Sperimentazione del framework TuSoW in ambiente Android

Francesca Tonetti
francesca.tonetti@studio.unibo.it

Ylenia Battistini
ylenia.battistini@studio.unibo.it

February 2020

Introduzione

Lo scopo del progetto consiste nell'utilizzo di un moderno middleware distribuito basato sul modello di coordinazione **LINDA**, all'interno di un ambiente Android. La volontà è quella di usufruire di un Tuple Space condiviso fornendo, così, inter-operabilità tra due o più dispositivi. In particolare l'obiettivo è quello di armonizzare le API di TuSoW con le API di Android nonché rendere il più naturale possibile l'impiego dello stesso framework per un programmatore Android.
L'accesso allo spazio di Tuple, sia in lettura sia in scrittura, avviene mettendo a disposizione un insieme di primitive tra le quali poter scegliere sulla base dell'azione che si vuole intraprendere.
Come ulteriore funzionalitá del sistema verrà studiato e sperimentato lo standard Wi-Fi Direct, il quale rappresenta una nuova metodologia di comunicazione tra dispositivi dotati di tecnologia Wi-fi.

Contents

Chapter 1

Obiettivi e Requisiti

In questo capitolo verranno descritti gli obiettivi del sistema ed i requisiti evidenziati per la sua realizzazione.

1.1 Obiettivi

Il sistema si basa principalmente sul funzionamento del framework TuSoW in ambiente Android. Per fa sì che la progettazione del sistema richiesto soddisfi tutti gli obiettivi, è necessario che si passi da una fase preliminare durante la quale vengono definiti i requisiti di sistema. Tali requisiti risponderanno al concetto di "cosa", cioè lo scopo del progetto, senza definire "come" verrà effettivamente implementato.
Al termine dello sviluppo di tale progetto ci si aspetta che per un programmatore Android possa essere semplice ed utile usufruire di un Tuple Space condiviso per l'interazione tra due o più dispositivi.
Si vuole testare inoltre la nuova tecnologia Wi-Fi Direct; per raggiungere tale obiettivo è necessario effettuare numerose ricerche affinchè si possa trovare una libreria che semplifichi il protocollo applicativo. Per questo motivo è molto importante concentrarsi anche su aspetti meno funzionali riguardanti la tecnologia, evidenziandone eventuali punti di debolezza.

1.2 Requisiti

1.2.1 Requisiti funzionali

I requisiti funzionali del progetto, ovvero quelli che specificano le iterazioni tra il sistema e l'ambiente, sono:

- Implementazione di un sistema dinamico a cui possono connettersi più activty,

- Implementazione di una comunicazione tra Activity/Service e Tuple Space tramite chiamate Http,
- Armonizzazione delle API TuSoW con i costrutti forniti da Android.

1.2.2 Requisiti non funzionali

I requisiti non funzionali, ma comunque necessari al raggiungimento degli obiettivi sono:

- Studio del middleware TuSoW,
- Studio di una libreria Android che permetta di effettuare chiamate Http,
- Ricerca e studio di una tecnologia che permetta di usufruire dello standard Wi-fi Direct per la comunicazione.

1.3 Piano di lavoro

1. Approfondimento del linguaggio Kotlin per la comprensione del codice `Linda-text-client` e relative dipendenze,
2. Deploy del framework TuSoW in ambiente Android con effettiva verifica di funzionamento,
3. Analisi e approfondimento delle primitive TuSoW,
4. Armonizzazione API di TuSoW con API di Android,
5. Implementazione di Intent e Service, all'interno dell'applicazione Android, che permettano di mascherare le operazioni effettuate sul Tuple Space condiviso,
6. Studio approfondito di Wi-Fi Direct e la sua possibile applicazione in ambiente Android,
7. Ricerca di una tecnologia che permetta di usufruire dello standard Wi-Fi Direct per la comunicazione inter-dispositivo su uno Tuple Space condiviso.

Chapter 2

Analisi dei requisiti

L'analisi dei requisiti rappresenta un'attività preliminare per lo sviluppo di un sistema distribuito, il cui scopo è quello di definire le funzionalità e i requisiti che devono essere soddisfatti. Questi ultimi descrivono ciò che ci si aspetta di ottenere dal sistema al termine dello sviluppo, descritti in modo chiaro, completo e univoco. La fase di analisi dei requisiti è stata fondamentale, in quanto ha permesso di studiare a fondo le tecnologie da utilizzare per implementare al meglio il nostro progetto. Inizialmente ci siamo imbattute nello studio del progetto TuSoW pre-esistente, principalmente per quanto riguarda le classi da noi wrappate. In seconda battuta ci siamo concentrate sulla scelta di una buona libreria Android che ci consentisse di effettuare chiamate Http verso il Service. Durante questa fase non ci siamo imbattute nella scelta di un framework, come magari può accadere nella maggior parte dei progetti, in quanto il nostro obiettivo è proprio quello di trasportare un framework già esistente su una piattaforma completamente differente permettendone così l'utilizzo da mobile fornendo all'utente un'interfaccia semplice ed intuitiva che nasconda tutta la complessità sottostante.
Una volta scelta la libreria ci siamo chieste come potesse essere strutturato il progetto in termini di comunicazione tra i componenti. Ovviamente per rispondere a questa domanda abbiamo predefinito due componenti principali ovvero Activity e Service, la cui comunicazione sfrutta le API di Android (descritte in modo approfondito nel Capitolo 5). Una scelta importante si è rivelata essere, come sopra descritto, la scelta di una libreria per le chiamate Http verso il Service la quale è ricaduta sulla libreria Volley. [1]

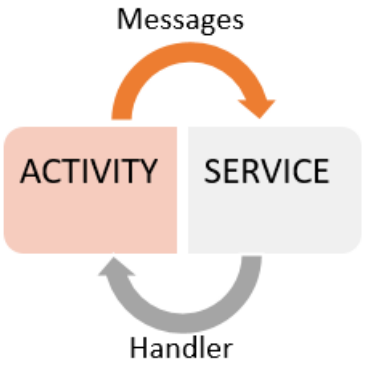


Figure 2.1: Esempio di comunicazione tra Activity e Service.

2.1 Risultati Attesi

TuSoW come descritto precedentemente è un middleware che permette agli utenti di usufruire di un Tuple Space condiviso su cui poter effettuare varie tipologie di azioni definite da primitive.
L'applicazione deve consentire agli agenti di mettere mano al Tuple Space condiviso, affinchè possano essere effettuate tutte le operazioni specificate dalle primitive del sistema, tramite l'utilizzo di una semplice interfaccia che nasconda l'intera implementazione sottostante e che permetta di non definire una "locazione" fissa al Tuple Space.
Nel nostro caso sarà presente un unico tipo di attore nonché l'agente che utilizzerà direttamente l'applicazione Android.

- **Agente:** colui incaricato ad interagire con l'applicazione Android, usufruendo delle primitive (*read*, *write*, *take*) messegli a disposizione per modificare il contenuto all'interno Tuple Space condiviso. L'agente rappresenta quindi un' entità attiva che può operare sul Tuple Space. Nel caso di un sistema distribuito verranno introdotti più agenti che, in maniera distribuita, possono interagirvi.

In figura vengono rappresentati i casi d'uso, ovvero le azioni che l'utente può richiamare sul Tuple Space. Nel nostro progetto verranno definite tre azioni che, volendo possono anche essere estese in futuro.

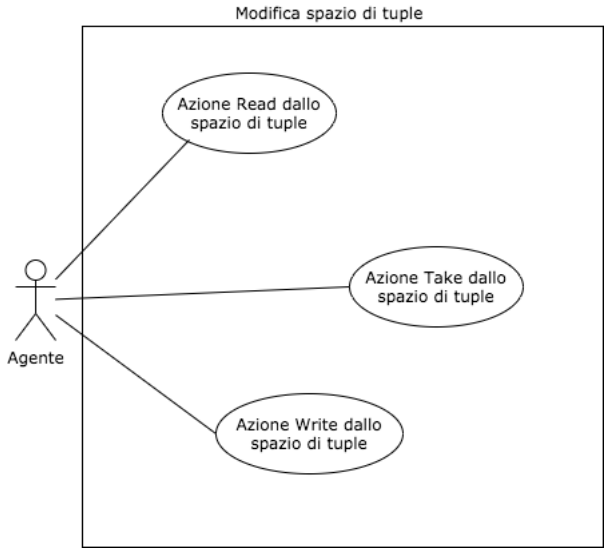


Figure 2.2: Diagramma dei casi d'uso per l'iterazione con il Tuple Space condiviso in ambiente Android.

Chapter 3

Stato dell'arte

Alla luce degli aspetti sopra riportati, prima di introdurre concetti tecnici legati alla realizzazione di tale progetto, è necessario fare leva su alcuni **aspetti teorici** alla base del problema in quanto l'intero sviluppo si basa sul modello di distribuzione **LINDA** sul quale si appoggia il middleware distribuito TuSoW.

3.1 Tuple-based coordination: modello LINDA

Il modello di coordinazione **LINDA** nasce nel mondo del calcolo parallelo, a metà degli anni Ottanta, per fornire agli sviluppatori di applicazioni distribuite la possibilità di gestire la comunicazione fra più processi tramite l'uso di un vero e proprio linguaggio distribuito.
Il punto di forza di **LINDA** consiste nel proporre un modello di coordinazione completamente distribuito sia a livello spaziale sia temporale, permettendo una estrema mobilità di processo.

L'espressività e la semplicità del modello **LINDA** lo rendono il modello di coordinazione di maggior successo oggi, in quanto permette di affrontare problemi molto complessi in modo semplice e chiaro. [4]
Tale modello si fonda sulla presenza di un ambiente computazionale e astratto che prende il nome di **Tuple Space**. Il Tuple Space aderisce al paradigma "associative shared memories", secondo il quale i dati presenti in memoria condivisa devono essere letti, specificando una parte del loro contenuto come chiave di ricerca.
I processi appartenenti al sistema distribuito comunicano attraverso strutture chiamate **Tuple** mediante invocazione di primitive come *write*, *read* o *take* sul Tuple Space.
Ciascuna implementazione del modello **LINDA** è in grado di definire il proprio *Tuple Language*. Per questo è possibile descrivere la comunicazione attraverso **LINDA** come la comunicazione tra più agenti distribuiti che attraverso delle operazioni di scrittura(writing), lettura(reading) e consumo, possono modificare

le Tuple presenti nel Tuple Space.

