso sviluppatore. Per il salvataggio dei file è stato utilizzato un servizio gratuito (per l'utilizzo privato) chiamato bitBucket[20] che oltre a visualizzare ad ogni modifica fatta sul codice permette di caricare e scaricare le diverse versioni dei propri algoritmi. Infatti in caso di errore irreparabile o a cui risulta difficile eseguire un fix, git permette di ritornare facilmente alla versione precedente del codice. Infatti grazie a delle chiamate push/pull tale software permette di caricare e scaricare una qualsiasi versione del proprio codice dovunque ci si trovi. Per maggiori informazioni si faccia riferimento alla pagina dedicata[21].

Author	Commit	Message	Date
Riccardo Fonta...	bf4cb35	Nuovo logo	2 hours ago
Riccardo Fonta...	8b388e3	codice formattato	13 hours ago
Riccardo Fonta...	97a449a	Modifiche paraziali per visualizzazione debugging	3 days ago
Riccardo Fonta...	de1f8c7	Funazionamento modificato con API per programmi esterni (solo C per il momento ...	2017-09-18
Riccardo Fonta...	781a23c	Funziona la comunicazione da client verso bot	2017-09-06
Riccardo Fonta...	b16687f	Insert mySQL API to save informaion	2017-09-03
Riccardo Fonta...	c5b9b74	PollBOT e gestione delle DIrect LINE API	2017-08-30
Riccardo Fonta...	cd68b27	Moca poller inserito	2017-08-24
Riccardo Fonta...	c6cd2d3	working http	2017-08-24
Riccardo Fonta...	ddfccbe	http	2017-08-24
Riccardo Fonta...	2b4343d	multithread	2017-08-23
Riccardo Fonta...	ce44da0	fontautili multithread	2017-08-23
Riccardo Fonta...	5bc0e6a	pthread implementati!	2017-08-21

Figura 10.2 Esempio di gestione delle modifiche codice

10.3 Sorgente MoCA Client

È un programma multithreading scritto in C per l'avvio di servizi progettati da studenti/programmatori in relazione ad uno stimolo proveniente dal BOT Core. Tale software si occupa di controllare i servizi esposti, comunicare con il BOT Framework e gestire l'apertura e la chiusura dei servizi richiesti.

Il sorgente del Client MoCA è costituito dal main-file più 5 librerie:

- **mocamain.c** è il main-file che si occupa di controllare le funzionalità ad esso collegate, avviare il sistema e fornire un feedback all'utente.
- **fontautil** è una liberiria ausiliaria dove sono contenute le macro che servono al programma più alcune funzioni di utilità generale
- **fontaMoCAProtClient** è la libreria che implementa il MOCA Protocol
- **fontahttp** libreria per richieste generiche http sia crittografate che non
- **fontacomunication** libreria ausiliaria per invio messaggi in AMQP, non utilizzata nella versione finale
- **fontabotconnector** libreria principale che implementa i servizi multi-thread e gestisce i servizi e comunica con il Bot Framework

10.3.1 mocamain.c

Espone sette variabili globali:

- **list**, è l'array dove sono contenuti i nomi di tutti i servizi del esposti dal client

[20] https://bitbucket.org
[21] https://git-scm.com/

- **nFoundFolders**, è il numero di servizi descritti nell'array list
- **absolutePath**, percorso assoluto del file esecutibile
- **UserID**, nome utente, in genere per gli studenti si utilizza la matricola
- **services**, è l'array di servizi. Viene allocato dinamicamente
- **lengthServices**, lunghezza dell'array di servizi
- **botconnectionparam**, parametri di connessione al BOT

Espone quattro funzioni:

- **void startscreen()**, visualizza il logo e le informazioni preliminari
- **void folderController()**, controlla le cartelle nella radice di avvio del programma, se una cartella non contiene un eseguibile allora genera un errore
- **int publishMySQLInfo(char *userid, char *mocaConversation)**, serve a pubblicare le informazioni al database MySQL attraverso le REST API. Di fatto manda un GET http alle API formattate secondo i parametri di ingresso
- **int startNewService(const char* nameservice, struct MoCAMessage calledMessage)**, serve ad avviare un nuovo servizio

10.3.1.1 startNewService

È la funzione fondamentale per avviare un servizio, accetta due parametri, il nome del servizio da avviare e il messaggio arrivato dal BOT framework che contiene tutti i parametri di avvio del servizio stesso. In caso di errore ritorna un intero negativo. È di fatto una funzione di callback, cioè viene scatenata solamente quando il BOT Framework richiede l'avvio di un servizio, la richiesta viene passata come calledMessage. Questa funzione lancia un thread al fine di avviare il servizio chiamando la funzione startService della libreria fontabotconnector.

```
int startNewService(const char *nameservice, struct MoCAMessage
calledMessage)
{
    int numberofservice = -1;
    if( (numberofservice = getEmptyService( &services, lengthServices)) <0 )
    {
        printf(ANSI_COLOR_RED "Error, no service available!"
ANSI_COLOR_RESET "\n");
        return -4;
    }
    printf("Found free service %d\n", numberofservice);
    if(insertControlNameService(list, nameservice,
&services[numberofservice]) < 0)//insert 1 service
    {
        printf(ANSI_COLOR_RED "Error, name of service not present or not
enough space!" ANSI_COLOR_RESET "\n");
        return -5;
    }
    //copy the connection parameters
    strcpy(services[numberofservice].collateralConversationID,
calledMessage.params[0]);
    strcpy(services[numberofservice].collateralConversationToken,
calledMessage.params[1]);
    strcpy(services[numberofservice].collateralConversationWatermark, "0");
```

```
    strcpy(services[numberofservice].userID , UserID);


    printf("Inserted name in services %d\nStarting service %d... \n",
numberofservice, numberofservice );
    //start new thread
    if( pthread_create( &(services[numberofservice].tid), NULL,
startService, (void *) &(services[numberofservice]) ) )
    {
        printf(ANSI_COLOR_RED "Error creating new thread!" ANSI_COLOR_RESET
"\n");
        return -6;
    }
    return 0;
}
```

Come si può notare sfrutta il multithreading grazie a `pthread_create` e alla libreria pthread.

