Alma Mater Studiorum · Università di Bologna

SECONDA FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in Ingegneria Elettronica, Informatica e Telecomunicazioni

FRAMEWORK MOBILE PER LO SVILUPPO DI SISTEMI COORDINATI

Elaborata nel corso di: Sistemi Distribuiti

Relatore:
Prof. ANDREA OMICINI

Presentata da:
ROBERTO D'ELIA

Correlatore:
Dott. STEFANO MARIANI

Sessione III
Anno Accademico 2011-2012

Desidero ringraziare il Prof. Andrea Omicini e l'Ing. Stefano Mariani per il prezioso contribuito dato al mio lavoro, i consigli, la disponibilità e la simpatia dimostrata. Ringrazio, inoltre, tutti coloro che hanno sempre creduto nelle mie capacità e che mi hanno supportato in questo percorso.

Indice

Elenco delle figure

Introduzione

Lo sviluppo del modo mobile che si sta attuando in questi ultimi anni è considerato un passo importante nell'evoluzione tecnologica. Si intravedono nuovi orizzonti e possibilità che un tempo, con gli strumenti a disposizione, erano impossibili da realizzare. Basti pensare alle numerose attività che possiamo ora eseguire, in qualunque momento e ovunque noi siamo, attraverso l'utilizzo di uno smartphone, invece che restando davanti al computer dell'ufficio o di casa.
Il mercato dei smartphone e dei tablet è in continuo aumento così come lo sono i contesti nei quali essi vengono utilizzati. Il grande numero di utenti al seguito non può essere più ignorato dagli sviluppatori di applicazioni che sono stati costretti ad estendere il loro proprio raggio d'azione nella produzioni di prodotti competitivi. Già in fase di progettazione, infatti, i servizi vengono realizzati per includere funzionalità compatibili con i dispositivi mobili presenti sul mercato.
Anche il campo della coordinazione, perciò, può beneficiare di questi nuovi strumenti sia per il suo sviluppo sia per rendere accessibili le proprie funzionalità ad un pubblico più vasto. In questa tesi si parlerà, in particolare, dell'applicazione Inspector, uno dei tools più importanti del modello di coordinazione TuCSoN. L'obbiettivo prefissato, quindi, è quello di rendere disponibili le funzionalità dell'Inspector anche nell'ambiente mobile. Per fare questo, nel primo capitolo verranno introdotti i concetti fondamentali del progetto TuCSoN. Nel secondo capitolo, invece, verrà descritta brevemente la suite di strumenti Android attraverso la quale faremo il nostro ingresso nel mondo mobile. Nel terzo capitolo, quindi, verranno descritte le problematiche che dovranno essere affrontate per il raggiungimento dello scopo finale della tesi: la realizzazione di un framework mobile per lo sviluppo di sistemi coordinati.

Capitolo 1

Modelli di Coordinazione e TuCSoN

In questo capitolo eseguiremo una panoramica del contesto in cui agisce l'Inspector. Si tratta di un capitolo di fondamentale importanza per capire le funzionalità e il ruolo dell'Inspector di cui inizieremo a descrivere le caratteristiche principali proprio alla fine di questo capitolo. Affronteremo, quindi, le motivazioni che spingono ad adottare un modello di coordinazione e le sue caratteristiche principali; in particolare, inizieremo ad analizzare il modello *Linda* fino ad arrivare al progetto TuCSoN. Introdurremo, quindi, il concetto di centro di tuple e la sua implementazione tramite linguaggio ReSpecT.

1.1 Verso Un Modello

Lo sviluppo tecnologico e scientifico ha portato alla creazione di sistemi complessi, composti da unità computazionali dotati della tecnologia necessaria per interagire tra loro. Tutto ciò ha provocato un sostanziale aumento del numero di computazioni che possono verificarsi in ogni istante in maniera concorrente, distribuita e su contesti di esecuzione differenti ed eterogenei. Basti pensare, infatti, alla crescente varietà di device introdotti sul mercato con cui abbiamo a che fare ogni giorno (telefoni cellulari, elettrodomestici) che ci fanno vivere come immersi in una grande nube computazionale. Per il raggiungimento di un obbiettivo comune, che renda di fatto sociali queste interazioni, è indispensabile adottare un modello che imponga leggi di coordinazione e sicurezza tra le molteplici attività.

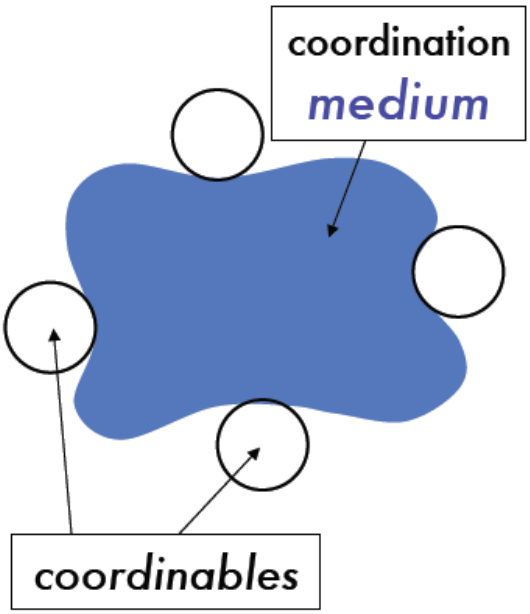

Figura 1.1: Entità coordinabili inserite in un coordination medium

Un sistema coordinato, secondo il meta–modello proposto da Ciancarini, è costituito da tre tipi di componenti:

- Entità Coordinabili
- Medium di coordinazione
- Leggi di Coordinazione

I coordinabili sono le entità che devono essere coordinate (quindi threads, processi, utenti...). Per fare questo devono interagire con il medium di coordinazione che possiede anche la funzione di collante, aggregando le varie entità. Le entità coordinabili (in realtà indipendenti e separate) sono abilitate e governate in accordo con le leggi di coordinazione. Tali leggi sono espresse in termini di *linguaggio di comunicazione*, ossia la sintassi usata per le strutture dati scambiate, e in *linguaggio di coordinazione*, cioè la sintassi e semantica delle primitive utilizzate dalle entità coordinabili per interagire con il medium di coordinazione.
Esistono due classi per i modelli di coordinazione:

- **Control–Oriented:** focalizzati sull'atto di comunicazione vero e proprio.

- **Data–Oriented:** focalizzati sull'informazione scambiata durante la comunicazione. Il medium di coordinazione diventa un deposito di informazione condivisa a cui possono accedere le entità coordinabili disaccoppiate temporalmente. Fanno parte di questa classe i modelli di coordinazione Tuple–based che andremo ad analizzare.

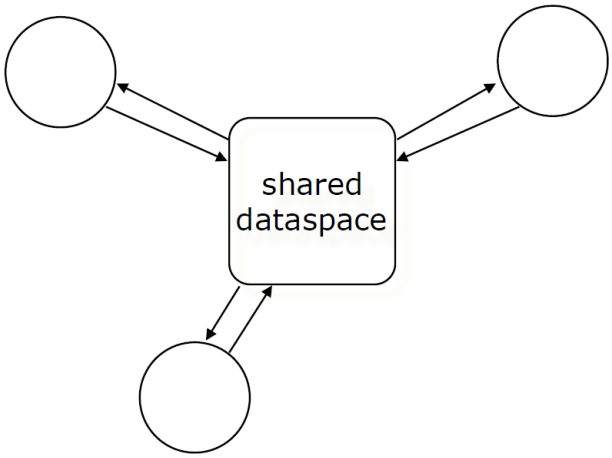

Figura 1.2: Modello tuple-based: Entità che interagiscono con un deposito condiviso

1.2 Modello Tuple–Based: Linda

Presenteremo qui il modello di coordinazione che sta alla base di TuCSoN : *Linda*. *Linda* è uno dei modello di coordinazione tuple–based più noti. Sfrutta come medium di coordinazione un deposito condiviso, che contiene un insieme di strutture chiamate *tuple*. Il deposito condiviso viene chiamato spazio di tuple (*tuple space*). Il linguaggio di comunicazione di *Linda* specifica cosa si intenda per tuple, ossia una collezione ordinata di dati tra loro anche eterogenei: rappresentano, quindi, l'unità elementare di informazione che può essere scambiata.

Esempio: t(10), studente(Mario,Rossi,21)...

Tramite meccanismi di matching, inoltre, è possibile rilevare l'appartenenza di una tupla alla relativa classe/insieme di tuple detta *template* o *anti–tupla*.

Esempio: t(X), studente(N,C,E)...

1.2.1 Il Linguaggio Di Coordinazione

Le entità coordinabili devono essere in grado di interagire con il medium di coordinazione affinchè le loro attività possano essere coordinate. Nel modello Linda le entità coordinabili possono interagire con lo spazio di tuple attraverso l'uso di primitive, la cui sintassi e semantica viene descritta dal linguaggio di coordinazione.

Primitive Base:

- **out(T):** inserisce la tupla T nello spazio di tuple.
- **in(TT):** recupera la tupla che fa matching con il template TT. Si tratta di una lettura distruttiva in quanto la tupla viene rimossa dallo spazio di tuple. Se più di una tupla fa matching con TT, ne viene scelta una in modo non deterministico. Se nessuna tupla fa matching con TT, l'operazione viene sospesa finchè una tupla non viene trovata (semantica sospensiva).
- **rd(TT):** viene letta la tupla che fa matching con il template TT. In questo caso la lettura non è distruttiva, quindi la tupla non viene rimossa dallo spazio di tuple.

Se più di una tupla fa matching con TT, ne viene scelta una in modo non deterministico. Anche questa volta viene adottata una semantica sospensiva nel caso in cui nessuna tupla faccia matching con TT.

Primitive predicative:

- **inp(TT):** simile a *in(TT)* tranne per il fatto che viene abbandonata la semantica sospensiva in favore di una semantica successo/fallimento.
- **rdp(TT):** simile a *rd(TT)* tranne per il fatto che viene abbandonata la semantica sospensiva in favore di una semantica successo/fallimento.

Semantica successo/fallimento: nel caso in cui nessuna tupla fa matching con il template TT, l'operazione termina con un fallimento, altrimenti viene riportata la tupla.

Primitive Bulk:

- **in_all(TT):** recupera l'insieme di tuple che fa matching con il template TT.Se nessuna tupla viene trovata viene restituito un insieme vuoto. La lettura è distruttiva.
- **rd_all(TT):** recupera l'insieme di tuple che fa matching con il template TT.Se nessuna tupla viene trovata viene restituito un insieme vuoto. La lettura non è distruttiva.

Le primitive predicative e quelle Bulk, in realtà, non erano presenti nella versione base di *Linda*, ma sono state introdotte per risolvere determinate casistiche di problemi. Le primitive Bulk, ad esempio, vengono utilizzate nei casi in cui è necessario gestire più tuple con una sola primitiva.

1.2.2 Caratteristiche Principali

- **Comunicazione Generativa:** le tuple generate dalle entità coordinabili hanno un'esistenza propria e indipendente da chi li ha generate. Le tuple non sono collegate a entità specifiche ma equamente accessibili da tutti i coordinabili. Sono, perciò, disaccoppiate nello spazio, nel tempo e nel nome.