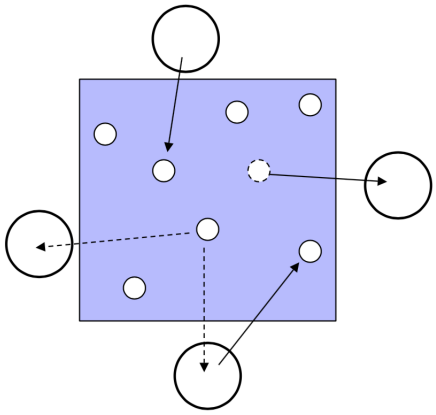


Figure 3.1: Meta-modello di un Tuple Space.

Generalmente le Tuple possiedono due proprietà fondamentali:

1. **eterogeneità**, gli elementi contenuti nel Tuple Space possono essere di tipo di dato diverso,

2. **ordine**, gli elementi appaiono in un ordine ben preciso.

Nel caso di **LINDA** le Tuple godono di una terza proprietà fondamentale che rappresenta il concetto di *esistenza indipendente* ovvero, ogni volta che le Tuple vengono create non rimangono legate all'esecuzione di alcun processo, questo poichè il modello **LINDA** è detto **generativo**.

3.1.1 Limiti del modello Linda

Poichè le caratteristiche di base del modello **LINDA** si adattano naturalmente al paradigma dell'Edge Computing (EC), il coordinamento basato su Tuple potrebbe essere adatto a governare l'iterazione tra dispositivi in applicazioni come CPS e WoT/IoT.
Gli aspetti di sincronizzazione tra azioni di agenti differenti ed il coordinamento delle loro stesse attività, nel caso di EC potrebbe tradursi nella gestione di flussi di lavoro distribuiti.
Il modello **LINDA** puó emulare modelli di interazioni come il passaggio di messaggi sincroni e asincroni, chiamate a procedure remote e altri aspetti utilizzati in applicazioni di EC e/o nel calcolo distribuito.
Tuttavia, recenti studi hanno dimostrato che il modello di coordinazione **LINDA** risulta strettamente legato ad aspetti come il formato di rappresentazione dei dati, il linguaggio di interrogazione o il protocollo di comunicazione. Per questi

e molti altri limiti è stato proposto un modello differente, sempre basato su **LINDA**, che permettesse però di superare questi vincoli nel tentativo di creare una tecnologia "off-the-shell" espressiva e flessibile, completamente adatta alle implementazioni di EC.

3.2 TuSoW

TuSoW è un moderno middleware distribuito basato su **LINDA** che ha lo scopo di fornire un'implementazione modulare, flessibile e altamente interoperabile per spazi di Tuple, particolarmente adatto nella coordinazione degli agenti software per EC.
Come per il modello **LINDA** sopra citato, un concetto che ritorna è il problema della coordinazione. Applicazioni basate sul cloud hanno lo stesso coordinatore centrale per dati, activities e devices. La domanda che sorge di fatto è come sia possibile mantenere coordinazione tra gli edge devices senza usufruire del Cloud.
Il modulo software di cui **TuSoW** si compone definisce un'astrazione del concetto di Tuplke Space, consentendo di interagirvi mettendo a disposizione una semplice API composta da primitive di comunicazione.
Il sistema fornisce un servizio asincrono in modo da garantire un'iterazione reattiva e mai bloccante tra gli agenti del sistema verso lo stesso Tuple Space condiviso. A causa della grande quantità di dati e di richieste da gestire, chiunque invochi delle primitive di comunicazione dovrà attendere una risposta che potrebbe risultare ritardata nel tempo. Come per il modello **LINDA**, anche nel caso del middleware **TuSoW**, la gestione delle richieste non segue un ordine particolare ma, di fatto l'obiettivo è quello di fornire un servizio di coordinazione il più possibile trasparente, che eseguirà le richieste da parte degli agenti in maniera casuale e senza alcuna sequenza temporale.

TuSoW consente, come rappresentato in figura 3.2, di avere un'implementazione locale che esula dalla piattaforma di utilizzo e dal protocollo di trasporto scelto dando la possibilità di serializzare tutte le chiamate verso uno spazio remoto.

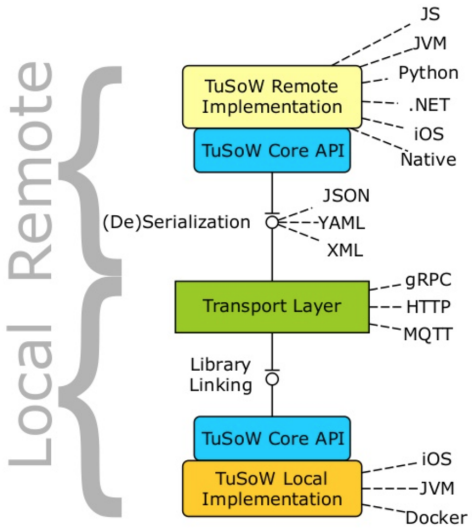


Figure 3.2: Prospettiva a livelli del modello TuSoW.

Questo rende il modello estremamente flessibile poiché si possono scegliere e/o utilizzare i protocolli e formati più idonei alla piattaforma su cui il middleware si va ad appoggiare.

Chapter 4

Tuple Space over the Web (TuSoW)

Nel capitolo corrente verranno descritte le primitive base sia per il modello **LINDA** sia per il modello **TuSoW**. Inoltre verrà introdotto il progetto di partenza da cui l'idea è partita.

4.1 Modello Linda - Primitive di base

Il modello **LINDA** mette a disposizione alcune operazioni base chiamate primitive di comunicazione, usate dai processi per interagire con e per mezzo di un Tuple Space.
Queste ultime possono essere viste come una sorta di API Tuple Space le cui caratteristiche permettono agli agenti di sincronizzare le loro attività. Le primitive di comunicazione di cui il modello usufruisce sono le seguenti:

- **in(*take*)** Consuma una Tupla all'interno del Tuple Space condiviso. Se quest'ultima non è immediatamente disponibile, la richiesta rimarrà sospesa fino al suo avvenuto inserimento;
- **rd(*read*)** Legge una Tupla all'interno del Tuple Space senza consumarla e, anche in questo caso, se la Tupla non è immediatamente disponibile, la richiestà rimarrà sospesa fino ad avvenuto inserimento;
- **out(*write*)** Inserisce una nuova Tupla all'interno del Tuple Space condiviso.

I template rappresentano lo strumento tramite il quale è possibile selezionare i dati presenti in un Tuple Space. Essendo il Tuple Space un meccanismo di memoria associativa, lo scopo dei template è quello di mantenere una conoscenza, parziale o totale, del dato che si vuole recuperare.

4.2 Modello TuSoW - Primitive di base

Precedentemente sono state descritte le primitive del modello **LINDA** attraverso le quali avviene la comunicazione. Nel modello **TuSoW** viene adottata una convenzione differente sui nomi delle primitive rispetto a quella comunemente utilizzata in letteratura. Nonostante la diversa convenzione sui nomi le primitive hanno la stessa modalità di utilizzo.
Il seguente mapping risulterà quello effettivamente utilizzato anche dal nostro sistema:

- **rd** →*read*
- **in** →*take*
- **out** →*write*

In termini generali le primitive accettano una o più rappresentazioni di Tuple come input e restituiscono in output zero o più rappresentazioni di Tuple a loro volta.

4.3 Rappresentazione delle primitive base

In questa breve sezione verranno rappresentate le primitive di base, sia per il modello **LINDA** sia per il modello **TuSoW**, sfruttando l'esempio della lavagna. Due utenti, che possono essere visti come agenti, Alice e Bob effettuano delle operazioni sul Tuple Space utilizzando le corrette API.
Vediamo l'esempio complessivo che comprende tutte le primitive, successivamente andremo nel dettaglio di ognuna di esse.

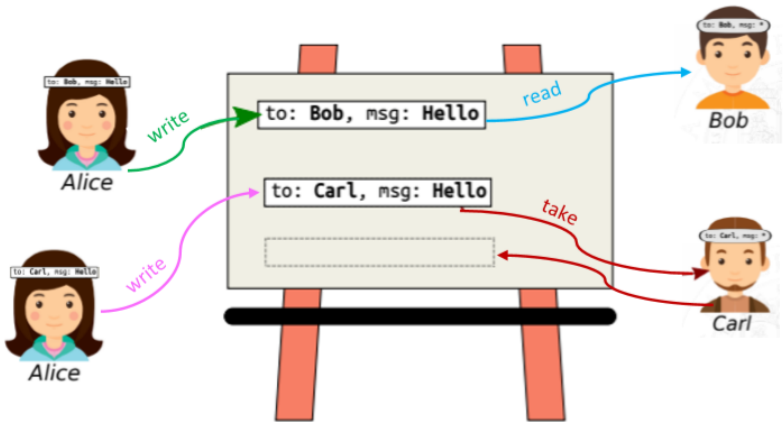


Figure 4.1: Lavagna per la spiegazione delle primitive *write*, *read* e *take*.

In questo esempio, sfruttando la metafora della lavagna, vediamo come i due agenti si scambiano informazioni usufruendo di uno spazio di indirizzamento condiviso. Scendiamo ora nel dettaglio di ogni primitiva utilizzabile.

4.3.1 Primitiva *WRITE*

La primitiva *Write* permette l'inserimento di una Tupla specificata all'interno Tuple Space.

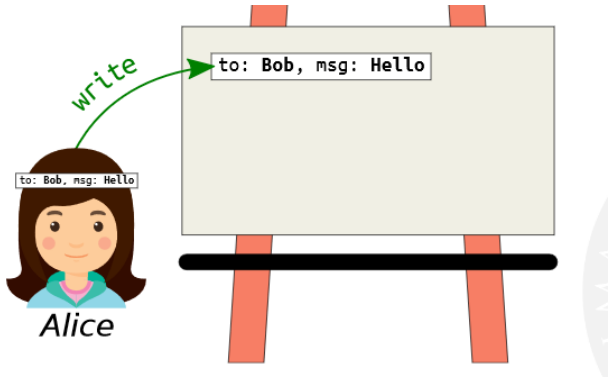


Figure 4.2: Lavagna per la spiegazione della primitiva *Write*.