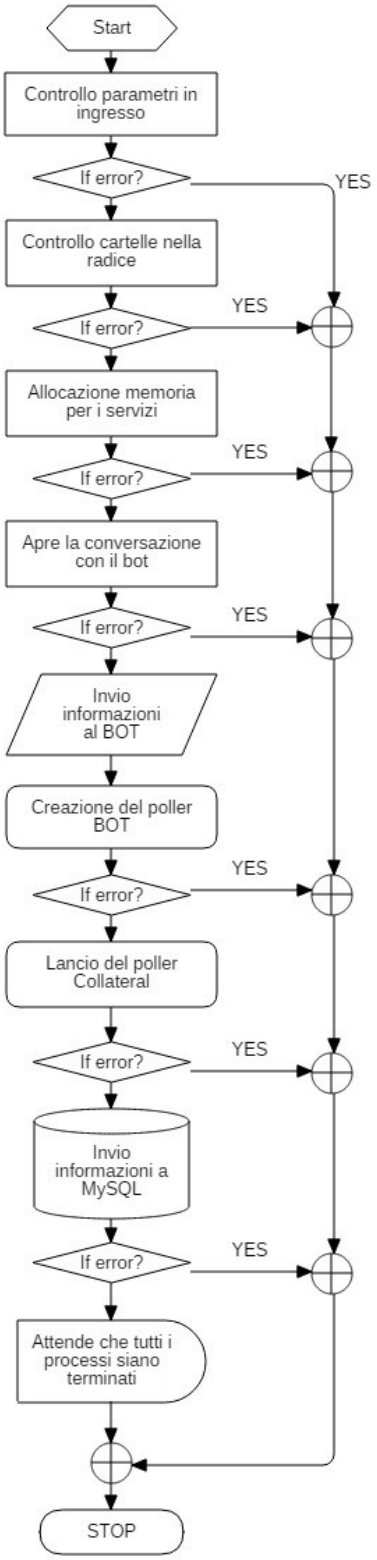


Figura 10.3 Flowchart del main del MoCA Client

10.3.1.2 Main

Qui a lato vengono illustrate sotto forma di flowchart le operazioni principali del main.

Mocamain accetta in ingresso vari parametri di configurazione:

- **-i <userid>** setta lo UserID per cui il client deve pubblicare le informazioni.
- **-id <userid>** come -i

Ritorna come codice errore un numero negativo se si verifica un problema. In caso di errore viene scritto in console la tipologia di errore e la descrizione in colore ROSSO.

Molte delle funzionalità del main derivano dalle librerie che andremo a discutere in seguito, in ogni caso volevo soffermarmi sull' apertura di un poller. Il poller di fatto è un processo che ad intervalli regolari esegue un determinato task. Può concludersi dopo un certo numero di iterazioni o lavorare a ciclo infinito. In ogni caso si tende sempre a definire una condizione di uscita: infatti nel caso in cui qualcosa non vada nel verso giusto un processo poller senza uscita potrebbe portare a sconvolgimenti del programma, crash o addirittura situazioni di deadlock.

Nel caso del poller BOT, il processo andrà a scaricare periodicamente i messaggi della conversazione per vedere se ce ne sono di nuovi, per fare questo il main lancia un thread che ha come parametri le opzioni di connessione al bot e la funzione da lanciare:

```
if( pthread_create(
&(botconnectionparam.pollerTID), NULL,
startPollBOT, (void *) &(botconnectionparam) )
)
    {
        printf(ANSI_COLOR_RED "Error creating
thread of poller!" ANSI_COLOR_RESET "\n");


        return -9;
    }
```

10.3.2 Fontautil

È una libreria ausiliaria in cui sono contenute le costanti, macro e funzioni di utilità generale per l'ambiente.

Espone dieci costanti:

- MAXNAMEFILE, massimo numero di caratteri per ogni nome di file eseguibile
- MAXSUBFOLDERS, massimo numero di servizi e sottocartelle
- MAXOPENPIDS, massimo numero di processi avviati simultaneamente
- MAXBUFFERSIZE, massimo numero di caratteri per il buffer in lettura del servizio
- TIMEOUTSECONDS, massimo numero di secondi da aspettare prima di uccidere il processo
- USERIDSIZE, massimo numero di caratteri per lo user ID
- CONVERSATIONIDSIZE, massimo numero di caratteri per il conversation ID
- TOKENSIZE, massimo numero di caratteri per ogni token
- WATERMARKSIZE, massimo numero di caratteri per ogni watermark
- MAXREADFROMSERVICEATTEMPTS, massimo numero di tentativi di lettura di un file

Inoltre questa libreria fornisce la definizione della struttura che descrive il servizio da avviare ServiceDescriptor:

```
typedef struct ServiceDescriptor
{
    char folderName[MAXNAMEFILE]; //name of folder (and service)
    pid_t pid; //pid of creating service
    char absolutepath[500]; //absolutepath of executible file
    pthread_t tid; //thread id
    char collateralConversationID[CONVERSATIONIDSIZE]; //id of collateral
conversation
    char collateralConversationToken[TOKENSIZE];
    char collateralConversationWatermark[WATERMARKSIZE];
    char userID[USERIDSIZE];
} ServiceDescriptor;
```

- Foldername identifica il nome del servizio e quindi anche della cartella in cui è installato l'eseguibile
- Pid è il Process ID assegnato nel momento in cui viene fatta la fork() per avviare il servizio
- Absolutepath è il percorso assoluto dell'eseguibile
- Tid è il thread id con il quale viene avviato il servizio
- collateralConversationID è l'ID della conversazione collateale associata al servizio
- collateralConversationWatermark è il watermark della conversione collaterale associata al servizio
- collateralConversationToken è il token di accesso alla conversazione collaterale associata al servizio
- userID è l'user id a cui associare il servizio

Fontautil espone svariate funzioni che possiedono compiti anche molto differenti:

- **listfolder(char list[MAXSUBFOLDERS][MAXNAMEFILE])**, ritorna il numero di sottocartelle e le inserisce in list
- **usage(const char *arg0)**, funzione di notifica di errori