- **Accesso Associativo:** è possibile accedere alle tuple tramite l'uso dei template e dei meccanismi di matching, ossia tramite il loro contenuto/struttura e non in base al loro nome, indirizzo o localizzazione.

- **Semantica Sospensiva:** alcune operazione possono sospendersi nel caso in cui nessuna tupla viene trovata, ed essere riprese quando le tuple risultano nuovamente disponibili. Questo permette di sincronizzare le varie unità computazionali che interagiscono con lo spazio di tuple.

1.3 Centri di Tuple

1.3.1 Verso i centri di tuple

Possiamo notare che, nel modello *Linda*, il comportamento del medium di coordinazione è fissato una volta per tutte. Questa è chiaramente una caratteristica che rende il modello poco flessibile, in quanto sarà sì in grado di risolvere bene alcuni problemi, ma altri non troveranno soluzione. Come conseguenza, il peso della coordinazione viene spesso messo a carico delle entità coordinabili che, al contrario, dovrebbero essere mantenute il più semplici possibili senza preoccuparsi della coordinazione. Questa tendenza va chiaramente contro ai principi ingegneristici e produce un software non compatibile a scenari aperti. Stesso risultato viene ottenuto attraverso l'utilizzo di primitive ad hoc (come nel caso delle primitive Bulk), le quali si limitano a risolvere solo specifici problemi.
La soluzione è rendere il comportamento del modello di coordinazione compatibile con il problema che si presenta, avendo cioè la possibilità di incapsulare nuove funzionalità all'interno del medium di coordinazione senza intaccare la sintassi e la semantica delle

leggi di *Linda*. Se il comportamento del medium di coordinazione non è fissato una volta per tutte, allora possiamo definirlo a nostro piacimento in accordo con le nostre necessità.
Per avere questa possibilità, è necessario aggiungere un livello Control–Driven accanto a quello Information–Driven nativo di *Linda*, dirigendoci verso un'architettura che possiamo definire *ibrida*.

1.3.2 Caratteristiche Principali

Per Centro di Tuple(*Tuple Centre*) si intende uno spazio di tuple che include una specifica comportamentale collegata a particolari eventi coordinativi. La specifica viene espressa in termini di *reaction specification language* che definisce un insieme (anche vuoto) di attività computazionali dette *reactions* e le associa ad un evento del centro di tuple.
Ogni reazione può in principio:

- accedere e modificare il corrente stato del centro di tuple, rimuovendo o aggiungendo tuple.
- accedere alle informazioni dei relativi eventi che sono completamente osservabili.
- invocare primitive su altri centri di tuple.

Come risultato, le operazioni di coordinazione effettuabili possono essere complicate quanto come la specifica applicazione lo richiede. Dal punto di vista dei processi, il risultato dell'invocazione di una primitiva in un centro di tuple è la somma degli effetti della primitiva stessa e di tutte le reactions ad essa collegate, percepite però come una singola transizione di stato. Se in un centro di tuple non viene inserita alcuna specifica, si comporterà come un normale spazio di tuple.

1.4 ReSpecT

ReSpecT (*Reaction Specification Tuples*) è un linguaggio tramite il quale è possibile programmare la specifica di comportamento dei centri di tuple. I centri di tuple ReSpecT infatti, possiedono una duplice nozione di centro di tuple:

- **tuple space:** l'insieme delle tuple logiche ordinarie. Incapsula la conoscenza.
- **specification space:** tuple logiche ReSpecT dette reactions che specificano il comportamento. Incapsula la coordinazione.

Tramite questo linguaggio possiamo, quindi, definire le reactions del centro di tuple (aspetto procedurale) ma non solo: è necessario associare le reactions agli eventi che possono verificarsi nel nodo che ospita il tuple centre(aspetto dichiarativo).

1.4.1 Aspetto Procedurale

Come già annunciato precedentemente, la semantica delle reaction è *transazionale*:

- una ReSpecT reaction ha successo solo se tutte reactions ad essa relative hanno successo, altrimenti fallisce.
- se fallita, la ReSpecT reaction non ha effetti sul ReSpecT tuple centre
- tutte le reazioni associate ad un evento vengono eseguite prima che si verifichi l'evento successivo: in questo modo i vari agenti risultano trasparenti alla catena di reazioni e percepiscono quest'ultima come un unica transizione atomica.

1.4.2 Aspetto Dichiarativo

Le tuple di specifica possiedono la sintassi

reaction(E,G,R)

Questa notazione associa tutte le reactions contenute in R, agli eventi descritti in E solo se le condizioni specificate nella guardia G sono verificate.

1.4.3 Proprietà dei ReSpecT Tuple Centre

- **Inspectable:** ispezionabili sia dai processi attraverso l'uso delle primitivi *rd* o *in*, sia dagli ingegneri attraverso l'uso dello strumento *Inspector* (che introdurremo poco più avanti).
- **Malleability:** è possibile modificare la specifica di comportamento o lo spazio di tuple ordinario attraverso le primitive *in_s* e *out_s*) o *in* e *out*.
- **Linkability:** è possibili accedere e modificare centri di tuple distribuiti in tutta la rete grazie all'identificatore completo.

1.5 TuCSoN

TuCSoN (*Tuple Centres Spread over the Network*) è un modello di coordinazione per processi e agenti (autonomi e mobili) basato su centri di tuple localizzati su nodi sparsi nella rete. In particolare le entità principali sono (compatibilmente al meta–modello proposto da Ciancarini):

- **TuCSoN Agent:** le entità coordinabili.
- **TuCSoN Tuple Centres:** centri di tuple distribuiti nella rete, rappresentano il medium di coordinazione.
- **TuCSoN Node:** l'astrazione topologica di base che ospita i centri di tuple.

I TuCSoN Agent sono entità pro–attive che necessitano di essere coordinate tra loro. Per far questo devo interagire con i centri di tuple attraverso l'uso di primitive rese disponibili dal TuCSoN Coordination Language. I centri di tuple sono entità reattive che offrono uno spazio condiviso programmabile tramite linguaggio ReSpecT . La loro mobilità, a differenza dei TuCSoN Agent, è legata al dispositivo che li ospita; un singolo device, infatti, può ospitare più centri di tuple. Agenti, centri di tuple e nodi sono identificati univocamente all'interno del *TuCSoN System*. La sintassi utilizzata per identificare un nodo ospitato su un dispositivo di rete è:

netid : portno

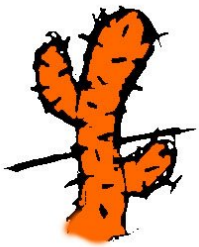

Figura 1.3: Logo TuCSoN

- **netid:** l'indirizzo IP o l'entry DNS del device che ospita il nodo nella rete.
- **portno:** numero di porta dove il TuCSoN coordination service rimane in ascolto dell'invocazioni di operazioni di coordinazione.

I vari centri di tuple ospitati da un nodo vengono identificati attraverso un termine *Prolog ground* (senza variabili); la sintassi completa, quindi, è:

tname @netid : portno

Anche i TuCSoN Agent sono indicati attraverso un termine Prolog ground a cui viene affiancato, al momento del loro ingresso nel TuCSoN System, un *universally unique identifier* (UUID):

aname : uuid

1.5.1 Architettura base

TuCSoN è un Java–based middleware e Prolog–based (in particolare *tuProlog*). Come detto all'inizio del paragrafo, il modello di coordinazione TuCSoN è costituito da una collezione di nodi distribuiti nella rete. Il *TuCSoN Node* è caratterizzato dal device di rete che lo ospita e dal numero di porta dove il *TuCSoN Service* ascolta le richieste in arrivo. Il TuCSoN Service è incaricato di:

- rimanere in ascolto delle invocazioni di operazioni su una particolare porta del device di rete.
- trasferire le invocazioni ricevute al target tuple centre.
- ritornare l'operazione completata al target agent.

Un singolo device di rete può ospitare più nodi, ognuno dei quali si affaccia su un numero di porta differente. Il numero di porta di default utilizzato in TuCSoN è 20504; così un agente può invocare un'operazione utilizzando la sintassi

tname @ netid ? op

senza specificare il numero di porta (viene sottinteso quello di default). Ogni TuCSoN Node fornisce un centro di tuple di default (chiamato appunto *default*) che è il destinatario di tutte le operazioni ricevute dal nodo nelle quali non si è indicato il target tuple centre. Come risultato il TuCSoN node accetta la seguente sintassi :

@netid : portno ? op

Con *global coordination space* si intende la collezione di tutti i centri di tuple che si trovano sulla rete, identificati dal loro nome completo. Dato un device di rete, invece, *local coordination space* indica l'insieme dei centri di tuple ospitati dal device; essi possiedono, quindi, lo stesso netid ma diverso portno. Un agente che si trova in esecuzione sullo stesso device puo' interagire con il local coordination space invocando le operazioni nella forma:

tname : portno ? op

1.5.2 Il Linguaggio Di Coordinazione

Il TuCSoN Coordination Language permette ai TuCSoN Agent di interagire con i centri di tuple eseguendo delle operazioni di coordinazione. Le operazioni di coordinazione consentono agli agenti di leggere, scrivere e di consumare tuple all'interno dei centri di tupla, al fine di sincronizzarsi tra loro. Ogni operazione è eseguita, perciò, da un source agent su un target tuple centre ed è composta da due fasi:

- **invocation:** la richiesta effettuata da un source agent ad un target tuple centre, contenete tutte le informazioni necessarie per l'invocazione
- **completion:** la risposta che viene data dal target tuple centre al source agent, contenete tutte le informazioni di esecuzione dal tuple centre.

La sintassi utilizzata per le operazioni di coordinazione è

tcid ? op

dove tcid indica il nome completo del centro di tuple, quindi:

tname @ netid : portno ? op

Primitive di base:

Molte delle primitive base sono recuperate da Linda e corrispondo in tutto e per tutto: *out, rd, rdp, in, inp.* Alcune nuove sono state introdotte:

- **no(TT):** verifica se sono presenti tuple che fanno match con TT. Se non sono presenti l'esecuzione termina con successo e TT viene ritornato, altrimenti l'esecuzione viene sospesa fino a quando nessuna tupla fa match.

- **nop(TT):** è la versione predicativa di no(TT), se viene trovata una tupla che fa match con TT l'esecuzione termina con fallimento e la tupla che fa match viene ritornata.

- **get():** legge tutte le tuple del target tuple centre e ritorna una lista completa. Se nessuna tupla viene trovata, ritorna una lista vuota.

- **set(Tuples):** sovrascrive il target tuple centre con una nuova lista di tuple. Una volta terminata ritorna la lista.

Primitive Bulk:

Oltre alle *rd_all* e *in_all* sono state introdotte le seguenti:

- **out_all(Tuples):** inserisce nel target tuple centre la lista di tuple data in ingresso.

- **no_all(TT):** verifica la presenza di tuple che fanno match con TT.Ritorna una lista vuota in caso di successo, l'intera lista di tuple in caso di fallimento.

Primitive uniformi:

Le primitive uniformi inseriscono meccanismi probabilistici all'interno di quelli coordinativi, utili per modellare il comportamento stocastico di un dato sistema coordinativo. *urd, urdp, uin, inp, uno, unop* sostituiscono le primitive non deterministiche di *Linda* assicurando una distribuzione probabilistica uniforme.