4.3.2 Primitiva *READ*

La primitiva *Read* legge una Tupla all'interno del Tuple Space senza consumarla ma, nel caso in cui la Tupla non fosse immediatamente disponibile, la richiestà viene mantenuta in sospeso fino ad avvenuto inserimento.

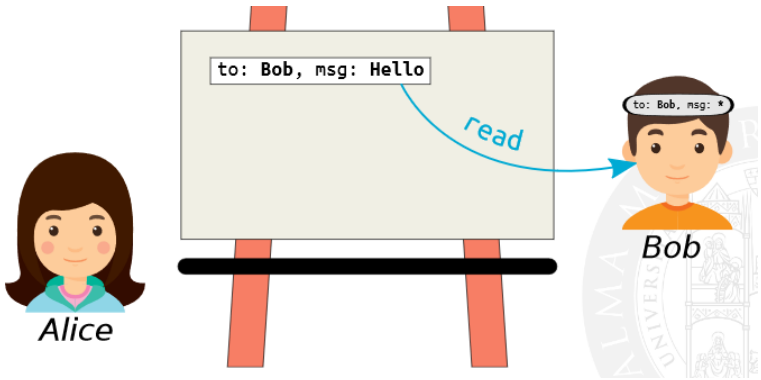


Figure 4.3: Lavagna per la spiegazione della primitiva *Read*.

Come sopra descritto questa primitiva legge la Tupla ma non la consuma infatti, nel caso in cui l'utente richieda una *Read*, il Tuple Space rimane inalterato come possiamo vedere dalla metafora della lavagna.

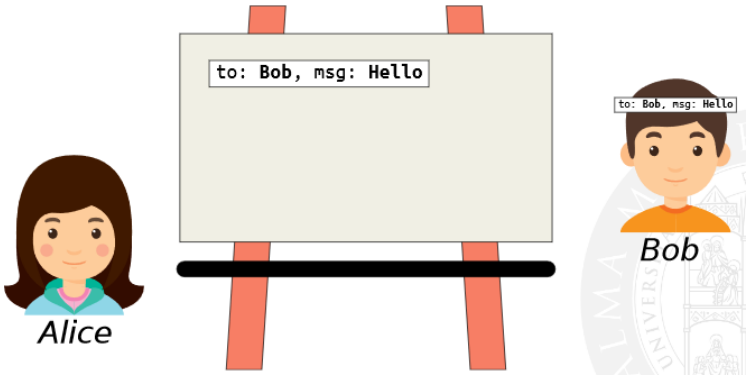


Figure 4.4: Lavagna per la spiegazione della primitiva *Read*.

4.3.3 Primitiva *TAKE*

La primitiva *Take* consuma una Tupla all'interno del Tuple Space condiviso. Se quest'ultima non è immediatamente disponibile, la richiesta rimarrà sospesa fino al suo avvenuto inserimento, allo stesso modo della *Read*.

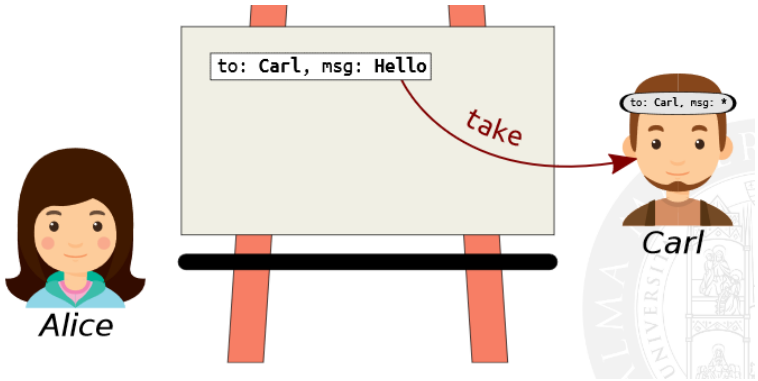


Figure 4.5: Lavagna per la spiegazione della primitiva *Take*.

La differenza sostanziale tra la *Read* e la *Take* ricade sul fatto che la primitiva *Read* non consuma la Tupla lasciando il Tuple Space inalterato mentre, la primitiva *Take* consuma la Tupla letta perciò modifica lo stato Tuple Space.

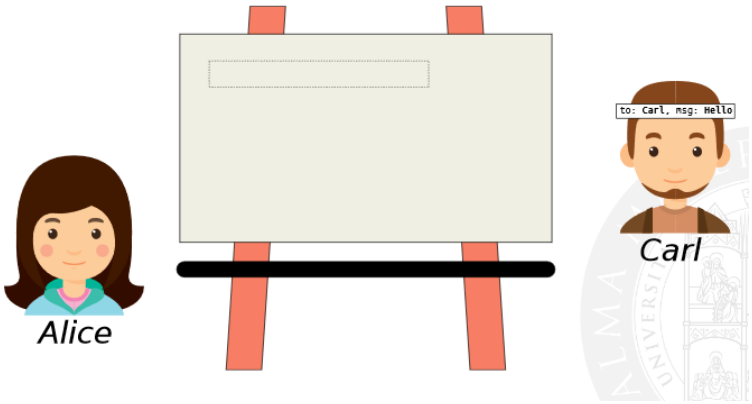


Figure 4.6: Lavagna per la spiegazione della primitiva *Take*.

4.4 Progetto di partenza

Questo progetto, proposto dal professor. Giovanni Ciatto, fa riferimento alla categoria `Tuple Based Coordination` e, si basa, principlamente sul funzionamento del framework **TuSoW** in ambiente Android.
L'idea del progetto consiste nel completare il deploy di **TuSoW** su Android cercando di armonizzare le API sfruttate dal Tuple Space con quelle fornite dalla piattaforma. La coordinazione con il progetto **TuSoW** è costituita da diversi moduli che si occupano della gestione e del coordinamento di applicazioni multi-processo, distribuite o multi-agente e multi-threaded tramite Tuple Space **LINDA**.
Il progetto è costruito su una gerarchia di moduli; in particolare troviamo:

- `Moduli Linda-*`,
- `Moduli Tusow-*`.

I moduli `Linda-*` consentono allo sviluppatore l'utilizzo di un Tuple Space a livello di codice in applicazioni simultanee, locali, ovvero non distribuite. La gerarchia dei moduli `Linda-*` si compone di sotto moduli, prevalentemente di tipo:

- `Linda-Text`,
- `Linda-Logic`.

I primi consentono un utilizzo del Tuple Space identificabile dal termine "text", ovvero testuale, che prevede le operazioni di inserimento e/o lettura di Tuple rappresentate come stringhe. I secondi prevedono un utilizzo del Tuple Space in cui le operazioni di letture e/o scrittura sono definite tramite l'uso di template. I moduli `Linda-logic-client` e `Linda-text-client`, equivalenti remoti di `Linda-Logic` e `Linda-Text`, permettono l'interazione con **TuSoW** sfruttando

le API messe a disposizione. E', inoltre possibile, interagire con **TuSoW** tramite qualsiasi client Http; in entrambe i casi è consentito l'utilizzo di Tuple Space remoti utilizzando la stessa API utilizzata per quelli locali.
Qui sotto una rappresentazione gerarchica del modello appena descritto, con particolare attenzione ai moduli appartenenti a `Linda-Text-*` poichè saranno quelli utilizzati per il raggiungimento degli obiettivi di progetto.

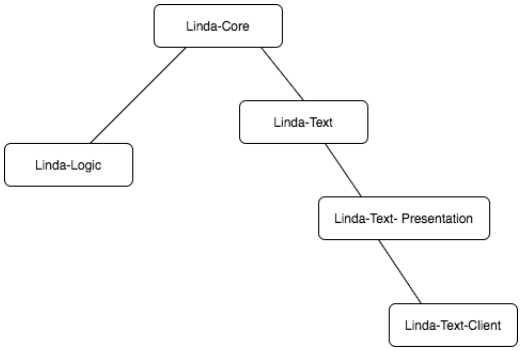


Figure 4.7: Gerarchia modello progettuale.

I moduli `Tusow-*` realizzano un wrapper Web per Tuple Space esposti come servizi Web, tramite un'API ReST-ful. **TuSoW** fa riferimento alle implementazioni di `Linda-core` sopra descritte.
Per poter interagire con il Tuple Space è stato definito un ulteriore modulo `Tusow-cli` che ne definisce un'interfaccia a riga di comando.
In seguito ad una attenta analisi degli aspetti sopra introdotti ed un approfondito studio dei meccanismi messi a disposizione dal sistema Android, nel capitolo successivo verranno presentati ed analizzati i componenti sfruttati per la realizzazione del sistema.

Chapter 5

Design

Nei capitoli precedenti è stata effettuata un'analisi delle funzionalità che il modello **TuSoW** dovrebbe offrire nonché un servizio applicato all'ambiente Android che permetta una coordinazione tra sistemi distribuiti, basato su Tuple.
In questa sezione verrano definiti e descritti i componenti fondamentali introdotti nella fase di implementazione dell'intero progetto ed infine rappresentate le modalità di iterazione tra questi.
In particolare verranno analizzati i seguenti elementi:

1. *Activity*,
2. *Service*.

5.1 Activity

Le Activity rappresentano il componente fondamentale di un'applicazione Android, anche in ambiente distribuito, la cui attività permette a ogni utente di definire la propria esperienza dinamica sull'app mobile.
Per questo ciascuna Activity funge da punto di ingresso per l'interazione con un'applicazione, ognuna avente la propria interfaccia utente. Generalmente, e nel nostro caso specifico, l'applicazione si compone di più Activity che possono interagire tra loro e avere dipendenze, a partire da una Main Activity. Ogni applicazione Android dev'essere accompagnata da un file chiamato *AndroidManifest.xml* nella sua cartella principale. Il Manifest raccoglie informazioni basilari sull'app tra cui le componenti dell'applicazione (attività, servizi, ecc.). Perciò, per poter usufruire di ciascuna Activity, è necessario che queste vengano inserite nel file Manifest del progetto, affinchè l'utente possa interagirvi.