- **controlProgramInFolder(char list[MAXSUBFOLDERS][MAXNAMEFILE], int n_folders)**, controlla l'eseguibile in ogni cartella della radice, ritorna 0 se non ci sono problemi
- **insertControlNameService(char list[MAXSUBFOLDERS][MAXNAMEFILE], char *nameservice, ServiceDescriptor *service)**, serve per inizializzare il servizio "service"
- **allocateMemoryServices(ServiceDescriptor **serviceArray, int lengthServiceArray, int size, char *absolpth)**, alloca la memoria necessaria per un numero lengthServiceArray di servizi
- **deallocateMemoryServices(ServiceDescriptor **serviceArray)**, rimuove la memoria allocata dall'array serviceArray
- **getEmptyService(ServiceDescriptor **serviceArray, int length)**, ritorna l'indice dell'array dei servizi che non è occupato da un servizio avviato
- **mocaWriteinService(struct ServiceDescriptor *service, const char *buffer)**, funzione di scrittura di un messaggio in un servizio, molto simile a quelli descritti in MoCA SDK
- **mocaReadfromService(struct ServiceDescriptor *service, char *buffer, int sizeBuffer)**, legge un messaggio dal servizio, molto simile a quelli descritti in MoCA SDK

10.3.2.1 allocateMemoryService

Questa particolare funzione alloca dinamicamente memoria all'array di servizi. È stata fatta questa scelta, invece che realizzare una allocazione statica, per una migliore scalabilità: in una visione di sviluppo futura il client deve poter allocare memoria proporzionalmente al numero di richieste che arrivano. Predisponendo il client già con una allocazione dinamica, non bisognerà riprogettare tutto il codice di gestione delle risorse, ma semplicemente applicare quello che già è presente.

```
//allocate dinamic memory
if (*serviceArray == NULL && lengthServiceArray == 0)
{
 *serviceArray = (ServiceDescriptor *) malloc(size *
sizeof(ServiceDescriptor));
    if (*serviceArray == NULL)
    {
        //can not allocate memory
        return -1;
    }
}
else if ((*serviceArray == NULL && lengthServiceArray != 0) ||
(*serviceArray != NULL && lengthServiceArray == 0))
{
    //not possible condition
    return -1;
}
else
{
    //allocats new blocks
    *serviceArray = (ServiceDescriptor *)realloc(*serviceArray, (size +
lengthServiceArray) * sizeof(ServiceDescriptor));
}
```

Si nota dalla precedente porzione di codice che all'interno di un selettore if – else if – else sono presenti 3 casi:

1. la memoria non è ancora stata allocata, quindi vado a fare una malloc per allocare la quantità di memoria desiderata
2. c'è un problema di lunghezza dell'array il quale risulta nullo, ma la lunghezza dello stesso risulta diversa da 0 o vice versa
3. è stata allocata già della memoria, quindi vado ad eseguire una realloc per adattare l'array alla richiesta

10.3.3 fontaMoCAProtClient

Questa libreria è stata creata per interpretare i messaggi del MOCA Protocol.

Possiede 6 costanti:

- **MOCACREATEDCOLLATERAL**, definisce il tipo "created" nella conversazione collaterale
- **MOCADESTROYCOLLATERAL**, definisce il tipo "destroy" nella conversazione collaterale
- **MOCANOTFOUNDTYPE**, definisce il tipo "non trovato"
- **MAXMOCAPARAMS**, numero massimo di parametri
- **MAXMOCAPARAMSLENGTH**, numero massimo di caratteri per ogni parametro
- **MAXTYPELENTH**, lunghezza massima del tipo

Definisce la struttura per descrivere il "messaggio":

```
struct MoCAMessage
{
    int type;
    char typeField[MAXTYPELENTH];
    char params[MAXMOCAPARAMS][MAXMOCAPARAMSLENGTH];
};
```

- type è il tipo di messaggio
- typeField rappresenta il tipo di messaggio che identifica la provenienza dello stesso
- params parametri del messaggio

10.3.4 fontahttp

Questa liberia è stata creata per inviare messaggi http sulla rete, una prima implementazione sfruttava le socket per l'invio di messaggi non crittografati, mentre successivamente si è deciso di sfruttare la libreria CURL[22] per implementare metodi più semplici ma basati su TSL.

Espone cinque costanti:

- **RCVBUFFER**, massima lunghezza del buffer di ricezione
- **SENDBUFFER**, massima lunghezza del biffer di invio
- **MAXHOSTSIZE**, massima lunghezza del campo host
- **WAITSECONDS**, secondi di attesa per un polling http
- **WAITCOLLATERALPOLLSECONDS**, secondi di attesa tra due richiesta di messaggi nella conversazione collaterale

Sono presenti 3 strutture, la più utilizzata è HttpsDescriptor siccome è impiegata per le connessioni crittografate

```
typedef struct MemoryStruct
{
```

[22] https://curl.haxx.se/

```
    char *memory;
    size_t size;
} MemoryStruct;
//Type that carry HTTP options
typedef struct HttpDescriptor
{
    struct hostent *server; //server info
    struct sockaddr_in serv_addr; //socket address
    int sockfd; //socket file descriptor
    char host[MAXHOSTSIZE]; //server url
    int port; //service port (80)
    pthread_t tid; //thread id
} HttpDescriptor;
typedef struct HttpsDescriptor
{
    CURL *curl; //curl
    char host[MAXHOSTSIZE]; //server url
    int port; //service port (80)
    pthread_t tid; //thread id

} HttpsDescriptor;
```

HttpsDescriptor possiede:

- curl, un puntatore che serve per definire un oggetto di tipo CURL, necessario per l'invio dei dati
- host, è l'URL del server a cui si vuole inviare i dati
- port, è la porta su cui collegarsi
- tid, thread per l'invio asincrono dei messaggi

Possiede delle funzioni per l'invio di un messaggio http non crittografato

- **openHTTPConnection(HttpDescriptor *http)**, apre una connessione non crittografata
- **HTTPsendANDrcv(char *msgtosend, char **msgrcv, HttpDescriptor *http)**, invia un messaggio http e inserisce la risposta in msgrcv
- **closeHTTPConnection(HttpDescriptor *http)**, chiude una connessione non crittografata
- **getHTTPHeader(char *message, char **header)**, copia l'header del messaggio
- **getHTTPBody(char *message, char **body)**, copia il corpo del messaggio