Primitive di Meta–Coordinazione:

Attraverso le primitive di meta–coordinazione è possibile interagire con la specifica dei centri di tuple nello stesso modo con cui i TuCSoN agent interagiscono con i centri di tuple. Le seguenti operazioni sono coerenti con le loro controparti:

- rd_s, in_s, out_s
- rdp_s, inp_s
- no_s, nop_s
- get_s, set_s

1.5.3 Agent Coordination Contexts

L' *Agent Coordination Context* (ACC) è una sorta di interfaccia runtime e stateful che viene fornita dall'infrastruttura all'agente che vuole interagire con il sistema. L'ACC permette di abilitare ma allo stesso tempo limitare le ammissibili operazioni; funge quindi da mediatore dell'interazione, offrendo all'utilizzatore una visione personalizzata del sistema. L'ACC dovrebbe essere rilasciato solo attraverso una negoziazione fra agente e sistema, nel quale si stabilirà l'insieme delle operazioni ammissibili. Possiamo considerare l'ACC, quindi, come l'astrazione rappresentante il modello RBAC (*Role–Based Access Control*): un unico framework coerente che integra aspetti di organizzazione e sicurezza. In conclusione, possiamo dire che vengono assegnate delle regole ai vari agenti, basate sull'organizzazione in cui essi operano, per regolare l'accesso alle risorse e ai servizi. Nell'implementazione di TuCSoN 1.10.3.0206 l'ACC viene supportato solo parzialmente. Non è previsto, infatti, il meccanismo di negoziazione di rilascio dell'ACC necessario per interagire con il centro di tupla. Il sistema rilascerà, invece, sempre un *EnnhancedACC* che permetterà semplicemente di eseguire tutte le operazioni esistenti.

1.6 L'Inspector

L'Inspector è un tool implementato in linguaggio Java, dotato di interfaccia grafica che permette agli ingegneri e amministratori del sistema TuCSoN di monitorare lo spazio di coordinazione. Si tratta di uno strumento ancora in fase di sviluppo il cui codice sorgente e quasi interamente contenuto all'interno dei package *alice.tucson.introspection* e *alice.tucson.introspection.tools* del jar TuCSoN. Lo spazio di coordinazione in TuCSoN, come sappiamo, è costituito dell'insieme dei centri di tuple distribuiti nella rete. Attraverso l'inspector è possibile analizzare lo stato attuale dei centri di tuple focalizzandosi sugli aspetti fondamentali. Per fare questo, quindi, l'Inspector implementa un TuCSoN agent che insieme a tutti gli altri agenti costituiscono le entità coordinabili del sistema. È possibile,infatti, lanciare più istanze dell'Inspector via linea di comando digitando l'istruzione :

java -cp TuCSoN-1.10.3.0206.jar alice.introspection.tools.Inspector

Possono essere inserite le seguenti opzioni:

- **-aid:** il nome dell'Inspector agent
- **-netid:** l'indirizzo ip del device che ospita il centro di tuple da ispezionare
- **-portno:** il numero di porta dove il servizio TuCSoN è in attesa di richieste
- **-tcname:** il nome del centro di tuple da monitorare

Se l'inspector viene lanciato senza specificare le opzioni qui sopra elencate, comparirà a video la prima interfaccia grafica nella quale sarà possibile inserire i vari parametri atti ad identificare in maniera univoca il centro di tuple da monitorare.
Nella schermata successiva è possibile accedere a quello che si vuole ispezionare all'interno del centro di tuple:

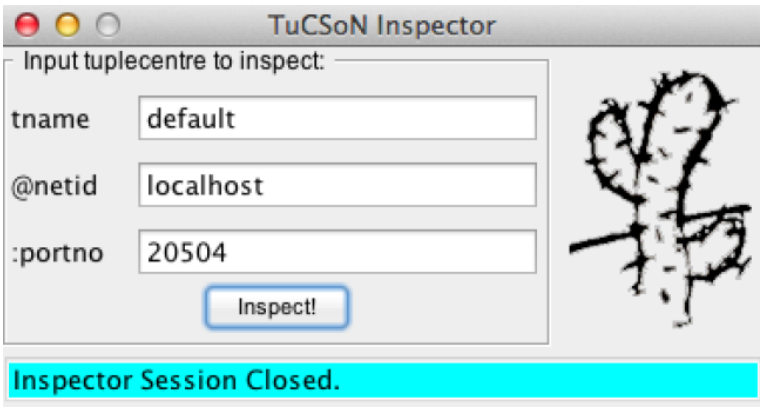

Figura 1.4: Interfaccia iniziale Inspector

- **Tuple Space:** visualizza lo stato dello spazio di tuple ordinario. Consente, quindi, di mostrare l'intera lista di tuple e visualizzarne il numero totale. È possibile scegliere, inoltre, se ispezionare lo spazio di tuple in maniera proattiva o reattiva. Il metodo proattivo consente di avere la lista aggiornata di tuple ogni qualvolta si verifichi un cambiamento di stato (quindi un update automatico), il metodo reattivo consente, tramite l'utilizzo del pulsante Observe!, di richiedere l'update della lista al TuCSoN Service ogni qual volta lo si desidera. Sono state inserite altre funzionalità tra cui la possibilità di filtrare i risultati inserendo un template Prolog ammissibile e di stamparli su un file log.

- **Pending Ops:** visualizza l'insieme delle operazioni che sono in attesa di essere completate. Tipicamente le operazioni visualizzate sono quelle dalla semantica sospensiva che per qualche motivo non sono terminate immediatamente. Oltre all'operazione sospesa viene indicata l'entità agente che ha invocato la primitive e il target tuple centre. Anche in questa grafica è possibile selezionare il metodo proattivo e reattivo, filtrare i risultati e stamparli su un file log.

- **ReSpecT Reactions:** visualizza l'insieme delle tuple di specifica (reactions) che sono state innestate dalle operazioni sul centro di tuple, e che sono terminate con successo o fallimento. Viene mostrata, quindi, l'intera reaction, lo stato con cui

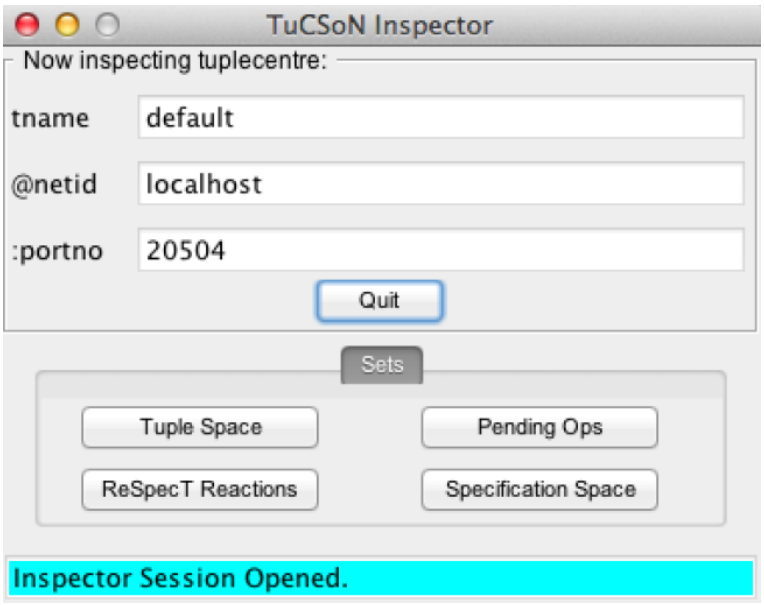

Figura 1.5: Interfaccia di selezione Inspector

è terminata, l'istate di tempo nel quale essa è avvenuta. Ulteriore funzionalità è quella di stampare i risultati ottenuti su un file di log.

- **Specification Space:** visualizza lo stato dello spazio di tuple di specifica. Consente, quindi, di mostrare l'intera lista di reactions con la quale si è implementato il comportamento del centro di tuple. È possibile, inoltre, modificare la corrente implementazione e inviarla al centro di tuple attraverso il pulsante set_s. Invece che modificare direttamente l'implementazione è possibile caricare un file che descrive la specifica desiderata. Ulteriore funzionalità è quella di stampare la specifica ottenuta su un file di log.