5.1.1 Activity Lifecycle

All'interno del sistema, le Activity, sono gestire come stack di attività, di conseguenza quando viene avviata una nuova Activity questa verrà posizionata nella

parte superiore dello stack corrente per tutta la durata della sua esecuzione, lasciando l'Activity precedente al di sotto di essa, in modo che il suo riferimento venga posizionato sulla cima dello stack non appena la successiva terminerà.
Per questo motivo ad ogni Activity è possibile associare un ciclo di vita, in relazione agli stati che attraversa durante la sua esecuzione, usufruendo di un meccanismo di callback per gestire le transizioni tra questi.
Nello specifico è possibile individuare quattro stati essenziali:

- Active o Running,
- Visible,
- Stopped o Hidden,
- Destroyed.

Il diagramma seguente mostra il ciclo di vita di un'Activity in cui le callback sono raffigurate dai rettangoli. Gli stati principali in cui l'Activity può collocarsi sono rappresentati dagli ovali colorati.

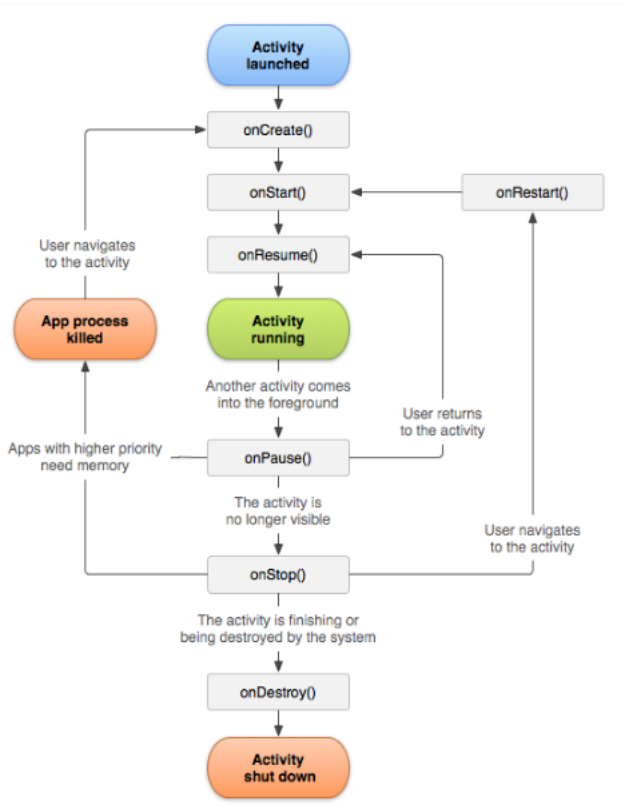


Figure 5.1: Activity lifecycle.

5.2 Service

Quando parliamo di Service invece, introduciamo il componente dedicato all' esecuzione di operazioni long-running in background. Il Service, a differenza dell'Activity, non fornisce un'interfaccia utente ed ha la caratteristica di poter continuare la sua esecuzione in background anche se l'applicazione che lo ha avviato viene interrotta.
Qualsiasi altro componente dell'applicazione può connettersi ad un Service per interagirvi ed eseguire IPC.
Ad esempio un Service permette di gestire transizioni di rete, eseguire operazioni di I/O su file, riprodurre musica o altro ancora. Esistono tre principali tipologie di Service:

1. *Foreground Service:* permette di eseguire operazioni evidenti all'utente in primo piano e può continuare ad eseguire anche se l'utente non interagisce con l'applicazione. Questa tipologia di Service deve poter visualizzare notifiche nel display.

2. *Background Service:* permette di eseguire operazioni non evidenti all'utente per questo motivo non ha la necessità di visualizzare notifiche .

3. *Bound Service:* questa tipologia di Service permette ad un Activiy di associarvi ad esso usufruendo della chiamata bindService(). Un servizio di questo genere offre un'interfaccia client-server attraverso cui permette ai componenti di interagirvi, inviando richieste o ricevendo risultati. Più componenti possono connettersi allo stesso Bound Service. La peculiarità di questa tipologia di Service è che nel momento in cui tutti i componenti ad esso connessi si disconnettono il Service viene distrutto, di conseguenza il suo tempo di vita risulta condizionato dai componenti collegati.

Oltre alle tipologie sopra descritte esiste un'ulteriore distinzione del componente Service in Started e Bound Service.
Quando un Service è di tipo Started il ciclo di vita risulta indipendente dal comportamento che lo ha avviato. Generalmente esegue un compito preciso e non restituisce nessun risultato.
Mentre un Service si dice Bound quando un altro componente si lega ad esso attraverso un Intent, usufruendo dell'interfaccia client-server per intergirvi.
In base a tali caratteristiche il design più appropriato per modellare il Service nel nostro caso ricade sullo Started Service. Nonostante all'interno del nostro progetto sia fondamentale restiure un risultato al componente che ha avviato il Service o, in generale a qualsiasi Activity che ne richieda uno, l'utilizzo di un Service di tipo Bound non rispettava le richieste. Abbiamo perciò definito un approccio ibrdio, consentito in ambiente Android, che ci permetta di utilizzare un Service Started restituendo però un risultato.
Una domanda lecita, a questo punto, potrebbe essere, "perchè non è stato scelto un Service Bound, vista la necessità di restuire un risultato?". La risposta a questa domanda ricade sul fatto che un Service di tipo Bound viene distrutto

nel momento in cui tutti i componenti ad esso connessi si disconnettono e, per il concetto di Service che accoglie un Tuple Space, questo non è l'approccio più congegnale.

5.2.1 Service Lifecycle

Analogamente ad un'Activity, anche il componente Service prevede un proprio ciclo di vita. Il sistema Android arresta un Service solo quando la memoria è in esaurimento e/o deve recuperare delle risorse di sistema per l'Activity che ha il focus ovvero, che è in cima allo stack di esecuzione. Di seguito viene fornito il diagramma del ciclo di vita di un Service distinguendo tra Started e Bound.

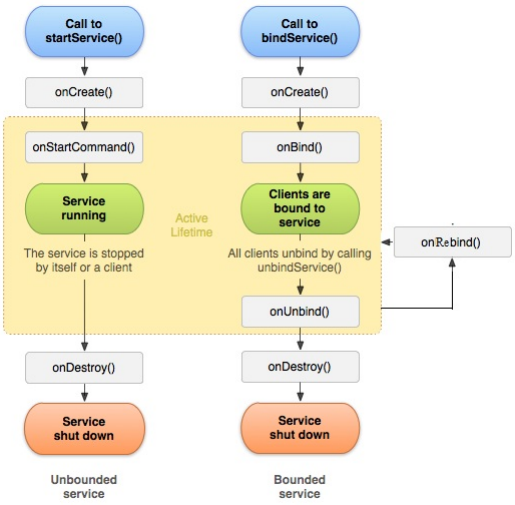


Figure 5.2: Service lifecycle.

Un aspetto fondamentale del componente Service riguarda la propria esecuzione nel Main Thread, ciò significa che un Service non crea un thread separato né viene eseguito in un processo a se stante.
Nel caso in cui il Service dovesse eseguire compiti bloccanti o di tipo long-running, per la motivazione sopra descritta, questi dovranno essere demandati ad un thread separato.
Nel caso specifico del nostro progetto oltre alla scelta del Service di tipo Started, sarà importarte modellare correttamente l'integrazione tra le funzionalità dello stesso e le primitive **TuSoW** in modo da consentire ai processi distribuiti un corretto utilizzo del Tuple Space.

5.3 Struttura progetto

TuSoW è un middleware distribuito in cui ogni componente principale, attraverso l'applicazione mobile, viene identificato come agente del sistema. Cias-

cun agente interagisce con l'applicazione tramite le interfacce messe a disposizione dalle Activity con cui effettuare delle richieste. Queste vengono comunicate direttamente al Service il quale a sua volta richiama il Tuple Space.
Per rendere possibile tutto ciò è stato utilizzato un meccanismo di Intent con cui l'Activity può sottomettersi al Service. Una volta sottomessa questa può comunicarci direttamente.
Di seguito in figura 5.3, viene raffigurata la comunicazione e l'interazione tra le varie componenti del sistema tramite un diagramma di sequenza [3]. L'entità protagonista nel diagramma è identificata dall'Agent che interagisce con l'Activity utilizzando le API messe a disposizione da **TuSoW**. Quest'ultima avrà il compito di rimpallare la richiesta verso il servizio a cui è collegata, il quale effettuarà la chiamata Http concreta verso lo spazio di Tuple.