E funzioni per l'invio di un messaggio crittografato:

- **openHTTPSConnection(HttpsDescriptor *http, struct curl_slist *list)**, apre una connessione https crittografata
- **HTTPSsendANDrcv(char *sendbody, char **bodyrcv, HttpsDescriptor *http)**, manda un messaggio https e mette in bodyrcv la risposta
- **closeHTTPSConnection(HttpsDescriptor *http)**, chiude la connessione crittografata
- **WriteMemoryCallback(void *contents, size_t size, size_t nmemb, void *userp)**, usato per copiare la memoria da un oggetto CURL ad un buffer qualunque

10.3.4.1 HTTPSsendANDrcv

Quello che segue è l'algoritmo di base per inviare un messaggio crittografato. Si noti che l'utilizzo della libreria CURL semplifica molto il codice e snellisce anche gli algoritmi. Se l'apertura della connessione è andata a buon fine, si può inviare un messaggio chiamando HTTPSsendANDrcv:

```
int HTTPSsendANDrcv(char *sendbody, char **bodyrcv, HttpsDescriptor *http)
{
    if(http->curl == NULL)
    {
        //curl is null
        return -1;
    }
    CURLcode res;
    MemoryStruct mem;
    //allocate  byte (other allocated dinamically)
    mem.size = 0;
    mem.memory = (char *) malloc(1);
    /* we pass our memory struct to the callback function */
    if( (res = curl_easy_perform(http->curl)) != CURLE_OK)
    {
        //someting goes wrong
        fprintf(stderr, "curl_easy_perform() failed: \n");
        return -2;
    }
    return mem.size;
}
```

10.3.4.2 WriteMemoryCallback

Tale funzione serve per copiare la memoria dal buffer di ingresso adattandolo dinamicamente:

```
size_t WriteMemoryCallback(void *contents, size_t size, size_t nmemb, void
*userp)
{
  size_t realsize = size * nmemb;
  struct MemoryStruct *mem = (struct MemoryStruct *)userp;
  mem->memory = realloc(mem->memory, mem->size + realsize + 1);
  if(mem->memory == NULL)
  {
  printf("not enough memory (realloc returned NULL)\n");/* out of memory! */
    return 0;
  }
  memcpy(&(mem->memory[mem->size]), contents, realsize);
  mem->size += realsize;
  mem->memory[mem->size] = 0;
  return realsize;
}
```

10.3.5 Fontabotconnector

Questa è la libreria principale di tutto il Client MoCA. Permette lo scambio di messaggi tra client e BOT Core, inoltre implementa tutti i poller di sistema ed il metodo per avviare i vari servizi.

Oltre alle macro di debug non contiene nessuna costante (sono utilizzate quelle di fontautil) ma contiene la definizione per il puntatore a funzione che permette di richiamare la funzione di avvio dei servizi startNewService di mocamain:

```
/*
Used to connect a message from MontessoroBOT to mocamain
*/
typedef void (*CollateralHandler)(struct MoCAMessage message);
```

Inoltre definisce la struttura necessaria per la connessione con il bot:

```
typedef struct BotConnectionParams
{
        char conversationId[CONVERSATIONIDSIZE]; //conversationID to excange
information with bot
        char token[TOKENSIZE]; //token used to connect to bot
        char userID[USERIDSIZE]; //id of this terminal
        pthread_t pollerTID; //thread id of poller
        pthread_t pollerCollateralConversationID;//thread id of collateral
conversation poller
        char *pollWatermark[WATERMARKSIZE];
        CollateralHandler handler; //used to create a connection between
BotPOLLER and mocamain

}BotConnectionParams;
```

Essa presenta:

- **conversationId**, ID della conversazione
- **token**, token di accesso alla conversazione
- **userID**, id a cui è associato il MoCA Client
- **pollerTID**, thread id del poller di conversazione
- **pollerCollateralConversationID**, thread id del poller della conversazione collaterale
- **pollWatermark**, watermark della conversazione collaterale
- **handler**, puntatore a funzione per richiamare la funzione di callback all'arrivo di un messaggio dal bot

Espone svariate funzioni:

- **openConversation(char *conversationId, char *token)**, apre una conversazione con il BOT core
- **startPollHttp(void * arg)**, è un poller http generic, può essere utilizzato in vari ambiti
- **sendMessagetoBOT(char *message, BotConnectionParams conn, char **result)**, manda un messaggio sulla conversazione con parametri "conn"
- **sendEndConversationtoBOT(BotConnectionParams conn, char **result)**, manda un messaggio di fine conversazione, per chiuderla e rilasciare le risorse
- **startPollBOT(void *arg)**, serve per eseguire il polling sulla conversazione principale del Client MoCA

- **startService(void *arg)**, è la funzione principale su cui tutto il sistema si basa, serve per avviare un servizio
- **clearService(ServiceDescriptor *service)**, cancella un servizio
- **startPollCollateralConversationBOT(void *arg)**, è la funzione poller delle conversazioni collaterali
- **writeInService(const char *textToWrite, struct ServiceDescriptor *service)**, funzione ausiliaria per scrivere dati dal BOT al servizio

Generalmente le funzioni di polling sono formate da un while a loop infinito con una funzione di delay e una serie di task che quindi vengono eseguiti ciclicamente. Anche se possono risultare abbastanza complesse guardando il codice per la prima volta, in genere non presentano una grande difficoltà concettuale, viene rimandata ai commenti sul codice la loro descrizione. Al contrario andremo ad analizzare la funzione startservice perché parte del suo codice viene eseguito dal processo padre ed un'altra dal processo figlio che si creano dopo la fork(), quindi potrebbe risultare ostica ad una prima occhiata.