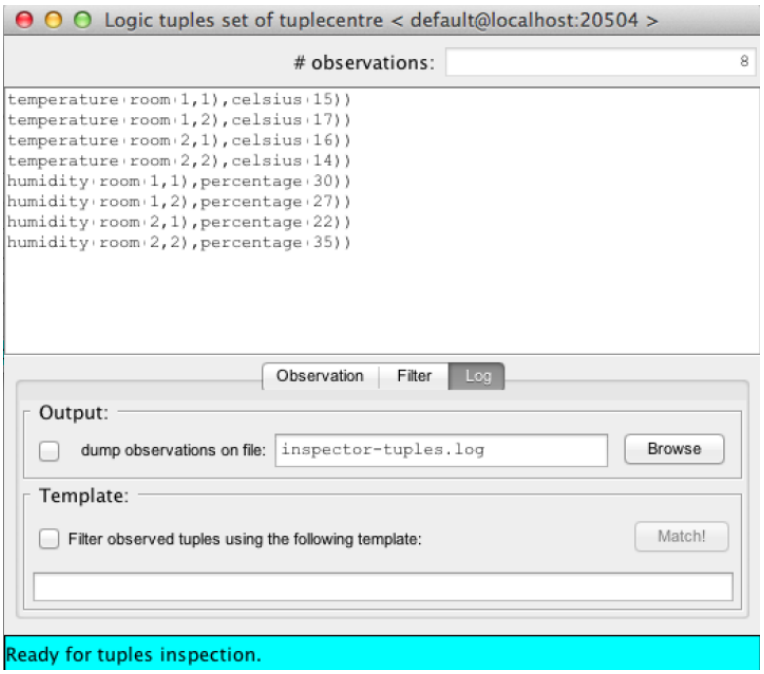

Figura 1.6: Tuple Space View

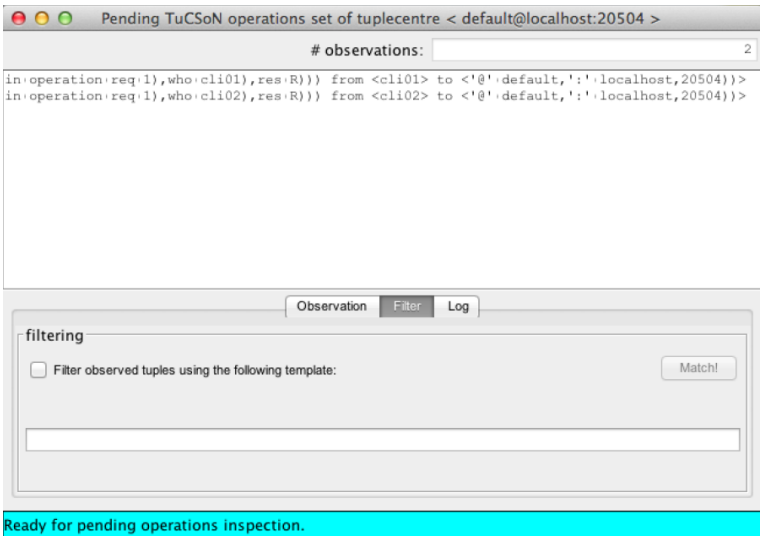

Figura 1.7: Pending Ops View

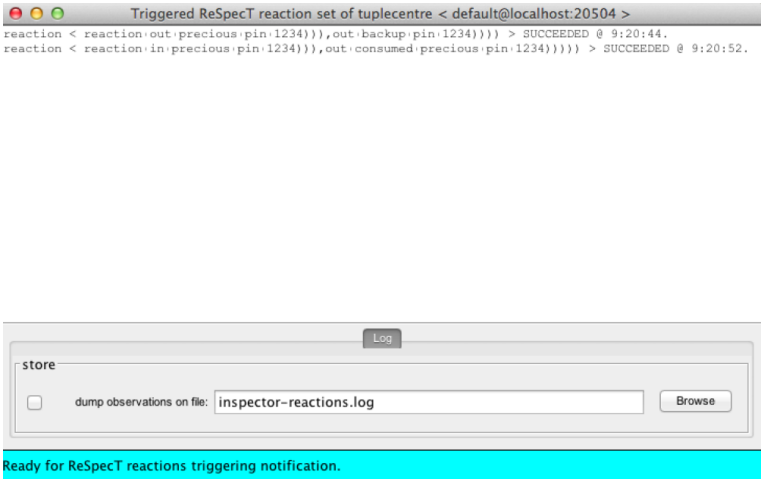

Figura 1.8: ReSpecT Reaction View

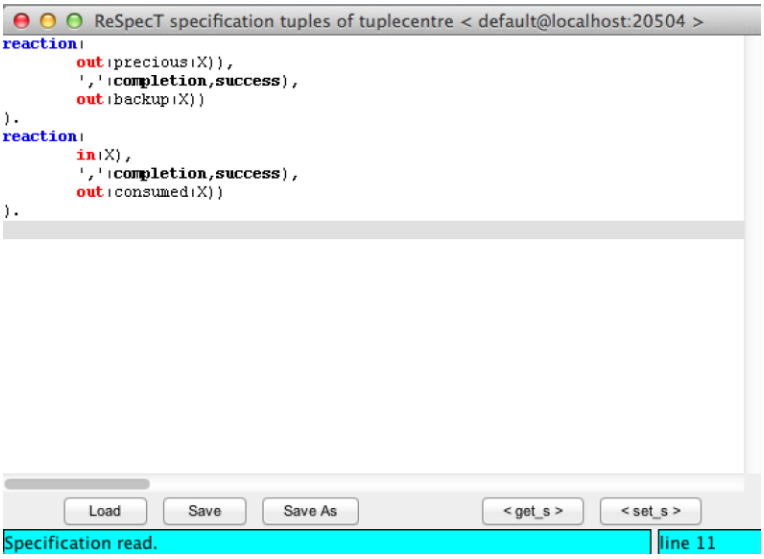

Figura 1.9: Specification Space View

Capitolo 2

Android

In questo capitolo introdurremo Android, la tecnologia che ci fornirà gli strumenti necessari per la creazione di un framework equivalente all'Inspector ma su un dispositivo mobile. Questo capitolo non vuole essere di certo una guida alla programmazione Android, ma si cercherà comunque di introdurre una panoramica sui concetti che potrebbero ritornare utili al nostro scopo.

2.1 Che Cos'è Android

La rivoluzione informatica che stiamo vivendo in questo periodo ha senza dubbio tra i sui protagonisti principali il mondo mobile e, in particolare, *Android*. La maggior parte delle cose che un tempo era possibile fare solamente attraverso il pc, ora è possibile farle in qualsiasi momento e in qualsiasi luogo attraverso l'utilizzo di un cellulare o tablet. I vari servizi e applicazioni accessibili da questi dispositivi sono, infatti, in continuo aumento; rappresentano di fatto il punto di forza che promuove il successo di questa tecnologia. Nasce un nuovo modo di pensare e di concepire i servizi al fine di migliorare la loro accessibilità e compatibilità al fenomeno mobile che rappresenta, oramai, la strada da percorrere per il futuro. Le principali caratteristiche di questi dispositivi sono la mobilità, le dimensioni ridotte e le prestazioni limitate (ormai neanche più di tanto). In questo contesto i principali costruttori di cellulari iniziano a fornire i propri tools e ambienti di sviluppo anche molto eterogenei tra loro. Android mette a disposizione dello

sviluppatore un set di strumenti che vanno dal sistema operativo a delle librerie, con il fine di realizzare le proprie applicazioni mobili. Ciò che distingue Android dalla maggior parte delle alternative oggi esistenti è quello di essere *Open Source*. Questo significa che il codice di Android è completamente consultabile e puo' essere modificato da chiunque voglia contribuire a migliorarlo. Questo ha sicuramente portato un vantaggio sulle altre tecnologie, come confermano le statistiche degli ultimi anni, ma siamo ancora lontani da una possibile standardizzazione. La librerie messe a disposizione per le nostre applicazioni, tra l'altro, sono le stesse utilizzate per l'implementazione del sistema operativo basato su un kernel *Linux*. Ogni componente di Android, fatta eccezione di alcuni casi particolari, può essere infatti rimpiazzato con uno dei nostri o personalizzato in base alle proprie esigenze.

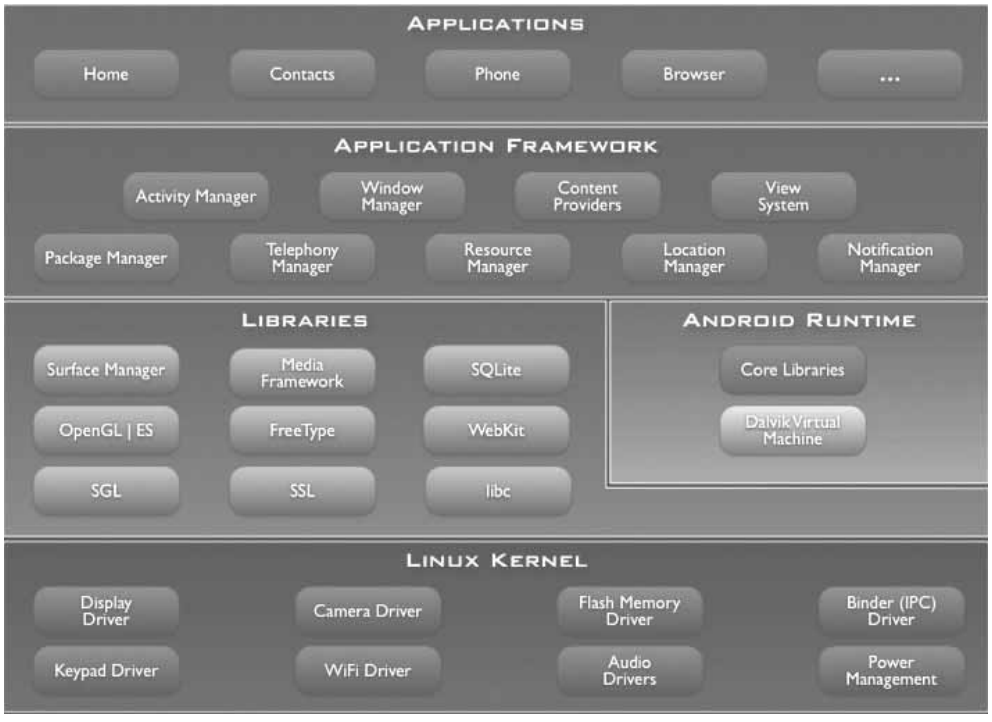

Figura 2.1: Architettura di Android

2.1.1 Java e la DVM

Il linguaggio principale utilizzato per sviluppare in Android è *Java*. Questa scelta ha il grande vantaggio di non dover imparare un nuovo linguaggio di programmazione e facilita, inoltre, il porting di applicazioni inizialmente pensate per un diverso ambiente di esecuzione (come l'inspector nel nostro caso). Adottare Java significava anche, però, adottare la Java Virtual Machine (JVM) che richiede il pagamento di una royalty, in contrasto con la natura open del progetto. Per questo, ma anche per altri motivi, Google decise di adottare una propria VM che prese il nome di *Dalvik Virtual Machine* (DVM). Si tratta di una macchina virtuale basata su quella di Java ma ottimizzata per essere eseguita su dispositivi mobile con risorse limitate. Una virtual machine, quindi, più leggera e veloce con lo scopo di offrire all'utente un'esperienza quanto più fluida nell'interazione con il dispositivo. Tecnicamente parlando, la DVM esegue codice contenuto all'interno di file di estensione *.dex* ottenuti a loro volta, in fase di building, a partire da file *.class* di bytecode Java. Non tutte le classi Java, però, sono disponibili in Android: le librerie *AWT* e *Swing*, infatti, non sono presenti. L'aspetto grafico dell'applicazioni Android, non a caso, è uno degli aspetti fondamentali, se non quello più importante di cui bisognerà tenere conto.

2.2 Activities

L'*Activity* è uno dei componenti principali della nostra applicazione, legata al concetto di schermata. L'obbiettivo dell'activity è quello di fornire una finestra in cui visualizzare tanti componenti (tutti discendenti dalla classe *View*) che nel complesso costituiranno l'interfaccia con la quale l'utente sarà in grado di interagire. I vari componenti saranno organizzati in un layout definito in un file *.xml* in maniera dichiarativa. Solitamente, un'applicazione è costituita da una molteplicità di activities collegate tra loro, dove una di esse è specificata essere la *main activity* ossia la prima schermata che appare quando viene avviata l'applicazione. Quando una nuova activity viene avviata e riceve, quindi, il focus dell'utente, l'activity precedente viene stoppata e inserita all'interno di una struttura chiamata *back stack*. Questa struttura ha lo scopo di preservare l'activity

nel caso in cui l'utente decida di riesumarla ad esempio tramite la pressione del tasto back (meccanismo base di first in/first out).

2.2.1 Ciclo di Vita di una Activity

Un'aspetto fondamentale dell'architettura Android riguarda il fatto che il ciclo di vita di ogni componente è di completa responsabilità dell'ambiente; ciò che, invece, deve essere fatto a seguito di una transizione di stato è di responsabilità dello sviluppatore del componente. Quando avviene una transizione, è il sistema, tramite alcuni *callback* di metodi , che notifica il cambiamento di stato. Ogni callback fornisce l'opportunità di specificare il lavoro da eseguire in quella determinata transizione, attraverso l'overriding del relativo metodo. Gestire il ciclo di vita di una activity è fondamentale per la creazione

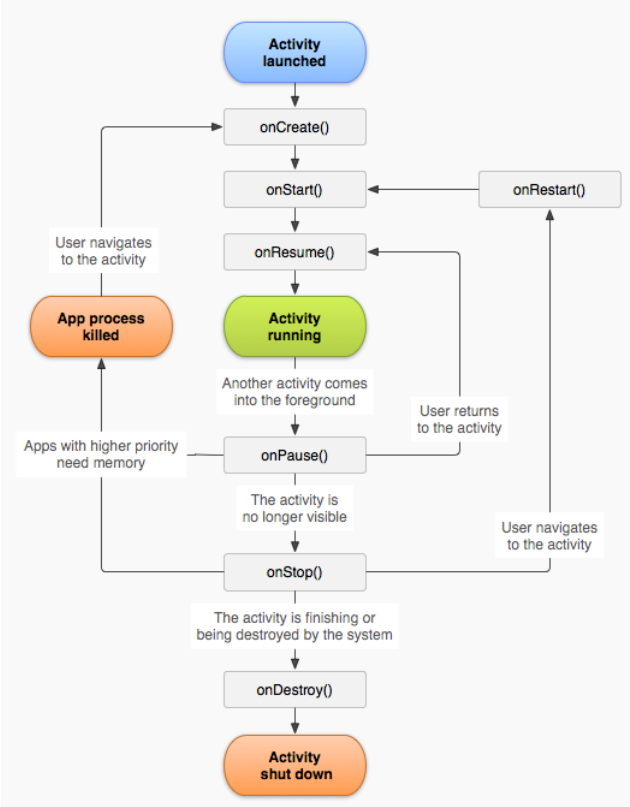

Figura 2.2: Ciclo di vita di una activity

di applicazioni flessibili e con elevato grado di interazione.
I metodi fondamentali sono:

- **onCreate():** primo metodo che viene richiamato, si occupa della creazione dell'activity. In questo metodo bisognerà effettuare le operazioni principali di inizializzazione tra cui quella di impostare il layout dell'activity. Molto importante è anche il parametro di tipo *Bundle* passato in ingresso, la cui funzione è quella di recuperare lo stato precedente alla distruzione dell'activity.