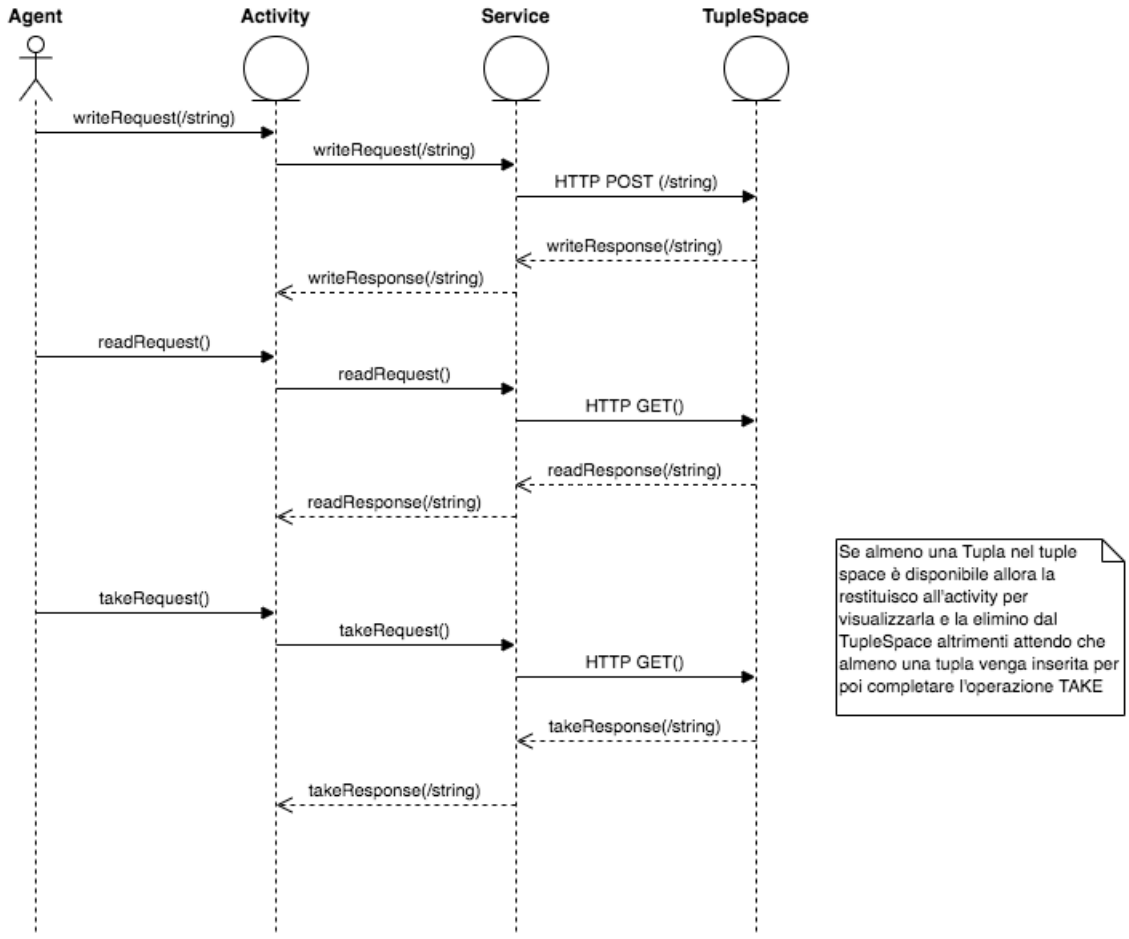


Figure 5.3: Diagramma di sequenza - interazione entità.

Chapter 6

Dettagli implementativi

In questo capitolo entreremo nel dettaglio degli strumenti utilizzati per la realizzazione dell'intero sistema. Inoltre, verranno presentate porzioni di codice rilevanti del progetto.

6.1 Strumenti utilizzati

Gli strumenti utilizzati per l'implementazione sono i seguenti:

1. *Android Studio* con rispettivo SDK, come piattaforma per lo sviluppo progettuale;
2. *Vert.x*, per la portabilità del servizio;
3. *Libreria Volley*, per la realizzazione di chiamate Http;

6.2 Android Studio e Android SDK

Android Studio è un ambiente di sviluppo integrato (IDE), lanciato di recente da Google per il sistema operativo Android. É disponibile gratuitamente ed è basato su IntelliJ.
Il cuore della programmazione Android risiede in un ambiente noto come Android SDK. Questo consente allo sviluppatore di realizzare applicazioni avendo a disposizione una serie di strumenti tra i quali un debugger, librerie e la possibilità di testarle su dispositivi emulati dal sistema. La versione dell'SDK si modifica a seguito di aggiornamenti, per questo motivo se si vuole prevedere la possibilità di creare applicazioni che funzionino su disposivi meno recenti bisogna scegliere un SDK che li copra. Android Studio supporta, inoltre, la tecnologia Wi-Fi Direct attraverso API ufficiali a partire dalla versione 4.0 (Jelly Bean), SDK 14.
Nello specifico, il progetto è stato implementato scegliendo come Android SDK

la versione 29; per garantire retrocompatibilità è stata definita anche una versione minima di SDK che supporta l'app, in questo caso la versione scelta è la 26.
Il testing dell'applicazione viene effettuato su emulatori di sistema; al momento della creazione di uno di essi Android permette di scegliere un image system nonché immagine di sistema. L'image system include l'accesso ai servizi di Google; noi abbiamo scelto come immagine di sistema *Oreo* (per la retrocompatibilità dell'SDK).

6.3 Vertx

Vertx è un framework ampiamente utilizzato per lo sviluppo di applicazioni reattive, event-driven, basate sulla JVM. E' estremamente modulare in quanto i vari moduli estendono il core con ulteriori funzioni di comunicazione web, supporto a differenti protocolli, tecnologie e linguaggi JVM (e.g Java, Kotlin, Scala,..).
Un'applicazione Vert.x consiste di uno o più componenti chiamati Verticle che comunicano tra di loro tramite un Event-Bus semplicemente attraverso lo scambio di messaggi. Vert.x permette di scalare molto facilmente infatti, a fronte di carichi di richieste crescenti utilizza tutti i thread a disposizione sulla base delle capacità della macchina su cui gira.
Durante la fase di analisi, ci siamo rese conto che il progetto fornitoci utilizzava il framework Vert.x. Non essendo a conoscenza del supporto che Android Studio mette a disposizione di quest'ultimo sono state effettuate alcune ricerche e sono state realizzate delle prove a riguardo. Per poter utilizzare Vert.x su Android è necessario importare la libreria su *build.gradle*. Le prove effettuate sono state suddivise in due step principali:

- Step 0, implementazione di un semplice server per verificare il corretto supporto di Vert.x da parte del sistema Android.
- Step 1, una volta verificato il supporto del framework abbiamo provato, tramite la creazione di un client, a contattare il server verificando così che non ci fossero determinati problemi di permission.

Una volta effettuate queste verifiche, ottenendo per entrambe esito positivo, ci siamo concentrate su aspetti di maggior interesse quali l'implementazione delle primitive.

6.4 Volley

L'accesso alla rete è un'attività molto comune per le applicazioni Android ed essendo una operazione condizionata da tempi di latenza variabile, è necessario che sia svolta su un thread secondario. Per questo motivo è stata predisposta una libreria apposita denominata Volley che si occupa dei seguenti aspetti:

- gestione autonoma delle richieste e connessioni multiple;

- *caching* delle risposte sia in memoria che su disco;
- classi per il supporto dei tipi più comuni di richieste;
- gestione delle priorità;
- strumenti di log, debug e tracciamento delle attività;

L'uso di tale libreria è reso possibile attraverso l'aggiunta della *permission* INTERNET all'interno dell'*AndroidManifest.xml* della nostra applicazione e della dipendenza alla libreria stessa dentro al file *build.gradle*.
Ci sono due concetti fondamentali che riguardano la libreria Volley: **Request** di tipo GET o POST e RequestQueue. Infatti, il protocollo Http si basa sulla iterazione tra client e server in termini di richiesta-risposta rispettivamente come i concetti sopra citati.
In pratica non è solo necessario effettuare direttamente una richiesta di accesso alla rete ma la stessa verrà inserita in una coda di richieste. Sarà poi la libreria Volley ad eseguirla appena possibile secondo le politiche e le condizioni del sistema definite.
Volley prevede alcuni tipi di richiesta per le forme più comuni di comunicazione ad esempio StringRequest che restituisce, all'interno di una stringa, il contenuto remoto richiesto o ancora ImageRequest, JsonArrayRequest, JsonObjectRequest. Un aspetto importante riguarda il fatto che per ogni Request saranno inclusi i riferimenti a due *listener* per l'invocazione, nel caso in cui la richiesta venga svolta con successo o viceversa in caso di errore. Non solo, sarà disponibile un oggetto VolleyError per mascherare eventuali errori relativi alla connessione di rete Http.

```
@Override
public CompletableFuture write(Tuple tuple) {
    CompletableFuture cp = new CompletableFuture();

    StringRequest request = new StringRequest(Request.Method.POST, this.myUrl,
            new com.android.volley.Response.Listener<String>() {
                @Override
                public void onResponse(String response) {
                    try {
                        Log.d(TAG, msg: "Risposta OK");
                        ResponseOut responseOut = new ResponseOut(response);
                        responseQueue.add(responseOut);
                        if (response.isEmpty()) {
                            cp.completeExceptionally(new IllegalStateException("Error"));
                        } else {
                            cp.complete(response);
                        }
                    } catch (Exception e) {
                        e.printStackTrace();
                        Log.d(TAG, msg: "problema");
                    }
                }
            }, new com.android.volley.Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            cp.completeExceptionally(new IllegalStateException("Remote Exception status code:" + error.getMessage()));
            Log.d(TAG, msg: "Errore!!!->" + error.networkResponse.statusCode);
        }
    }) {
        @Override
        public String getBodyContentType() {
            return "application/json; charset=utf-8";
            //return stringListTupleSerializer((StringTuple) tuple);
        }
```

Figure 6.1: Screen richiesta API *write*.

```
        @Override
        public byte[] getBody() {
            Log.d(TAG, msg: "tupla->" + tuple);
            try {
                Log.d(TAG, msg: "STRINGTUPLE->" + StringTuple.class + " mimetypes->" + MIMETypes.APPLICATION_JSON);
                Serializer<StringTuple> serializer = Presentation.INSTANCE.serializerOf(StringTuple.class, MIMETypes.APPLICATION_JSON);
                String tupleSerialized = serializer.toString((StringTuple) tuple);
                return tupleSerialized.getBytes( charsetName: "utf-8");
                //return stringListTupleSerializer((StringTuple) tuple).getBytes();
            } catch (Exception e) {
                e.printStackTrace();
                return null;
            }
        }

        @Override
        protected Map<String, String> getParams() {
            Map<String, String> params = new HashMap<>();
            params.put("bulk", "false");
            return params;
        }

        @Override
        public Map<String, String> getHeaders() {
            Map<String, String> params = new HashMap<>();
            params.put(HttpHeaders.CONTENT_TYPE.toString(), "application/json");
            params.put(HttpHeaders.ACCEPT.toString(), "application/json");
            return params;
        }
    };

    addInVolleyQueue(request);
    return cp;
}
```