10.3.5.1 StartService

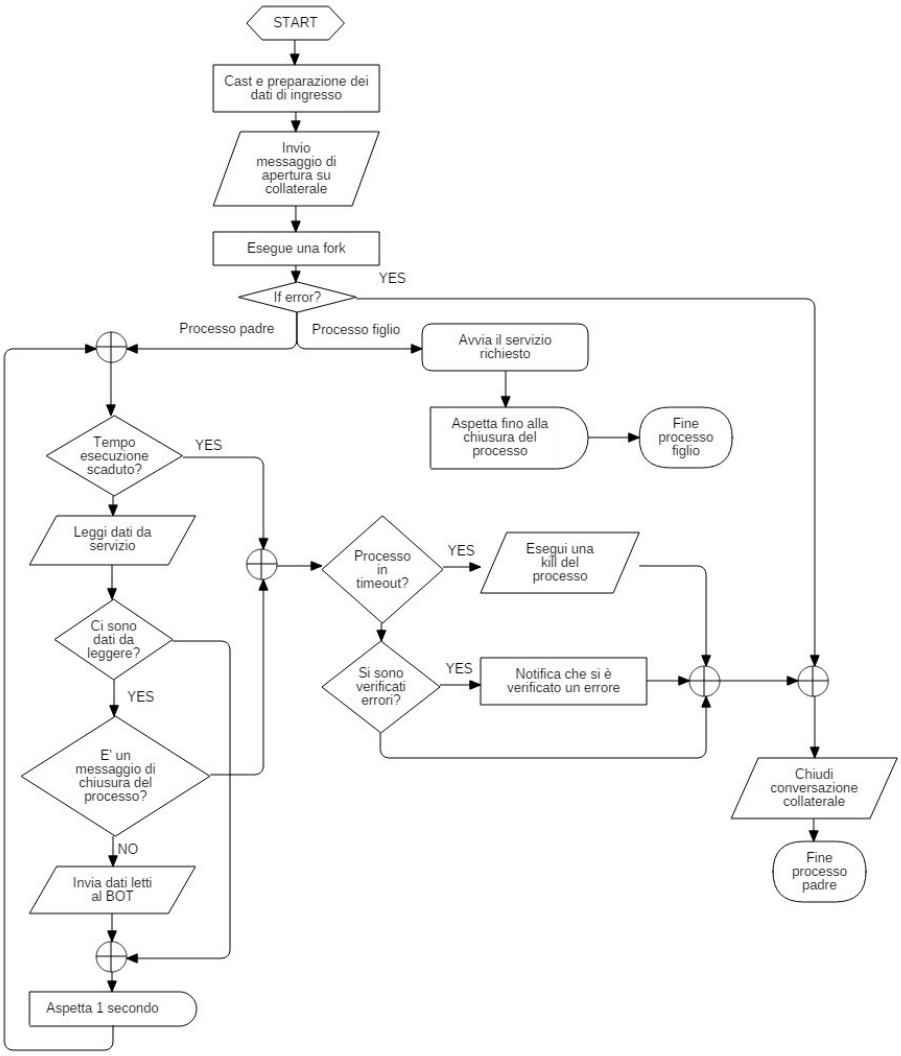


Figura 10.4 Flowchart di startService

Come si vede questo algoritmo ha la sua peculiarità nell'eseguire una fork() creando così due processi: il processo padre entra nel loop di lettura dei dati provenienti da servizio, mentre il figlio esegue il

servizio in sé. Si ricorda che tale funzione è scatenata quando un utente dal BOT fa richiesta di avvio di un servizio. Il processo figlio viene terminato quando il servizio completa la sua funzione oppure quando il processo padre, precedentemente entrato nel loop di lettura, supera il tempo di timeout. Il tempo di timeout è stato implementato grazie ad un contatore che conta fino a TIMEOUTSECONDS (definito in fontautil) come si vede dalla seguente porzione di codice:

```
int selectResult = 0, bytesRead = 0, numofread = 0;
char buffer[MAXBUFFERSIZE];
for (numofread = 0; numofread < TIMEOUTSECONDS; numofread++)
{
    int tot = mocaReadfromService(service, buffer, MAXBUFFERSIZE);
    if (tot > 0)
    {
        //if a close conversation message arrives from service
        if (strcmp(buffer, "/closeconversation") == 0)
        {
            //service close
            selectResult = 1;
            break;
        }
        else
        {
            //normal message arrived from service
            numofread = 0;
            char *result;
            char standardcollateral[] = "/collateralclient";
            char newbuf[tot + strlen(standardcollateral) + 1];
            sprintf(newbuf, "%s%s", standardcollateral, buffer);
            sendMessagetoBOT(newbuf, collateralConnectionParams, &result);
            free(result);
        }
    }
    usleep(1000000);
}
```

Inoltre se si verifica una condizione di timeout, il processo deve essere arrestato forzatamente per non incorrere in situazioni di deadlock, per fare ciò ci si affida alla funzione kill:

```
kill(service->pid, 1);
```

Per scrivere un messaggio sul buffer del servizio ci si deve affidare a mocaWriteinService di fontautil.

10.4 Sorgente BOT Core

Il BOT Core è un programma sviluppato in C# che si appoggia all'infrastruttura Microsoft Bot Framework. Serve a collegare i Social Network con il MoCA Client appena descritti ed a interpretare quello che un utente gli scrive via chat. Infatti il sistema è collegato attraverso delle REST API al servizio LUIS: tale servizio è descritto meglio nella sezione dedicata, ma sostanzialmente è un'intelligenza artificiale che fornisce un "significato" alle varie frasi immesse dall'utente.

Il codice è suddiviso in classi che fanno parte di sei categorie:

- App_Start, standard e generata alla creazione del progetto dall'IDE, di scarso interesse per capire le varie funzionalità del BOT Core
- Controllers, sezione contenente il MessagesController che è la classe che inoltra i messaggi ad un Dialog
- Dialogs, sono il vero e proprio cuore del BOT Core, questi Dialog sono delle classi che suddividono i messaggi per tipologia e significato, attuano diverse azione quando sono invocati e permettono un controllo di flusso dei messaggi
- Functions, rappresentano dei spezzoni di codice che attuano determinate funzioni
- MoCA, sono le classi che servono per il collegamento con i MoCA Client
- Sentence, è una classe in cui sono racchiuse tutte le risposte che il BOT fornisce.

La descrizione delle sezioni App_start e Controller sono rimandate alla documentazione ufficiale di Microsoft che risulta sufficientemente completa e chiara.