- **onStart():** metodo richiamato quando l'activity sta per diventare visibile all'utente.

- **onResume():** l'attività si prepara ad interagire con l'utente, vengono quindi avviate le eventuali animazioni e la gestione delle risorse esclusive come la fotocamera. Il metodo viene richiamato dopo *onStart()* oppure dopo *onPause()* quando viene recuperato il focus dell'utente. A quest punto l'activity si troverà in cima all'activity stack.

- **onPause():** metodo richiamato quando l'activity perde il focus dell'utente perchè un'altra activity vine messa in foreground. L'activity in pausa rimane, però, parzialmente visibile e assolutamente in vita(tutte le informazioni dello stato vengono mantenute).

- **onStop():** metodo richiamato quando l'attività non è più visibile all'utente. Si tratta di un metodo molto importante dal punto di vista prestazionale in quanto, solitamente, vengono rilasciate tutte quelle risorse che temporaneamente non servono più all'applicazione. Anche in questo caso l'activity rimane in vita ed è possibile riesumarla (dal back stack in cui viene inserita automaticamente) ad esempio premendo il tasto indietro del dispositivo (fare riferimento alla fig:2.2). In questo stato il sistema potrebbe anche decidere di eliminare l'attività per liberare risorse: se l'utente decidesse poi di riesumarla, verrà richiamato il metodo *onCreate()*.

- **onRestart():** metodo chiamato dopo che l'activity è stata stoppata, per riesumarla.

- **onDestroy():** metodo chiamato prima che l'applicazione venga definitivamente distrutta. Può essere chiamato sia a causa dell'invocazione del metodo *finish()* su di essa, sia a causa della volontà del sistema di liberare risorse (è possibile distinguere i due casi attraverso il metodo *isFinishing()*).

Importante sottolineare che, nel ridefinire i metodi sopra elencati, non bisogna dimenticare di invocare il relativo metodo della classe padre attraverso *super()* per eseguire tutta una serie di operazioni fondamentali che riguardano anche il ciclo precedentemente descritto.

2.3 Fragments

Ci sono vari modi per implementare un'interfaccia grafica efficace in Android, uno di questi è utilizzando i *fragments*. I fragments sono uno strumento ideale per la realizzazione di interfacce dinamiche offrendo all'utente una esperienza piacevole e ad un alto livello di interazione. Essi, infatti, scompongono la grafica (e le funzionalità) di un'applicazione all'interno di tanti moduli riusabili. Un fragment rappresenta, quindi, un comportamento o una porzione di interfaccia all'interno di una activity. È possibile aggregare un inseme di fragments diversi all'interno di un'unica activity, così come è possibile utilizzare un fragment all'interno di più activities. Ogni fragments, può essere quindi inserito, rimosso o rimpiazzato attraverso delle *fragment transactions*.
Quando viene incluso all'interno di una activity, il ciclo di vita del fragment è intrinsecamente legato a quello dell'activity stessa. Ad esempio quando un'activity entra in stato di stop, lo diventano anche tutti i fragments in essa contenuti, e quando essa viene distrutta, lo sono anche i fragments. Quando viene eseguita una fragment transaction, è possibile inserirla all'interno di un back stack gestito dall'activity. Ciò che contraddistingue il ciclo di vita di un fragment da quello di un'activity (di per sè, in realtà, molto simili) è il fatto che bisogna precisare se il fragment deve essere inserito all'interno del back stack oppure no. Grazie a questo l'utente è in grado di invertire la transizione premendo il tasto back. Un fragment può essere inserito sia attraverso un approccio dichiarativo, inserendo l'apposito tag allínterno del layout *.xml* dell'activity, sia via codice.

2.3.1 Filosofia di Design

In principio i fragments furono introdotti in Android per permettere la creazione di interfacce dinamiche su dispositivi con display molto grandi come quelli dei tablet (in quanto in grado di visualizzare molti più oggetti rispetto ad un semplice smartphone). La filosofia di progettazione dei fragments permette di implementare con poca fatica diverse visualizzazioni grafiche ognuna adatta a specifici device. In particolare, quando si vuole progettare un'applicazione che funzioni sia per tablet che per smartphone, è possibile cambiare la configurazione dei fragments ottenendo una migliore gestione dello spazio. Ad esempio, un'applicazione per tablet che implementa nella stessa activity due fragments: uno per la visualizzazione di una lista di articoli, uno per visualizzare il dettaglio dell'articolo selezionato nella lista. Entrambi i fragments vengono visualizzati nella stessa activity, ma possiedono comunque il loro proprio ciclo di vita, gestiscono i loro input–events, definiscono il loro layout. La stessa applicazione su uno smartphone, perciò, beneficerà di questo includendo un solo fragment per activity e mantenendo così l'esperienza finale piacevole.

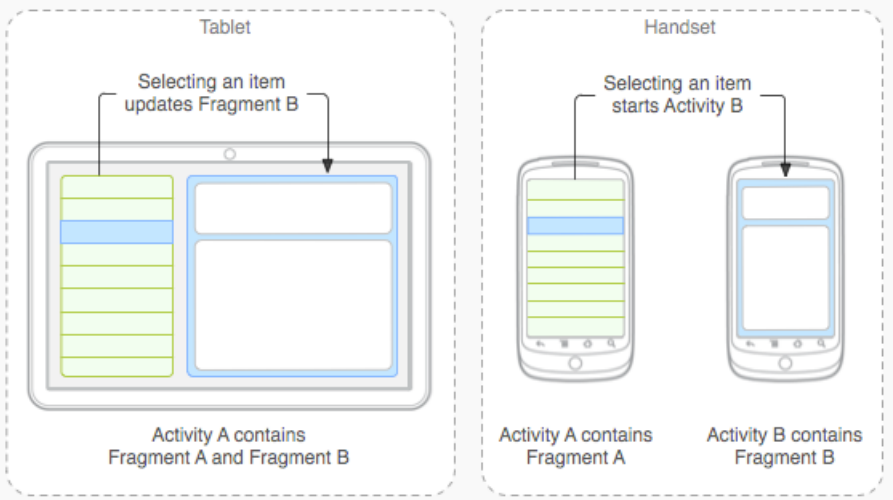

Figura 2.3: Un esempio sulla filosofia di progettazione dei fragments

2.4 Intents

Le componenti principali di un'applicazione sono in grado di comunicare attraverso l'uso di messaggi chiamati *intents*. Essi fanno parte del meccanismo che permettono ai componenti di Android di utilizzare risorse note solo a *run–time*.
Tramite gli intents è possibile ad esempio

- avviare una nuova activity o far eseguire qualcosa di nuovo ad una activity già esistente tramite i metodi *startActivity()* o *startActivityForResult*.
- avviare un nuovo service (di cui parleremo più avanti) o inviare nuove istruzioni ad un service già in esecuzione tramite il metodo *startService()*.

Un oggetto intent è, prima di tutto, una struttura dati contenente un insieme di informazioni destinate al componente a cui è rivolto l'intent (ad esempio il file da aprire) più informazioni che saranno utilizzate dal sistema (come la categoria a cui appartiene il componente target).
Alcune delle informazioni associabili ad un intent sono:

- **Component name:** Si tratta del nome (completo di package) del componente che dovrà gestire l'intent (ad esempio *sd.inspector.activities.tspace*). È un'informazione facoltativa: se non presente, infatti, il sistema recupererà il componente target attraverso altri meccanismi. Si può settare attraverso i metodi *setComponent()*, *setClass()*, o *setClassName()*.
- **Action:** identifica il tipo di azione generica associata all'intent. Sono definite una serie di costanti di base ma è possibile definirne delle proprie.
- **Data:** L'URI e il MIME tipe del dato da associare all'intent. Oltre ad indicare dove si trova il dato, infatti, è importante specificare il tipo di dato che coinvolgerà l'azione associata all'intent. Si possono settare entrambi attraverso il metodo *setDataAndType()*.
- **Category:** stringa che contiene informazioni aggiuntive sul tipo di componente che dovrà gestire l'intent. Settabile attraverso il metodo *addCategory()*.

- **Extras:** coppie chiave–valore di informazioni aggiuntive che si vogliono comunicare al component target attraverso l'intent. Sono settabili e recuperabili attraverso i metodi *putExtra()* e *getExtra()*.

Gli intents possono così dividersi:

- **Espliciti:** quando viene specificato il component name: si è a conoscenza, quindi, a design–time del componente che dovra ricevere l'intent. Solitamente vengono utilizzati per scambiare messaggi con i componenti interni di un'applicazione.

- **Impliciti:** in assenza di un component name target, il sistema dovra preoccuparsi di selezionare il componente migliore. Per far questo il contenuto dell'intent viene analizzato e confortato con gli intent filtres, strutture associate ai componenti che potenzialmente possono gestire gli intent. Vengono spesso utilizzati per comunicare con componenti esterni all'applicazione, di cui a run–time non se nè è a conoscenza.

2.5 La gestione dei Processi

In Android ciascun'applicazione viene eseguita all'interno di un proprio processo Linux. A seconda delle necessità, abbiamo visto come il sistema potrebbe decidere la terminazione di alcuni di questi con il fine di liberare memoria per favorire l'esecuzione dei processi visibili ovvero in *foreground* . La logica adottata da Android non è casuale ma segue delle linee importanti atte a favorire le prestazioni sia dell'applicazione che del dispositivo.
A tale proposito le tipologie dei processi sono state classificate in:

- **foreground process**

- **visible process**

- **service process**

- **background process**

- **empty process**

Il foreground process è il processo che si occupa di eseguire l'applicazione che in quel momento si trova in cima allo stack. Si occupa, perciò, dell'aspetto fondamentale di esecuzione dei componenti d'interazione con l'utente. I foreground process sono i processi con la priorità più alta e vengono eliminati solo quando il sistema non dispone o non può più recuperare la memoria sufficiente per la loro esecuzione.
Una volta che l'attività non risulta più completamente visibile nello schermo, ed entra quindi nello stato *PAUSED*, il processo viene classificato come un visible process. I visible process hanno una priorità inferiore a quelli foreground in quanto non interagiscono con l'utente, ma rimangono comunque molto importanti perchè ancora parzialmente visibili. Essi vengono, quindi, eliminati solo in situazioni estreme, anche se comunque prima di quelli foreground.
I service process, a differenza delle prime due tipologie, non si occupano di gestire gli aspetti di visualizzazione. Questi processi (che vedremo più nel dettaglio nel prossimo paragrafo) non possiedono interfaccia grafica ma hanno una priorità molto alta pari a quelli delle activity in stato *RUNNING*. Ad esempio, sono i processi che si occupano della riproduzione di un MP3 mentre un utente sta facendo altro.
Quando un activity entra in stato di *STOP*, il suo processo viene classificato come un background process. I processi di questa categoria sono solitamente quelli più numerosi; anche per questo motivo essi vengono ordinati secondo la logica *LRU (Last Recently Used)*. La priorità di questi processi è chiaramente più bassa di quelli descritti in precedenza.
Infine i empty process sono quei processi che non rientrano in nessuna delle categorie qui descritte e , quindi, sono i primi candidati all'eliminazione.
È responsabilità dello sviluppatore, quindi, implementare i propri processi in maniera corretta, facendoli rientrare nella giusta categoria desiderata.