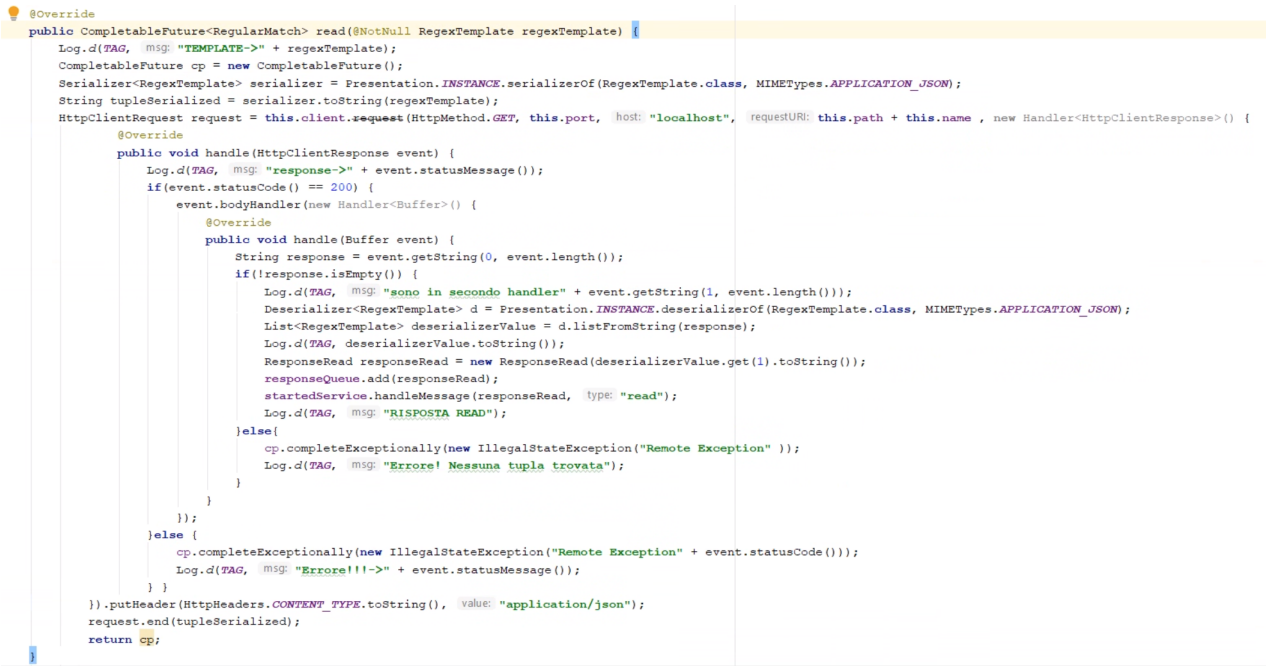
Figure 6.2: Screen richiesta API *write*.

La comunicazione tra client e server prevede che le Tuple in ingresso ed in risposta siano serializzate e deserializzate, per questo motivo all'interno della

nostra chiamata Http è stato necessario utilizzare due oggetti immutabili quali *Serializer* e *Deserializer* i quali contengono la logica per serializzare e deserializzare oggeti, nel nostro caso *StringTuple*. Per poter mandare una Tupla dal client verso il server è necessario prima serializzarla. Per serializzare abbiamo utilizzato il *Serializer* automatico il quale prende in ingresso due parametri:

- la classe degli elementi che vogliamo serializzare, nel nostro caso StringTuple.class,
- MIMEType che si vuole utilizzare per serializzare, nel nostro caso abbiamo scelto *APPLICATION_JSON*.

In questo modo abbiamo ottenuto un'instanza del serializzatore. Il *Serializer* ha due metodi per serializzare che sono rispettivamente *toString(StringTuple s)* e *toString(Collection<T> c)*. Il primo accetta uno StringTuple e lo converte in una stringa che è codificata in JSON. Il secondo metodo invece accetta una Collection di StringTuple e la converte in un Array<JSON>; per questo motivo è stato utilizzato il metodo *toString(StringTuple s)* nel caso in cui ci fosse la necessità di serializzare una sola Tupla in caso contrario verrà utilizzato il metodo *toString(Collection<T>)*.
Ovviamente quando il server restituisce un risultato è importante deserializzare. Per fare ciò viene utilizzato un *Deserializer* il quale ha un comportamento che è inverso a quello del *Serializer*, ovvero è un oggetto che data una stringa mi restituisce un'instanza di un determinato tipo. Nel nostro caso, volendo deserializzare una StringTuple andiamo a creare un *Deserializer<StringTuple>*. Come per il *Serializer* anche il *Deserializer* ha due metodi per la deserializzazione, in particolare *fromString(String s)* e *listFromString()*, il primo viene utilizzato se ho una sola stringa da deserializzare, gli viene passata come parametro in input la stringa e in output restituisce un'instanza, nel nostro caso, di *StringTuple* mentre il secondo metodo viene utilizzato quando si vuole deserializzare un Array Json di Tuple ottenendo quindi una lista Java di Tuple.
Per poter effettuare una richiesta verso il server è però necessario anche andare a specifcare il parametro *bulk*, un booleano che discrimina, lato server, se ho una lista di Tuple oppure se devo crearla. Inoltre, come nella maggior parte delle richieste Http è necessario specificare il body contenente la Tupla da mandare wrappata dentro la lista e serializzata come Json; questo però non basta, bisogna ricordare di settare nell'header sia il content type sia l'accept. Per rendere questo procedimento più semplice abbiamo provato a mettere su carta, per ogni tipologia di API, come deve essere strutturata la richiesta e di quali parametri il server ha necessità cercando poi, in uno step successivo, di accorpare quelli simili andando a migliorare sia la leggibilità del codice sia la sua struttura.



```
@Override
public CompletableFuture<RegularMatch> read(@NotNull RegexTemplate regexTemplate) {
    Log.d(TAG, msg: "TEMPLATE->" + regexTemplate);
    CompletableFuture cp = new CompletableFuture();
    Serializer<RegexTemplate> serializer = Presentation.INSTANCE.serializerOf(RegexTemplate.class, MIMETypes.APPLICATION_JSON);
    String tupleSerialized = serializer.toString(regexTemplate);
    HttpClientRequest request = this.client.request(HttpMethod.GET, this.port, host: "localhost", requestURI: this.path + this.name , new Handler<HttpClientResponse>() {
            @Override
            public void handle(HttpClientResponse event) {
                Log.d(TAG, msg: "response->" + event.statusMessage());
                if(event.statusCode() == 200) {
                    event.bodyHandler(new Handler<Buffer>() {
                        @Override
                        public void handle(Buffer event) {
                            String response = event.getString(0, event.length());
                            if(!response.isEmpty()) {
                                Log.d(TAG, msg: "sono in secondo handler" + event.getString(1, event.length()));
                                Deserializer<RegexTemplate> d = Presentation.INSTANCE.deserializerOf(RegexTemplate.class, MIMETypes.APPLICATION_JSON);
                                List<RegexTemplate> deserializerValue = d.listFromString(response);
                                Log.d(TAG, deserializerValue.toString());
                                ResponseRead responseRead = new ResponseRead(deserializerValue.get(1).toString());
                                responseQueue.add(responseRead);
                                startedService.handleMessage(responseRead, type: "read");
                                Log.d(TAG, msg: "RISPOSTA READ");
                            }else{
                                cp.completeExceptionally(new IllegalStateException("Remote Exception" ));
                                Log.d(TAG, msg: "Errore! Nessuna tupla trovata");
                            }
                        }
                    });
                }else {
                    cp.completeExceptionally(new IllegalStateException("Remote Exception" + event.statusCode()));
                    Log.d(TAG, msg: "Errore!!!->" + event.statusMessage());
                } }
        }).putHeader(HttpHeaders.CONTENT_TYPE.toString(), value: "application/json");
        request.end(tupleSerialized);
        return cp;
}
```

Figure 6.3: Screen richiesta API *read*.

Dopo aver implementato correttamente la *write* 6.3 attraverso l'uso della libreria Volley, ci siamo bloccate nello sviluppo delle primitive *read* e *take* in quanto necessitano di parametri in input.
Per questo motivo, analizzando a fondo la libreria, ci siamo accorte del fatto che Volley non mette a disposizione l'uso di parametri per effettuare richieste a meno che non siano di tipo POST. Di conseguenza siamo dovute ricorrere all'utilizzo di Vert.X in particolare introducendo una richiesta GET di tipo *HttpClientRequest* sulla quale è possibile impostare intestazioni e inserire dati all'interno del corpo.
Nonostante ciò abbiamo comunque scelto di mantenere l'implementazione della richiesta *write* sfruttando la libreria Volley per mostrare il paragone ed evidenziare le differenze tra le due librerie da noi utilizzate.

Chapter 7

Auto Validazione

Per validare il nostro progetto abbiamo proceduto con l'implementazione e l'esecuzione di alcuni test. Questi sono stati pensati in maniera tale da verificare le proprietà salienti del sistema ed il rispetto dei requisiti individuati in fase di progettazione.
TextOUTAPI Lo scopo del test è quello di dimostrare la corretta implementazione della primitiva *IN* **TuSoW**, con relativa risposta di avvenuto inserimento all'interno del Tuple Space. La tipologia di chiamata Http utilizzata per questo tipo di richiesta è POST.
TextRDAPI Lo scopo del test è quello di dimostrare la corretta implementazione della primitiva *RD* **TuSoW**, con relativa risposta contenete la Tupla letta dallo spazio di Tuple. La tipologia di chiamata Http utilizzata per questo tipo di richiesta è GET.
TextINAPI Lo scopo del test è quello di dimostrare la corretta implementazione della primitiva *OUT* **TuSoW**, con relativa risposta contente la Tupla letta dal Tuple Space ed avvenuta cancellazione della stessa. La tipologia di chiamata Http utilizzata per questo tipo di richiesta è GET seguita da DELETE.

Chapter 8

Esempi d'uso

Nel cAPItolo seguente sono raffigurati gli screen principali inerenti all'uso dell'applicazione Android.

All'avvio del sistema si aprirà la Main Activity contenete il login attraverso cui sarà possibile registrarsi come nuovo utente oppure effettuare l'accesso inserendo credenziali precedentemente memorizzate nel database.

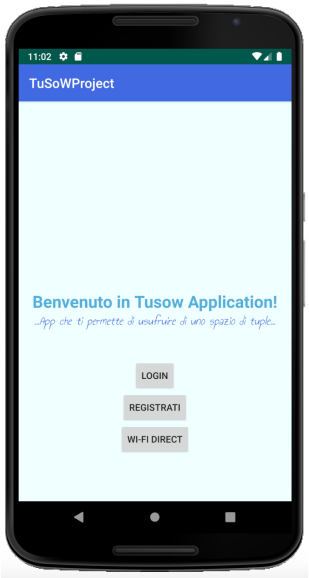


Figure 8.1: Interfaccia principale per il login dell'utente.