10.4.1 Dialogs

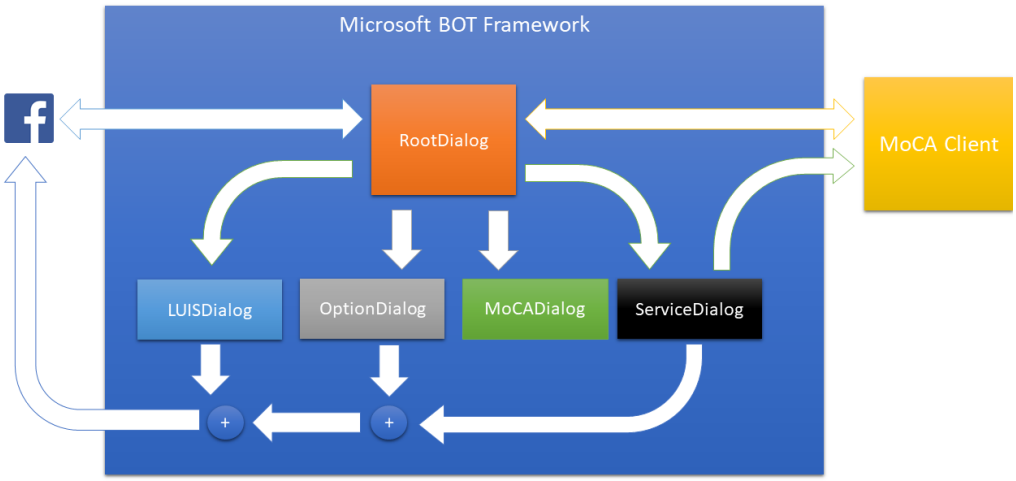


Figura 10.5 Flussi dei Dialog nel BOT Core

Come precedentemente affermato, ogni Dialog fornisce un significato diverso ai messaggi che accoglie

- **RootDialog**, è la classe che suddivide i messaggi tra gli altri Dialog, appunto ha una funzione di root
- **LUISDialog**, è la classe dei messaggi che arrivano direttamente dagli utenti e devono essere tradotti dall'intelligenza artificiale.
- **MoCADialog**, raccoglie tutti i messaggi che arrivano dai MoCA Client nelle conversazioni principali
- **OptionsDialog**, sono i messaggi di configurazione riservati al progettista
- **ServiceDialog**, è la classe che crea il collegamento con il MoCA Client ed inoltra tutti i messaggi dell'utente al MoCA Client una volta che esso ha avviato un servizio
- **UserData** è una classe di supporto che viene utilizzata solo come agglomerato di informazioni

10.4.1.1 LUISDialog

Quando arriva un messaggio destinato al LUISDialog, viene subito spedito tramite REST API al servizio LUIS. Di ritorno otterremo il messaggio originale più ulteriori informazioni come l'Intent a cui è più probabile sia associato la frase e le varie Entities se sono presenti. La classe LUISDialog è formata principalmente da "metodi di callback", ognuno di questi metodi corrisponde ad un Intent, ogni volta che arriva un messaggio associato ad un determinato Intent, viene scatenato il metodo corrispondente.

Un esempio è quando l'utente chiede la modifica dello user ID digitando "imposta <userid>", nel LUISDialog viene scatenato:

```
[LuisIntent("ModifyUserID")]
public async Task ModifyUserID(IDialogContext context,
IAwaitable<IMessageActivity> activity, LuisResult result)
        {
            var message = await activity;
            this.sendTyping(context);

            try
            {
                EntityRecommendation modi;

                if (result.TryFindEntity(this.NameEntity, out modi))
                {
                    this.UserID = modi.Entity;

                    //save user id on cloud
                    this.saveUserIDonCloud(message);

                    await
context.PostAsync(Sentence.RandomSentence.getModifyUserIDRandomSentence(this
.UserID));
                }
                else
                {
                    await
context.PostAsync(Sentence.RandomSentence.getNoneRandomSentence());
                }
                //var entities = new List< EntityRecommendation >(
result.Entities );

            }
            catch (Exception e)
            {
                Debug.WriteLine(e.Message);
            }
            context.Wait(this.MessageReceived);
        }
```

Questo metodo tenta prima di salvare lo UserID richiesto sul cloud poi in un campo di LUISDialog, in modo che quando lo stesso utente, dopo un po' di tempo, vuole riutilizzare i servizi associati a quell'userID, non serve che re-inserisca lo UserID, in quanto il sistema andrà a cercarlo all'interno di Azure.

Si nota che la risposta derivante dalla linea: `await context.PostAsync( Sentence.RandomSentence.getNoneRandomSentence() );` Sfrutta la classe RandomSentence. Tale classe come si vedrà in seguito racchiude tutte le risposte possibili del BOT.

10.4.1.2 ServiceDialog

Questo particolare dialog serve a raccogliere tutti i messaggi da/per una conversazione collaterale. Cioè una volta avviato il servizio, i messaggi dell'utente non verranno inoltrati più al LUISDialog, ma verranno inviati a questo Dialog. Il serviceDialog a sua volta li inoltrerà alla conversazione competente. Inoltre all'avvio di serviceDialog viene creato un task che ogni tot secondi esegue il polling della conversazione collaterale per ricevere i messaggi che vengono inviati dal MoCA Client:

```
public async Task doWork(IDialogContext context)
        {
            int i = 0;
            while (!this.stopWork && i < 20)
            {
                i++;
                try
                {
                    //wait 3 seconds
                    await Task.Delay(new TimeSpan(0, 0, 3));
                    List<ActivityMessage> messages = await
this.moca.SinglePOLLCollateralConversation();
                    //for each message arrived, send his contenent to user
                    foreach (ActivityMessage s in messages)
                    {
                        if( s.type.Equals("message"))
                        {
                            await context.PostAsync(s.text);
                        }
                        else if (s.type.Equals("endOfConversation"))
                        {
                            await
context.PostAsync(RandomSentence.getArrestedServiceRandomSentence());
                            i = 100;
                            this.stopWork = true;
                            //exit from the service
                        }
                    }
                }
                catch(Exception e)
                {
                    Debug.WriteLine("do work "+e.Message);
                }
            }
            this.stopWork = true;
        }
```

Anche in questo caso, come nel MoCA Client, si tratta di un while con temporizzazione, si nota che quando il tipo del messaggio equivale con `"endOfConversation"` esce dal ciclo e rilascia le risorse.