2.6 Services

Il service è un componente senza interfaccia grafica, solitamente usato per eseguire operazioni di lunga durata in background. La caratteristica principale è che una volta avviato può rimanere attivo anche quando l'utente cambia activity o esce dall'applicazione.

In poche parole, dopo essere stato avviato il service viene eseguito indipendentemente dal componente che lo ha fatto partire, esegue le sue operazioni in background fino a che non termina spontaneamente o un altro componente decide di fermarlo. Come abbiamo visto un service puo' essere avviato tramite l'utilizzo di un intent e in particolare tramite il metodo *startService()*. Un tipo particolare di service viene chiamato *bound service*: si tratta di un servizio già in esecuzione al quale è possibile collegarsi attraverso il metodo *bindService()*. Bisogna ricordare, però, che una volta creato, il service viene eseguito sul thread principale dell'applicazione: se si vogliono compiere operazioni lunghe che potrebbero bloccare per molto tempo il thread principale, quindi, è necessario creare dei thread separati insieme al service (altrimenti l'applicazione potrebbe bloccarsi e non rispondere più).

2.6.1 Ciclo di Vita di un Service

Il ciclo di vita di un service è, in realtà, più semplice di quello di un'activity. Bisogna, però, fare attenzione a quando il service viene creato e quando distrutto, in quanto esso potrebbe continuare ad essere in esecuzione anche una volta usciti dall'applicazione, senza che l'utente se ne sia effettivamente accorto. Come già ricordato, per avviare un service è necessario creare il relativo intent e chiamare il metodo *startService()*. Ogni volta che viene eseguito questo metodo, il service richiama a sua volta *onStartCommand()* al quale viene passato l'intent chiamante da cui sarà possibile recuperare le varie informazioni. Terminato questo ultimo metodo, il service sarà in esecuzione in background fino a chè non esegua l'istruzione *stopSelf()* oppure terminerà a causa della chiamata di *stopService()*. È importante ricordarsi di fermare il server una volta finito il lavoro richiesto in modo da risparmiare le risorse di sistema e la batteria del device. Il metodo *onCreate()*, come nel caso delle activity, verrà chiamato solo la prima volta al momento della creazione del service. Per quanto riguarda i bounded services, al posto di *onStartCommand()* troviamo *onBind()*, richiamato ogni qual volta un'applicazione desideri collegarsi al service: il metodo dovrà ritornare un'interfaccia *IBinder* che servirà per comunicare con il servizio. Se il server si ritrova senza componenti ad esso collegati, il sistema avvierà la procedura di terminazione. Abbiamo già specificato che se il service è associato ad una applicazione che possiede il focus dell'utente, difficilmente il sistema deciderà di termi-

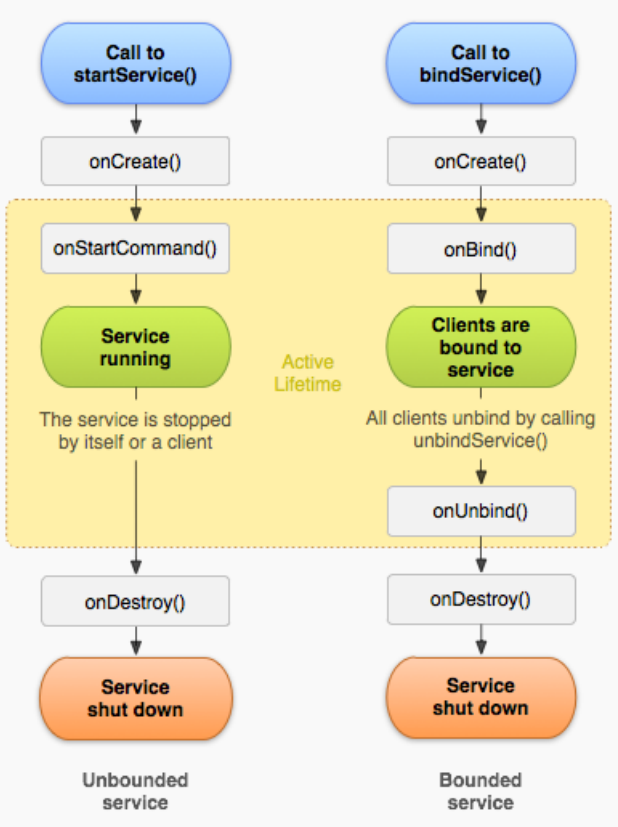

Figura 2.4: Ciclo di vita di un Service

narlo. Se invece il service è in esecuzione da molto tempo il sistema potrebbe decidere la sua terminazione per recuperare memoria sufficiente ad eseguire l'applicazione in quel momento in cima allo stack. Se il sistema termina il servizio, esso ripartirà non appena le risorse ritorneranno disponibili, ma ciò dipende anche dal valore di ritorno settato nel metodo *onStartCommand()*:

- **START_STICKY :** l'ambiente potrà ricreare il servizio invocando nuovamente *onStartCommand()* passando però un intent a null a meno che non ve ne siano di pendenti.
- **START_NOT_STICKY :** Il Service, se non ci sono Intent in attesa, non viene ricreato fino a una esplicita invocazione del metodo *startService()*. Questo garantisce che l intent passato sia sempre diverso da null.

- **START_REDELIVER_INTENT :** In questo caso viene rischedulata una nuova partenza del Service con un reinvio dello stesso Intent.

- **START_STICKY_COMPATIBILITY :** Indica la non garanzia nella chiamata al metodo onStartCommand() nel caso di gestione con START_STICKY.

2.6.2 Service o Thread ?

Quando bisogna utilizzare un service invece che un semplice thread? Il service è un componente che esegue operazioni in background senza che sia necessaria l'interazione con l'utente (anche quando la nostra applicazione è terminata); il service, quindi, dovrà essere utilizzato in questi contesti. Quando c'è la necessità di eseguire del lavoro all'esterno del main thread mentre l'utente sta interagendo con l'applicazione, probabilmente ciò che serve è, invece, un thread.

2.7 Gesture

Attraverso gli schermi touch sensibili al tocco degli utenti (in dotazione a tutti i device che ospitano il sistema operativo Android) è possibile eseguire delle operazioni in risposta a dagli eventi touch. L'idea di base delle gesture, quindi, consiste nel permettere all'utente di eseguire alcune funzioni attraverso specifici movimenti del dito sul display o tracciando dei veri e propri disegni. Questa possibilità offre un tipo di esperienza nuova con il dispositivo: permette di aumentare l'iterazione dell'utente con l'applicazione, nonchè aumenta l'accessibilità delle funzioni da essa offerte. Le API Android permettono non solo di catturare e gestire gli eventi touch, ma anche di realizzarne e memorizzarne delle nuove.
La procedura per includere la funzionalità delle gesture all'interno della propria applicazione è:

- definire la gesture tramite il *Gesture Builder* che si trova tra le applicazioni dell'emulatore Android su Eclipse.

- caricamento delle informazioni di gesture all'interno dell'applicazione.

- riconoscimento delle gesture e relative azioni.

2.8 Il Manifest

Ogni applicazione Android possiede nella sua directory un file *.xml* chiamato AndroidManifest. Si tratta di un file estremamente importante in quanto fornisce al sistema Android una serie di informazioni sull'applicazione, necessarie prima che venga eseguito il codice. Queste informazioni descrivono le caratteristiche principali dell'applicazione, le sue componenti e i permessi che necessita per accedere alle varie risorse del device su cui sarà installata.
Le principali informazioni da dichiarare nel manifest sono:

- nome del Java package dell'applicazione, che verrà utilizzato come identificatore unico dell'applicazione
- le componenti dell'applicazione: quindi i nomi di tutte le activity e dei service, comprese le loro facoltà (ad esempio attraverso gli intent filters, gli intent che sono in grado di gestire). Queste descrizioni permettono al sistema Android di conoscere quali sono i componenti e sotto quali condizioni devono essere avviati.
- permessi per l'accesso a parti private delle API
- permessi per interagire con le componenti dell'applicazione
- il livello minimo delle API che l'applicazione richiede
- lista delle librerie che l'applicazione utilizza

Capitolo 3

Inspector Mobile

Una volta acquisite le nozioni introdotte nel primo capitolo, possiamo utilizzare gli strumenti forniti da Android descritti nel secondo capitolo per raggiungere lo scopo di questa tesi, ossia la creazione di un framework mobile per lo sviluppo di sistemi coordinati. Un'applicazione di questo genere è, per l'appunto, il già citato Inspector che è disponibile, però, per il solo ambiente desktop. Per rendere accessibili le sue funzionalità all'interno del mondo mobile, quindi, è necessario eseguire il *porting* dell'applicazione.

3.1 Porting

Il termine *portabilità* può assumere diversi significati a seconda del contesto in cui esso viene utilizzato. In informatica, la portabilità di un componente software è un cambiamento o modifica del componente al fine di consentire l'utilizzo di quest'ultimo in un ambiente diverso da quello originale. In generale, infatti, un componente software inizialmente progettato per un determinato ambiente non funziona se ne viene cambiato il contesto di esecuzione. Basti pensare alle differenze tecnologiche come ad esempio l'utilizzo di CPU diverse, interfacce dei sistemi operativi diverse, hardware diversi o ad un linguaggio di programmazione non supportato.
Nello scenario odierno, in effetti, la portabilità sta assumendo un ruolo sempre più centrale nella scelta delle tecnologie per lo sviluppo del software; ciò è dovuto anche alla diffusione sul mercato di particolari tecnologie (vedi smartphone).

Un componente software viene quindi definito portabile, se eseguirne il porting è semplice e poco costoso; ciò si traduce in un particolare caso di riusabilità del software (il caso limite è quello in cui il componente non necessita di alcuna modifica). Uno dei fattori che più incide sulla portabilità è senza dubbio il linguaggio di programmazione usato. Se l'ambiente su cui effettuare il porting non supporta alcun interprete o compilatore di quel linguaggio, ecco che il porting diventa un'operazione molto complicata e costosa. Per questo motivo alcuni linguaggi non posso essere definiti portabili (ovvero non consentono la scrittura di programmi portabili) anche per il solo fatto che i compilatori e interpreti possiedono sottili differenze semantiche e sintattiche che compromettono il parziale o totale funzionamento del programma.