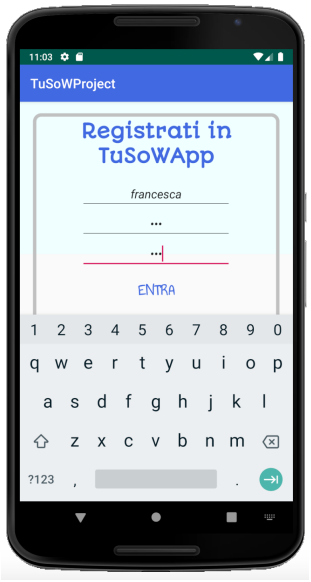


Figure 8.2: Interfaccia utente per effettuare una nuova registrazione nel sistema.

Al termine del login è possibile accedere direttamente all'Activity successiva raffigurante lo spazio dedicato dall'applicazione per le API disponibili per la modifica del TupleSpace, nello specifico parliamo delle operazioni *read*, *take* e *write*.

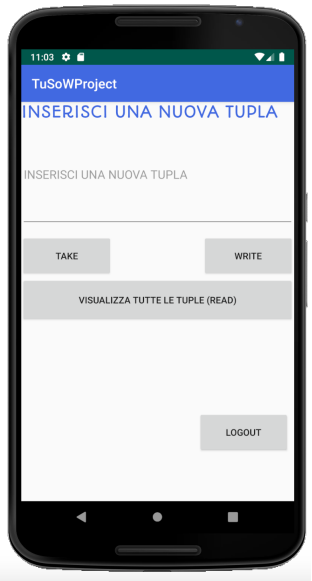


Figure 8.3: Interfaccia utente che mostra lo spazio di elaborazione e modifica del TupleSpace.

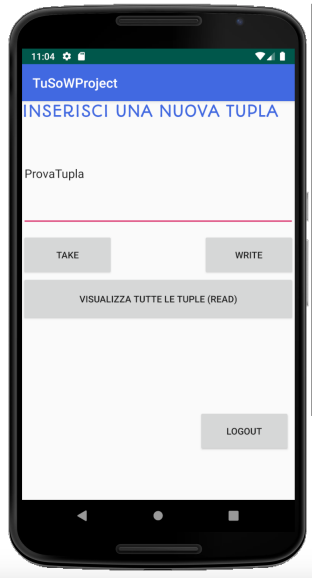


Figure 8.4: Interfaccia utente che mostra l'inserimento di una nuova Tupla nel TupleSpace.

Chapter 9

Wi-fi Direct

Un concetto appartenente allo sviluppo di tale progetto consiste nello studio e nella sperimentazione della tecnologia Wi-fi Direct che permette la comunicazione tra dispositivi dotati di tecnologia Wi-Fi.
Lo scopo è quello di consentire connessioni dinamiche, cioè prive di ruoli prestabiliti, che vengono invece definiti al momento della connessione. Questa tecnologia nasce in relazione all'uso di dispositivi mobili e distribuiti che non richiedendo alcun accesso ad Internet.
Nel nostro caso specifico la comunicazione tra dispositivi comporta la diffusione di Tuple all'interno di uno spazio condiviso, senza curarsi degli agenti o del concetto di mittente e destinatario ma, bensì broadcast.

9.1 Descrizione

Wi-Fi Direct, chiamato anche Wi-Fi P2P, è una tecnologia che consente ai dispositivi Android 4.0 (livello API 14) o successivi con l'hardware appropriato, di connettersi direttamente tra loro tramite Wi-Fi senza un punto di accesso intermedio.
Usando queste API è possibile connettersi con altri dispositivi supportano il Wi-Fi P2p comunicando così attraverso una connessione veloce, diretta e sicura, su distanze più lunghe rispetto una connessione Bluetooth. Questo aspetto risulta molto utile per applicazioni che, come nel nostro caso, richiedono la condivisione di risorse e dati tra più utenti connessi.
Le API appartenenti a questa nuova tecnologia sono composte dalle seguenti parti il cui uso è consentito a partire dalla classe WifiP2pManager:

1. Methods, consentono di rilevare e connettersi ai peer dello stesso gruppo.

2. Listeners, consentono di ricevere notifiche sull'esito positivo o negativo in seguito alle chiamate di metodo. É importante ricordare che ogni metodo può essere associato ad un prorpio listener passato come parametro allo stesso.

3. Intent, avvisano allo scattare eventi specifici rilevati dal framework come una connessione interrotta o un nuovo peer disponibile.

Spesso l'uso di queste tre componenti principali viene mischiato per effettuare certe operazioni. Ad esempio, è possibile effettuare una chiamata verso il metodo discoverPeers() della classe WifiP2pManager.ActionListener, in modo da poter ricevere una notifica con i metodi ActionListener.onSuccess() e ActionListener.onFailure().
Inoltre viene trasmesso un Intent WIFI_P2P_PEERS_CHANGED_ACTION se la chiamata del metodo discoverPeers() ottiene un elenco dei peer del gruppo cambiato.

9.1.1 Overview tecnica

I dispositivi abilitanti Wi-Fi Direct si cercano e scoprono tra loro formando un gruppo P2P. In ogni gruppo, un nodo eletto, chiamato *P2P capo gruppo* si connette attraverso l'Access Point mentre tutti gli altri rappresenteranno dei *client*.
Una novità che caratterizza questa tecnologia risulta essere, infatti, la dinamicità dei ruoli tra i peer del gruppo; ovvero il fatto che ciascun peer sia in grado di essere sia client che Access Point o capo gruppo.
Per stabilire una connessione quindi i dispositivi P2P devono concordare il ruolo che ciascuno di loro assumerà. Procedendo in maniera graduale andremo ad analizzare come tale comunicazione viene configurata, a partire dall'architettura generale che supporta il Wi-Fi Direct fino ad introdurre le principali procedure per il rilevamento dei dispositivi, la negoziazione dei ruoli tra peer ed altro ancora.

Architettura

I dispositivi P2P comunicano tra loro attraverso l'istituzione di gruppi P2P funzionalmente equivalenti alle infrastrutture di reti Wi-Fi tradizionali.
Dinamicamente all'interno del gruppo P2P viene identificato il proprietario (P2P GO o Owner) mentre tutti gli altri dispositivi agiranno come Client del sistema. Una volta istituito il gruppo altri Client possono accedervi man mano come in un gruppo di rete Wi-Fi tradizionale.
La figura del P2P GO è necessaria per eseguire il server DHCP (Dynamic Host Configuration Protocol) in modo da poter fornire ai client del gruppo gli indirizzi IP. Inoltre solo il proprietaro del gruppo può incrociare i collegamenti tra i vari dispositivi client su una rete esterna come ad esempio una infrastruttura WLAN.
Infine, la tecnologia Wi-Fi Direct non consente il trasferimento di ruolo ovvero nel caso in cui il P2P GO se ne vada, tutto il gruppo viene abbattuto in modo tale che debba essere ristabilita una connessione mediante l'uso di procedure specificate.

Formazione dei Gruppi

Esistono diversi modi in cui due o più dispositivi possono creare un gruppo P2P, di seguito verranno introdotti a partire dal caso più complesso denotato come caso Standard per poi introdurre due varianti più semplici denotate come Autonomo e Persistente.

- **Standard**: se i dispositivi non si conoscono e vogliono formare un nuovo gruppo dal nulla devono per prima cosa trovarsi e poi negoziare quale dei due deve adempiere al ruolo di Group Owner;
- **Autonomo**: un dispositivo può decidere indipendentemente di creare un gruppo P2P senza alcun client connesso, diventandone ovviamente il GO;
- **Persistente**: durante la creazione del gruppo i dispositivi possono salvarsi le credenziali per poterlo reinstanziare in un secondo momento.

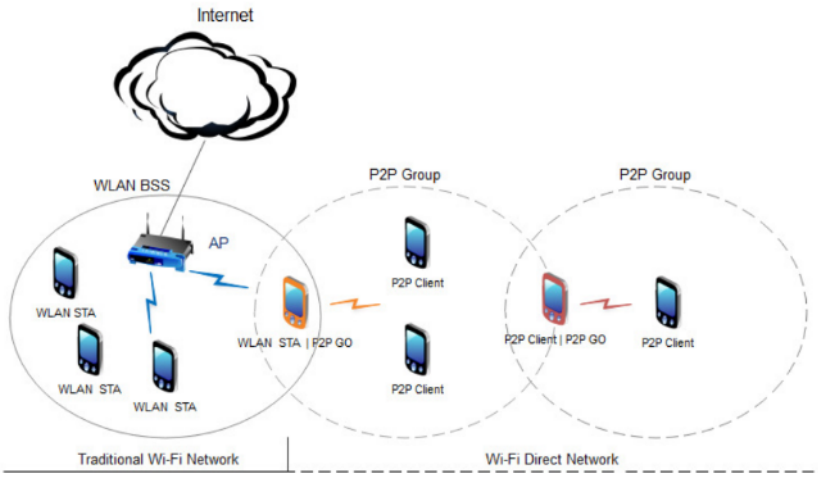


Figure 9.1: Architettura Wi-Fi e Wi-Fi Direct

Lo standard Wi-Fi Direct prevede la possibilità che un dispositivo sia connesso nello stesso momento a più gruppi diversi creando di fatto una rete a maglia sulla quale inviare informazioni in modalità broadcast.
Nel nostro caso la tecnologia Wi-Fi Direct deve permettere a più agenti distribuiti di connettersi allo stesso Tuple Space con il quale poter interagire scambiandosi Tuple ed effettuare operazioni attraverso l'uso delle primitive **TuSoW**.
Un aspetto importante su cui ci siamo concentrate è il fatto che utilizzando uno standard di questo tipo risulta ancora più limpido il fatto che gli utenti del sistema non hanno la necessità di conoscere la locazione fisica del Tuple Space.
Facciamo un esempio, ho due utenti di sistema Alice e Bob, entrambi hanno installato sul loro device l'applicazione da noi fornita che consente loro di utilizzare

un Tuple Space condiviso. Supponiamo inoltre che Bob fornisca l'interazione con l'applicazione per una serie di utenti esterni ed AAlice lo stesso. Gli utenti che comunicano con Bob e con Alice ai quali pongono le loro richieste non hanno necessità di sapere se il Service è effettivamente contenuto in Alice piuttosto che Bob. Bob ed Alice devono fornire un'interfaccia per la comunicazione che sia il più trasparente possibile.