10.4.2 Functions

Le functions sono generalmente delle funzioni assestanti l'una dalle altre. Vengono invocate dai Dialogs e possono essere considerate come dei servizi intrinsechi del bot senza che venga collegato nessun MoCA Client. È stata sviluppata a livello dimostrativo una sola function che ha lo scopo di scaricare ed eseguire un parsing della pagina web dell'ateneo per controllare quando il professor Montessoro ha ricevimento studenti e quindi ritornarlo all'utente che ne ha fatto richiesta. Si riporta di seguito la funzione principale:

```
public static async Task<string> getReception()
        {
            string uscita = null;
            using (var httpClient = new HttpClient())
            {
                // Do the actual request and await the response
                var httpResponse = await
httpClient.GetAsync("http://web.uniud.it/moreinfo.htm?id=bae7d8aaeb09f53f091
29e4fe0e7882f");
                // If the response contains content we want to read it!
                if (httpResponse.Content != null)
                {
                    var responseContent = await
httpResponse.Content.ReadAsStringAsync();
                    responseContent = Regex.Replace(responseContent,
"<.*?>", String.Empty);
                    //get services and conversation id
                    responseContent = responseContent.Replace("Credits",
"");
                    responseContent = responseContent.Replace("\n", "");
                    responseContent = responseContent.Replace("\r", "");
                    int ix = responseContent.IndexOf("Orario di
ricevimento");
                    string code = null;
                    if (ix != -1)
                    {
                         code = responseContent.Substring(ix);
                        // do something here
                    }
                    Debug.WriteLine("Reception: " + code);
                    return code;
                    // From here on you could deserialize the
ResponseContent back again to a concrete C# type using Json.Net
                }
            }
            return null;
        }
```

10.4.3 MoCA

In questa sezione sono presenti gli algoritmi per connettere i MoCA Client con il BOT Core.

Presenta quattro classi:

- ActivityMessage
- From
- MoCAConnector
- MoCAProtocol

Le prime due sono di scarso interesse siccome hanno la sola utilità di agglomerare dei dati, MoCAProtocol è una semplice implementazione del Protocollo MOCA, si veda la sezione specifica.

10.4.3.1 MoCAConnector

Questa classe risulta essere molto articolata, infatti è il codice che implementa il collegamento con il MoCA Client. Tale classe è utilizzata per ricavare dalle DIRECT LINE API e dal database MySQL i messaggi e le informazioni associate all'userID impostato dall'utente, inoltre è utilizzato dal serviceDialog per inoltrare i messaggi dall'utente al servizio. Un semplice esempio delle sue potenzialità è il metodo `createCollateralConversation()` il quale crea la conversazione collaterale necessaria alla comunicazione con il servizio:

```
/// <summary>
/// Creates the collateral conversation
/// </summary>
/// <returns>token of the collateral conversation</returns>
public async Task<string> createCollateralConversation()
        {
            string token = null;

            // Wrap our JSON inside a StringContent which then can be used
by the HttpClient class
            var httpContent = new StringContent("", Encoding.UTF8,
"application/json");

            using (var httpClient = new HttpClient())
            {
                httpClient.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", this.BotSecret);
                // Do the actual request and await the response
                var httpResponse = await
httpClient.PostAsync("https://directline.botframework.com/v3/directline/conv
ersations/", httpContent);

                // If the response contains content we want to read it!
                if (httpResponse.Content != null)
                {
                    var responseContent = await
httpResponse.Content.ReadAsStringAsync();
                    //parse
                    JObject obj = JObject.Parse(responseContent);
                    //get token
                    token = (string)obj["token"];
                    //get conversation id
```

```
                    string collateralid = (string)obj["conversationId"];
                    if (token != null)
                        this.CollateralConversationToken = token;
                    if (collateralid != null)
                        this.CollateralConvesationID = collateralid;
                }
            }
            return token;
        }
```

Questo algoritmo come si può intuire dal sorgente esegue prima una richiesta http verso l'endpoint di creazione delle conversazioni, in seguito esegue un parsing del messaggio di ritorno per ottenere il token di collegamento.

10.4.4 Sentence

È presente solamente una classe: RandomSentece. Questa classe contiene tutte le possibili risposte del bot ad un intent specifico e tramite una funzione randomica, randomizza il risultato d'uscita. In questo modo rende più reale l'esperienza d'uso del BOT. Per esempio se l'utente chiede "come va?!" il LUISDialog lancerà il metodo `getFeelRandomSentence` necessario per rispondere all'utente

```
/// <summary>
/// getGreetingRandomSentenc
/// get a sentence random to sey HI!
/// </summary>
public static string getFeelRandomSentence()
{
      string[] sentences = new string[]{  "Molto bene grazie!", "Non va
male, ma andrebbe meglio se il topo che mi sta mangiando il cavo
dell'alimentazione la smettesse!", "Ho uno serpente nello stivale!",
"Veramente bene!", "Mai stato meglio!", "Oh beh sai, un po' tetro un po'
buio che ci vuoi fare?!", "Bene, di sicuro meglio di Cortana, povera
ragazza, la insultano tutti!"};
       Random rnd = new Random();
       int n_sentence = rnd.Next(sentences.Length);
       return sentences[n_sentence];
}
```

10.5 MySQL REST API

Per limitare l'accesso al database MySQL e sfruttare sempre lo stesso tipo di comunicazione tra un servizio e l'altro si è deciso di creare delle semplici REST API da applicare al server che contiene il database. Così sia il BOT Core che il Client MoCA non necessitano di ulteriori librerie per ottenere i dati dal database che non siano quelle che già utilizzano per comunicare tra loro. Tali api sono molto semplici e sono sviluppate in PHP. Per quanto riguarda il loro funzionamento rimando alla sezione dedicata.