Il programma Inspector è implementato utilizzando il linguaggio Java che fa parte dei linguaggi definiti interpretati ossia che necessitano di un interprete. Un programma Java può essere eseguito in un qualunque ambiente in cui sia installata la Java Virtual Machine (prelevabile dal sito ufficiale della Oracle). Non è stata perciò casuale la scelta di Android come suite di strumenti per eseguire il porting su un dispositivo mobile. Android, infatti, supporta il linguaggio di programmazione Java e utilizza una macchina virtuale (la Dalvik Virtual Machine) che si basa proprio su quella distribuita da Oracle. Grazie a questa scelta, il porting dell'applicazione Inspector risulta essere, quindi, un'operazione fattibile. Bisogna tenere conto però che:

- non tutte le classi Java sono supportate dall'ambiente Android (ad esempio AWT e SWING).
- la filosofia di progettazione all'interno di un ambiente come quello mobile può presentare differenze abbastanza evidenti dalla progettazione di applicazioni desktop, come ad esempio la priorità che deve essere riservata agli aspetti prestazionali legati all'utilizzo di risorse del dispositivo, tra cui su tutte il consumo della batteria.

3.2 Struttura generale

Per realizzare l'applicazione bisogna utilizzare gli strumenti forniti da Android introdotti nel capitolo precedente e tenere conto delle inevitabili differenze derivabili dal loro uso.
Per prima cosa ci si rende conto che la mancanza delle librerie grafiche Java in Android presuppone un approccio totalmente diverso, anche solo per il fatto che la parte grafica svolge un ruolo fondamentale nelle applicazioni mobile. L'Inspector mobile sarà, quindi, composto da una serie di activity (finestre) ognuna delle quali avrà un proprio layout *.xml* (in realtà è possibile definire più di un layout in base al tipo di device e al suo orientamento). All'interno del layout verranno definiti i vari componenti grafici e la loro disposizione al fine di realizzare l'interfaccia utente. I vari componenti, tutti discendenti dalla classe *View*, possono essere caselle editabili, pulsanti, etichette, contenitori per altri componenti; per tutti questi è possibile definire i vari attributi oppure sostituirli con le proprie versioni custom.

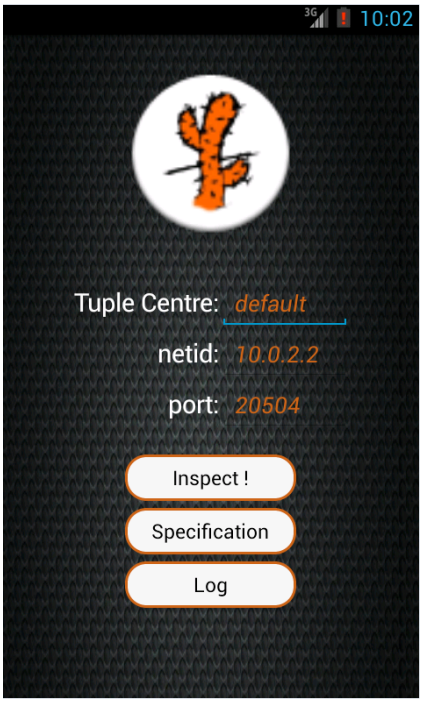

Figura 3.1: SelectNode per Smartphone

L'activity chiamata *SelectNode* è stata indicata (all'interno del manifest) come prima activity a dover essere istanziata nel momento dell'avvio dell'applicazione; essa permette all'utente, similmente a quella dell'inspector originale, di inserire il nome completo del centro di tuple da ispezionare. Tramite codice Java è possibile recuperare queste informazioni e renderle eventualmente disponibili alle altre componenti dell'applicazione (ad esempio un'altra activity o ad un service) tramite l'utilizzo degli intent.
Come abbiamo visto, infatti, gli intent sono come dei messaggi all'interno dei quali è possibile inserire e recuperare dati aggiuntivi attraverso i metodi *putExtra()* e *getExtra()*. Premendo il tasto *Inspect!* , infatti, viene lanciata una seconda activity chiamata *MainActivity* nella quale verranno visualizzate le informazioni del centro di tuple indicato dall'utente.
Per fare questo, nel metodo *onCreate()* viene creato l'intent per avviare il service che, come verrà descritto più avanti, avrà il compito di recuperare le informazioni necessarie. Le informazioni che dovrà visualizzare la *MainActivity* sono le stesse che venivano visualizzate nelle finestre dell'Inspector originale tramite i pulsanti *tuple space, pending ops* e *ReSpecT reactions*.
Al fine di ottimizzare lo spazio offerto dal display dei dispositivi ed offrire un'esperienza interattiva soddisfacente che permetta di passare dalla visualizzazione di un'informazione ad un altra, si è scelto di implementare l'interfaccia grafica della *MainActivity* attraverso l'utilizzo dei fragments.
Ogni fragment è implementato con un layout scrollabile che permette di visualizzare il corrispettivo contenuto anche nel caso in cui questo superi la quantità visualizzabile sul display. L'interfaccia grafica della *MainActivity* comprende una barra pulsanti che, attraverso l'esecuzione di una transaction (e di una piccola animazione), permette di sostituire il fragment in quel momento visualizzato con il fragment desiderato. Le stesse transaction sono effettuabili, anzichè dalla barra pulsanti, eseguendo una semplice gesture che simula la transizione laterale dei fragments.
A differenza dell'Inspector originale, è possibile accedere direttamente allo *specification space* attraverso la prima activity dell'applicazione. Essa, infatti, offrendo funzionalità di tipo reattive, cioè in risposta alle richieste dell'utente, non necessita la notifica in tempo

reale di un evento del centro di tuple; è possibile usufruirne, perciò , senza il bisogno di attivare un service come avviene con la *MainActivity*.
Nella activity *SelectNode*, inoltre, è presente il pulsante *Log* che permette di accedere ad una lista di file (di cui parleremo più avanti) contenenti le informazioni sui centri di tuple analizzati con l'applicazione, aggiornati all'ultima volta che si sono ispezionati. Premendo su uno di questi file, viene lanciato un intent implicito con lo scopo di aprire un'applicazione, eventualmente già installata sul dispositivo, in grado di visualizzarne il contenuto.

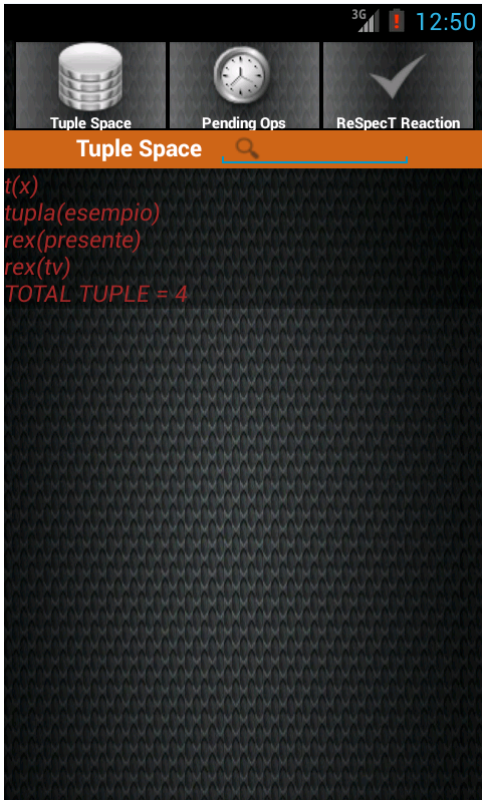

Figura 3.2: MainActivity per Smartphone

3.3 L'InspectorService

Una delle prime cose di cui bisogna tenere conto in Android, è la questione riguardante il main thread.
Il main thread è il thread principale dell'applicazione che in quel determinato momento possiede il focus dell'utente. Tra i suoi compiti principali, oltre a quello di gestire le componenti dell'applicazione, c'è quello dell'aggiornamento delle componenti grafiche (è l'unico thread che può fare questo).
Risulta evidente, quindi, che al fine di garantire un'esperienza interattiva ottimale, è conveniente non appesantire il main thread con l'esecuzione di operazioni lunghe e pensanti. Bloccare per molto tempo il main thread, infatti, potrebbe causare l'errore *Application Not Responding* (ANR) con la conseguente uscita dall'applicazione. Questo si traduce nel fatto che la gestione degli eventi e delle informazioni che saranno poi eventualmente visualizzate nel display devono avvenire in thread separati, in particolare gestiti attraverso l'utilizzo di un service.
Per questo si è scelto di implementare un service chiamato *InspectorService*, che sia in grado di eseguire quelle operazioni necessarie al funzionamento dell'Inspector senza bloccare il main thread. L'inspector necessita, infatti, di operazioni di configurazione per essere in grado di svolgere le sue funzionalità.
Le operazioni di configurazione, consistono in particolare nella creazione di:

- **TucsonAgentId:** identifica l'agente TuCSoN che si occupa di ispezionare un determinato centro di tuple.

- **TucsonTupleCentreId:** identifica il centro di tuple (e quindi anche il nodo e il numero di porta) che si vuole ispezionare.

Questi due oggetti verranno passati ad un terzo oggetto chiamato *InspectorContextStub* che si preoccuperà di inviare tutte le informazioni al TuCSoN Service attivo sul nodo da ispezionare. Tutti e tre sono già realizzati in linguaggio java e sono quindi facilmente importabili, così come sono, all'interno del progetto Android.
Ricordiamo che è fondamentale creare un ulteriore thread all'interno dell'implementazione del service, altrimenti esso verrà eseguito sempre a carico del main thread; dato

che è necessario bloccarsi in attesa di nuovi eventi dal TuCSoN Service, ciò causerebbe inevitabilmente un ANR e, quindi, l'uscita dall'applicazione.
La comunicazione tra il TuCSoN service e la versione desktop dell'Inspector viene gestita attraverso un protocollo. Esso permette di attivare varie opzioni come la possibilità di ispezionare in maniera reattiva oppure la volontà di osservare cambiamenti nello spazio di tuple. Nel caso della versione mobile si è scelto direttamente di utilizzare un protocollo che permette l'analisi di tutte le caratteristiche del centro di tuple secondo la modalità proattiva.
Il service dovrà, quindi, implementare un thread che si preoccuperà di:

- creare l'*InspectorContextStub*.
- definire il protocollo di comunicazione.
- implementare l'interfaccia *InspectorContextListener* e aggiungersi, quindi, come ascoltatore.
- rimanere in attesa di eventi notificati dal TuCSoNService.
- eseguire le operazioni necessarie in base al tipo di evento ricevuto.

Gli eventi vengono notificati a seconda del verificarsi o meno di cambiamenti nei centri di tuple ispezionati (ad esempio se viene aggiunta una nuova tupla o viene innestata una reaction, i vari thread saranno informati immediatamente).