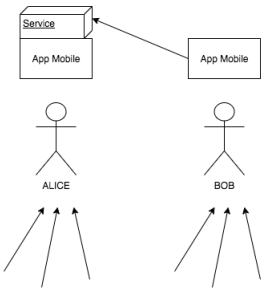


Figure 9.2: Interfaccia di comunicazione tra due utenti di sistema.

Implementare una comunicazione di questo genere è tutto forché banale, oltretutto utilizzando uno standard come Wi-fi Direct.

9.2 Esempi d'uso

La voglia di testare questa nuova tecnologia ci ha portato ad estendere il nostro sistema consentendo all'utente, a partire dalla Main Activity, di selezionare il bottone Wi-Fi Direct il quale consente ai dispositivi di connettersi tramite questo tipo di tecnologia. La nostra implementazione consente all'utente di attivare/disattivare il Wi-Fi del dispositivo in utilizzo tramite la semplice pressione di un bottone. Una volta attivato il Wi-Fi è possibile effettuare una fase di *discover* per poter trovare altri dispositivi connessi in rete tramite Wi-Fi. I dispositivi trovati vengono inseriti all'interno di una lista (identificati solo dal nome del dispositivo) e, tramite la pressione su di uno di questi è possibile connettervisi. Qui sotto una breve illustrazione del sistema appena descritto.In figura 9.4 vediamo come, dopo la fase di *discovery*, il sistema trova un dispositivo e lo aggiunge alla lista. In questo caso rileva un dispositivo nominato *Nexus5X* con cui poi avrà la possibilità di connettersi.

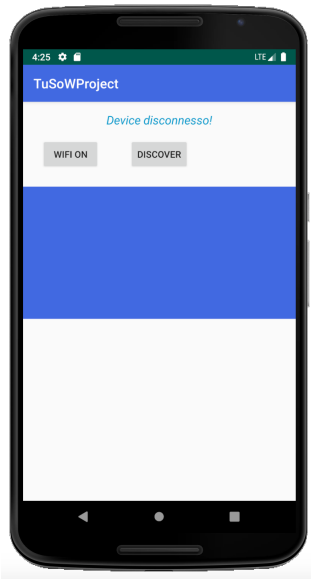


Figure 9.3: Interfaccia Wi-Fi Direct, prima della fase di discovery.

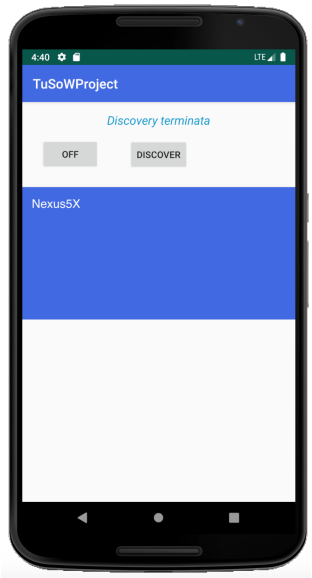


Figure 9.4: Interfaccia Wi-Fi Direct dopo la fase di discovery.

Chapter 10

Conclusioni

La realizzazione del progetto è stata indirizzata al raggiungimento degli obiettivi prefissati inizialmente. Ci riteniamo soddisfatte del lavoro svolto nonostante i tempi si siano allungati più del previsto. Complessivamente gli obiettivi prefissati sono stati raggiunti a pieno se pur il percorso non sia risultato privo di intoppi.
L'utilizzo di un moderno middleware distribuito, basato sul modello di coordinazione ***LINDA***, all'interno di un ambiente Android per realizzare un Tuple Space condiviso ci ha portato a risolvere i seguenti passaggi:

- Implementazione un sistema dinamico a cui possono connettersi più activty,
- Implementazione una comunicazione tra Activity/Service e Tuple Space tramite chiamate Http,
- Armonizzazione delle API **TuSoW** con i costrutti forniti da Android.
- Studio del middleware **TuSoW**,
- Studio di una libreria Android che permetta di effettuare chiamate Http,
- Ricerca e studio di una tecnologia che permetta di usufruire dello standard Wi-fi Direct per la comunicazione.

Il sistema realizzato ovviamente non risulta essere perfetto nonostante l'impegno, per questo motivo proviamo a soffermarci su alcune criticità che durante lo svolgimento del progetto ci hanno messo alla prova.

10.1 Criticità riscontrate

Durante lo svolgimento dell'intero progetto ci siamo imbattute in problemi che ci hanno messo a dura prova. La criticità che a nostro parere ci ha preso

molto tempo è l'implementazione delle chiamate Http verso il server. Utilizzando un server già fatto e dovendo per questo motivo sottostare a determinate modalità di utilizzo inizialmente non è stato assolutamente semplice cAPIre quali parametri dovevano essere passati e in quale modalità. Inoltre la libreria Volley, messa a disposizione da Android per le chiamate Http, non ha sicuramente aiutato in quanto ha una sintassi tutta sua che spesso non consente all'utilizzatore di cAPIre a fondo gli errori (che oltretutto non sono esaustivi). Inoltre un'ulteriore problematica riscontrata riguarda l'utilizzo del Service da noi implementato e la gestione del numero di richeste da dover gestire. Questa problematica riscontrata ci ha portato a pensare che fosse la reale applicazione del CAP Theorem[2], affrontato a lezione. Il CAP Theorem dimostra come in un sistema distribuito, sia impossibile la presenza delle seguenti tre proprietà contemporaneamente:

1. Consistenza, ovvero la coerenza dei dati. Questa indica che ogni utente che accede al Tuple Space debba vedere le stesse informazioni.

2. Disponibilità, ovvero la garanzia che ogni richiesta riceva una risposta.

3. Tolleranza ai guasti.

Il CAP Theorem prevede che al più possano essere presenti due delle tre sopracitate proprietà. Nel nostro progetto abbiamo dato più importanza alla consistenza dei dati con conseguente applicazione delle proprità 1 e 2. Viene violata la tolleranza ai guasti in quanto, risulta essere necessaria per l'utente la garanzia che ogni richiesta verso il Tuple Space riceva una risposta la quale può anche non essere sempre di successo.

10.2 Lavori futuri

La realizzazione dell'intero progetto si è basata sull'implementazione ed utilizzo di tre principali API (in, rd, out). In futuro saraà possibile estendere il progetto aggiungendo anche le primitive più avanzate come la *WriteAll*, *ReadAll*, *TakeAll*, ecc.. Inoltre noi ci siamo concentrate sullo sviluppo di un Tuple Space prettamente testuale, in realtà è possibile implementare anche un Tuple Space che preveda l'inserimento la lettura e la consumazione di Tuple logiche.
All'interno del nostro progetto è stata implementata una fase di login/registrazione per gli utenti che hanno la necessità di avere un Tuple Space condiviso. Una possibile espansione del progetto potrebbe prevedere l'utilizzo delle credenziali utente come forma di identificazione all'interno del TupleSpace. Il nome utente potrebbe essere aggiunto a qualunque tipologia di richiesta permettendo, così, di inserire all'interno Tuple Space non so la Tupla (testuale o logica) ma, anche l'utente che la ha condivisa.
Grazie all'utilizzo di un database per mantenere le informazioni riguardanti gli utenti registrati, si potrebbe prevedere una funzionalità che permetta ad un utente di associare ad una tipologia di richiesta un mittente scegliendo tra gli utenti di sistema. Ad esempio, se nella lista utenti sono presenti:

- Bob,
- Alice,
- Pippo,

e ciascuno è a conoscenza degli altri utenti, nel caso in cui Bob voglia effettuare una richiesta di tipo *write* sul Tuple Space andando a specificare la Tupla "CIAO" può decidere di direzionarla ad uno specifico destinatario ad esempio Alice, inserendo l'identificativo (che potrebbe essere il nome) nel payload del messaggio.
Per quanto riguarda la parte di progetto relativa al Wi-fi Direct è stata testata solamente per quanto riguarda la connessione tra dispositivi. Uno sviluppo futuro, di integrazione al sistema, potrebbe essere quello di prevedere la possibilità che due dispositivi, dopo la fase di connessione, possano scambiarsi messaggi in modo tale da poter interagire, ad esempio, in maniera del tutto astratta con lo spazio di Tuple. Cosa intendiamo con astratta? Con la parola astratta intendiamo senza la necessità che gli utenti siano a conoscenza della locazione fisica del Tuple Space.

10.3 Cosa abbiamo imparato

Questo progetto ha arricchito notevolmente il nostro bagaglio di conoscenze anche grazie all'intersezione con altre discipline ovvero Programmazione di Sistemi Mobile e Sistemi Distributi. Innanzitutto abbiamo ampliato la nostra conoscenza per quanto riguarda la programmazione su piattaforma Android inolte abbiamo appreso molto del funzionamento di ***LINDA***, compreso di struttura su cui è basata l'implementazione di **TuSoW**. Di notevole rilevanza è stata esperienza basata sull'uso del server di **TuSoW** e, della relativa implementazione delle API che ci hanno dato la possibilità di mettere le mani in modo più approfondito all'interno di strumenti e tecnologie come Vert.x.
Inoltre, più in generale, abbiamo esteso le nostre competenze riguardo ai sistemi e middleware distribuiti che ad oggi ricoprono un importante ruolo nella programmazione. Grazie a questo progetto, infatti, abbiamo potuto comprendere meglio gli aspetti trattati durante corso.

Bibliography

[1] Volley android. Technical report, 2002.

[2] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[3] Xiaoshan Li, Zhiming Liu, and He Jifeng. A formal semantics of uml sequence diagram. In *2004 Australian Software Engineering Conference. Proceedings.*, pages 168–177. IEEE, 2004.

[4] Madhan M Thirukonda and Ronaldo Menezes. On the use of linda as a framework for distributed database systems. Technical report, 2002.