Il seguente spezzone di codice racchiude la selezione delle informazioni relative alla richiesta GET http fatta dal BOT Core dalle REST API verso il database MySQL:

```
if($_GET["Function"] === "get")
      {
```

```
            $query_read='SELECT * FROM MoCAConnector WHERE
UserID="'.$_GET["UserID"].'";';
            $result=mysql_query($query_read,$myconn);
            if(!$result )
            {
                  //font found results
                  $error->ErrorID = 4;
                  $error->Error = "UserID non trovato nel database, MoCA
non attivato";
                  echo json_encode($error);
                  mysql_close();
                  return;
            }
            //found something in DB, lets fatch
            $row=mysql_fetch_assoc($result);//take first
            if($row === NULL or $row["UserID"] === NULL)
            {
                  //font found results
                  $error->ErrorID = 5;
                  $error->Error = "Errore nella query del DB";
                  echo json_encode($error);
                  mysql_close();
                  return;
            }
            //take informations about Conversation and services and crete
JSON
            $json->UserID = $row["UserID"];
            $json->MoCAConversationID = $row["MoCAConversationID"];
            $json->Services = explode(",", $row["Services"]);
            echo json_encode($json);
            mysql_close();
            return;

      }
```

11 CONCLUSIONI

Abbiamo dimostrato con questa tesi che i microservizi possono risultare davvero convenienti nello sviluppo architetturale di progetti software complessi. La peculiarità di poter scalare con facilità, sia in numero di servizi che in connessioni, accelera di molto il processo di ingegnerizzazione e di rilascio del software rispetto ad una visione monolitica del sistema. La scalabilità del software è migliorata anche dal fatto che è inoltre possibile istanziare servizi con medesime caratteristiche su macchine differenti e luoghi diversi. Questo oltre a ridurre le latenze, se consideriamo un accesso globale ad un servizio, permette una gestione automatica dei flussi di dati che quindi vengono smistati in più punti senza andare ad appesantire il sistema generale. La suddivisione di uno stesso servizio su più macchine permette anche di migliorare la manutenibilità del software stesso da parte degli operatori o dei possessori delle singole istanze. L'adattabilità del sistema progettato permette inoltre di sfruttare algoritmi differenti, sviluppati in linguaggi anche molto diversi tra loro; tale caratteristica permette all'apparato di essere molto appetibile a qualunque tipo di programmatore che può sfruttare le proprie conoscenze in termini di sviluppo per estendere il sistema senza dover imparare un nuovo linguaggio da zero. Un sistema molto simile è stato adottato da BitCoin[23] con il processo di mining. In poche parole il mining permette la condivisione delle proprie risorse per eseguire determinati algoritmi. La filosofia del sistema progettato non differisce di molto: i servizi istanziati su un determinato client (non necessariamente quello del programmatore dei servizi stessi) potrebbero risultare una fonte di guadagno per il fornitore di servizio, invogliando in questo modo persone differenti ad istanziare algoritmi non propri. Come è successo a BitCoin l'intero progetto potrebbe essere impiegato non solo come sistema di sviluppo ma anche come modello economico la cui fonte di guadagno quindi risulterebbero le aziende e i programmatori che necessitano di istanziare un determinato servizio pagando una quota indirettamente a chi istanzia il servizio stesso. L'esperienza di questa tesi mi ha permesso di apprendere anche molte nozioni sulle tecnologie Microsoft, in particolare riguardo allo sviluppo dei BOT: un mercato dove medie e grandi aziende, allo stato attuale, stanno investendo molto.

12 SVILUPPI FUTURI

Le integrazioni del sistema possono essere molteplici proprio grazie alla sua notevole versatilità:

- Come preannunciato all'interno della tesi il sistema MoCA è predisposto per dinamicizzare il numero di servizi eseguibili dal client stesso, però solo in fase di compilazione modificando il numero massimo di processi erogabili simultaneamente. L'implementazione successiva dovrebbe rendere dinamica questa meccanica rispetto alle richieste in ingresso al client.
- Estendere le MoCA SDK ad ulteriori linguaggi. Grazie a questa implementazione risulterebbe più semplice integrare linguaggi come JAVA, C#, Node.JS, C++, Pyton, PHP, Matlab, Ruby o Visual Basic. Si ricorda che il sistema è già predisposto per accoglierli, basta solamente modificare leggermente la procedura di avvio del servizio in relazione al tipo di linguaggio scelto.
- Implementare il MoCA Client in modo che sia avviabile su sistemi operativi differenti da UNIX.
- Migliorare latenze e sistemi di propagazione dei messaggi su bot.
- Creazione di un Framework opensource per applicare il modello di sviluppo a qualunque progetto software.
- Incrementare Function ed Intent del BOT Core per aumentarne le caratteristiche e funzionalità.
- Realizzare uno studio economico sulle possibilità di guadagno in funzione dell'allocazione di memoria e utilizzo delle risorse personali al fine di pubblicare le funzionalità del servizio su più client contemporaneamente in modo trasparente e sicuro. Un modello simile è stato applicato da BitCoin in relazione al mining.

[23] https://www.bitcoin.com/

BIBLIOGRAFIA

[1] A. S. Tanenbaum. Reti di computer, Prentice Hall, Utet Libreria, 2010

[2] J. F. Kurose, K. W. Ross, Reti di calcolatori e internet, Un approccio top-down, Person, 2013

[3] K. Brandl, R. Standefer, Key concepts in Direct Line API 3.0, https://docs.microsoft.com/en-us/bot-framework/rest-api/bot-framework-rest-direct-line-3-0-concepts, 2017

[4] K. Brandl, R. Standefer, K. Iqbal, J. Dempsey, D. Kroker, Bot Framework, docs.microsoft.com, 2017

[5] J. Lewis, M. Fowler, Microservices, https://martinfowler.com/articles/microservices.html, 2014

[6] S. Newman, Building microservices, O'Reilly, 2015

[7] R. Fielding,UC Irvine, J. Gettys, Compaq/W3C, J. Gettys, Compaq, H. Frystyk, W3C/MIT, L. Masinter, Xerox, P. Leach, Microsoft, Berners-Lee, Hypertext Transfer Protocol -- HTTP/1.1, https://www.ietf.org/rfc/rfc2616.txt, 1999

[8] M. Jones, Microsoft, D. Hardt, The OAuth 2.0 Authorization Framework: Bearer Token Usage, https://tools.ietf.org/html/rfc6750, 2012

[9] T. Dierks, The Transport Layer Security (TLS) Protocol Version 1.2, https://tools.ietf.org/html/rfc5246, 2008