3.4 Multi–Threading

Il mercato dei dispositivi mobili come tablet o smartphone sta diventando sempre più esigente in termini di qualità e prestazioni; tutti i dispositivi, oramai, sono costituiti da processori multi–core dalle prestazioni molto vicine a quelle di un personal computer. Anche Android supporta lo sviluppo di applicazioni multi–threading in grado di sfruttare appieno le potenzialità dei nuovi device.
Grazie all'Inspector, in ambiente desktop, si ha la possibilità di ispezionare più centri di tuple contemporaneamente semplicemente avviando più istanze dell'applicazione. Come già detto, anche Android permette l'esecuzione di programmi multi–treading ma, a causa delle dimensioni limitate del display, è possibile visualizzare solamente una schermata alla volta. La schermata non più visualizzabile viene quindi stoppata. Una volta avviata un'applicazione, inoltre, non è possibile avviarla una seconda volta in modo da avere due diverse istanze (viene semplicemente riesumata l'applicazione precedentemente stoppata).
Come possiamo ottenere questa funzionalità anche nell'Inspector mobile? Ancora una volta la scelta di implementare il service viene a nostro favore. Grazie ad esso infatti possiamo, ogni volta che l'utente ne ha la necessità, creare un nuovo thread responsabile di ispezionare il nuovo centro di tuple.
Quando l'utente decide di ispezionare un nuovo nodo, viene lanciato un nuovo intent per il service attraverso il metodo *startService()*. Se il service era già in esecuzione, richiamerà direttamente il metodo *onStartCommand()* all'interno del quale verificherà se effettivamente si vuole ispezionare un nuovo centro di tuple e, se necessario, creare un nuovo thread. Viene, quindi, tenuta traccia di tutti i thread creati e dei centri di tuple ispezionati in modo da:

- eliminare tutti i tread una volta che si decida di terminare il service.
- evitare di creare nuovi thread per nodi che stiamo già ispezionando e crearne dei nuovi per quelli ancora da ispezionare.

Le strutture dati utilizzate per memorizzare i thread e i centri di tuple, sono degli *ArrayList* che vengono creati nel metodo *onCreate* del service (metodo che viene richiamato solo la prima volta al momento della creazione del service).

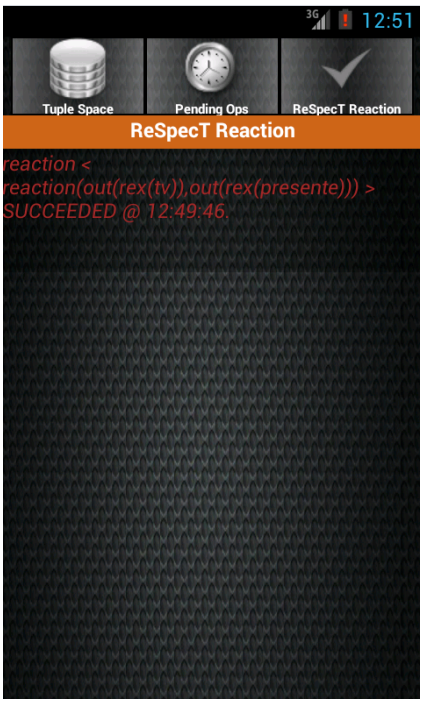

Figura 3.3: Reaction Space per Smartphone

3.5 Il Thread Inspector

Il thread Inspector è il thread che si occupa di rimanere in ascolto di eventuali eventi relativi al centro di tuple che si sta ispezionando. Esso, come già sottolineato, implementa l'interfaccia *InspectorContextListener* e quindi il metodo *onContextEvent()*. All'interno di quest'ultimo vengono eseguite le operazioni necessarie in base al tipo di evento ricevuto.

A differenza di quanto succede nell'Inspector versione desktop, il thread Inspector non può direttamente settare, nella grafica dell'applicazione, le informazioni contenute nell'evento ricevuto (ad esempio l'insieme delle tuple presenti nel centro di tuple). Questo perchè la responsabilità di aggiornare gli aspetti visivi dell'activity è unicamente del main thread. Essendo poi presenti, in generale, più thread attivi nello stesso momento (uno per ogni centro di tuple ispezionato), sarebbe opportuno salvare da qualche parte le informazioni da essi raccolte in modo che se l'utente decida di mettere in background l'applicazione, esse vengono comunque registrate per poi essere visualizzate nel momento

in cui l'applicazione ritorna ad essere visibile.
Quello che fà l'Inspector, quindi, è prima di tutto verificare la presenza della *sdcard* nel dispositivo al fine di creare, in una cartella dedicata all'applicazione, dei file su cui memorizzare le informazioni ricevute. I file sono nominati specificando il nome del centro di tuple, nodo, numero di porta e un tag che indica l'informazione da esso contenuta.
Le informazioni possono essere:

- l'insieme delle tuple.
- l'insieme delle operazioni sospese.
- l'insieme delle reaction innestate da quando si ispeziona il nodo.

In pratica saranno salvati sul dispositivo tre file diversi per ogni centro di tuple analizzato; questi file conterranno le informazioni aggiornate all'ultima ispezione.

3.6 Il Main Thread

Come più volte ricordato, il main thread è l'unico responsabile dell'aggiornamento dei componenti grafici dell'activity visualizzata. Per aggiornare il contenuto dei fragments, in particolare, un *timerTask* avverte circa ogni secondo il main thread di aggiornare il contenuto dell'unico fragment visibile in quel momento, andando a leggere il relativo file salvato nella *sd card*.
Ovviamente se si cercasse di settare il contenuto di un fragment al momento non inserito nell'activity, si avrebbe il lancio di un'eccezione; è possibile rendersi conto di questo richiamando il metodo *isVisible()*.
Come fa il *timerTask* ad interagire con il main thread? Questo è possibile attraverso l'utilizzo della classe *Handler* che offre un meccanismo per inviare messaggi al main thread. I messaggi verranno inseriti all'interno di una coda e processati ad uno a uno.

3.7 Prestazioni

Le prestazioni di una applicazione Android influenza notevolmente le prestazioni dell'intero dispositivo: è fondamentale, infatti, assicurarsi di utilizzare le varie risorse solo

quando strettamente necessario e , cosa ancora più importante, ricordarsi di rilasciarle quando non servono più. Una corretta progettazione risulterà indispensabile per ottenere una buona applicazione.
L'implementazione dei metodi di feedback (descritti nel capitolo su Android) permette di avere il controllo su queste risorse in base allo stato in cui si trova l'activity o il service. Il service è uno di quei componenti che impegna di più le capacità computazionali del dispositivo, in quanto è solitamente destinato ad ospitare le operazioni di calcolo più lunghe; per questo motivo è importante stabilire quando avviarlo e quando terminarlo. L'*InspectorService*, in particolare, viene lanciato quando l'utente indica l'intenzione di volere ispezionare un nodo, cioè dopo aver premuto il tasto *Inspect!* e quindi durante la creazione della *MainActivity*. Il service viene così mantenuto in vita per tutto il tempo fino a che l'utente decida di uscire definitivamente dall'applicazione. L'utente, infatti, potrebbe decidere di tornare indietro fino alla *SelectActivity* per ispezionare altri nodi, oppure potrebbe decidere di fare temporaneamente altro mettendo, così, l'applicazione in background. Per tutto questo tempo, il service deve chiaramente rimanere attivo e registrare tutti i cambiamenti dei nodi che si sta ispezionando. All'interno del metodo *onDestroy()*, l'InspectorService termina tutti i thread da lui creati che a loro volta avviseranno il relativo TuCSoN Service della loro terminazione.
La *MainActivity*, invece, utilizza un *TimerTask* che ricorda al main thread di aggiornare la grafica; ovviamente, nel caso l'activity venga stoppata, l'utilizzo del *TimerTask* diventa inutile e va quindi terminato all'interno del metodo *onStop()*. Una volta riesumata l'activity, attraverso *onRestart()* è possibile ricreare il *TimerTask* e fallo ripartire garantendo così il normale funzionamento.

3.8 Orientamento e Dimensioni del Device

Altro aspetto sicuramente da non sottovalutare in ambiente Android è la dimensione del device e ,in particolare, quella del display. La quantità di elementi grafici inseribili, infatti, è direttamente proporzionale alle dimensioni dell'area visualizzabile.
Le differenze fra i device vengono accentuate nel caso in cui l'applicazione è destinata ad includere sia il pubblico degli smartphone che quello dei tablet. Le dimensioni del

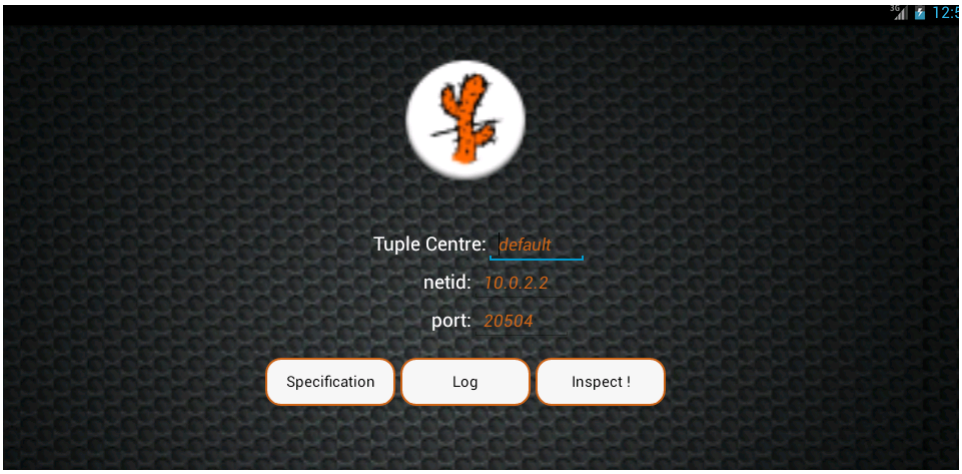

Figura 3.4: SelectNode per tablet

display dei tablet sono chiaramente maggiori e spesso garantiscono una risoluzione superiore rispetto a quelli degli smartphone.
Se tutto ciò non bastasse, i vari dispositivi, ruotando, possono far cambiare automaticamente l'orientamento dell'interfaccia grafica (sempre se l'applicazione visualizzata è stata abilitata a farlo). Anche questo potrebbe essere causa di malfunzionamenti nell'interfaccia: alcuni componenti grafici potrebbero essere visualizzati solo parzialmente.
L'Inspector, perciò , sfrutta la definizione di più layout per adattare al meglio l'interfaccia grafica ai vari tipi display e orientazioni. Per quanto riguarda l'utilizzo su tablet, le dimensioni di molti componenti sono state modificate rispetto alle relative su smartphone, con il fine di sfruttare il display più grande, la risoluzione maggiore e permetterne un acceso facilitato da parte dell'utente. La disposizione di alcuni componenti viene modificata in particolare nel momento in cui viene cambiato l'orientamento del device: quando si esegue l'applicazione in portrait si sfrutta il maggior spazio verticale, quando la si esegue in landscape si sfrutta il maggior spazio orizzontale.
Con la rotazione del device, in realtà, non viene semplicemente cambiato il layout dell'activity: essa, infatti, viene prima distrutta e poi completamente ricreata con il nuovo layout. L'utente, chiaramente, non deve avere percezione di quanto accade se non per il fatto che l'interfaccia è ruotata. Deve quindi poter tornare ad interagire con un'activity che si trovi nello stesso stato in cui si trovava prima della rotazione.

Per fare questo la *MainActivity* implementa il metodo *onSaveIstanceState()* all'interno del quale salva, nella struttura dati *Bundle*, il fragment in quel momento visualizzato. Il *Bundle* sarà poi recuperabile all'interno del metodo *onCreate()* dove verrà ripristinata l'interfaccia precedente.

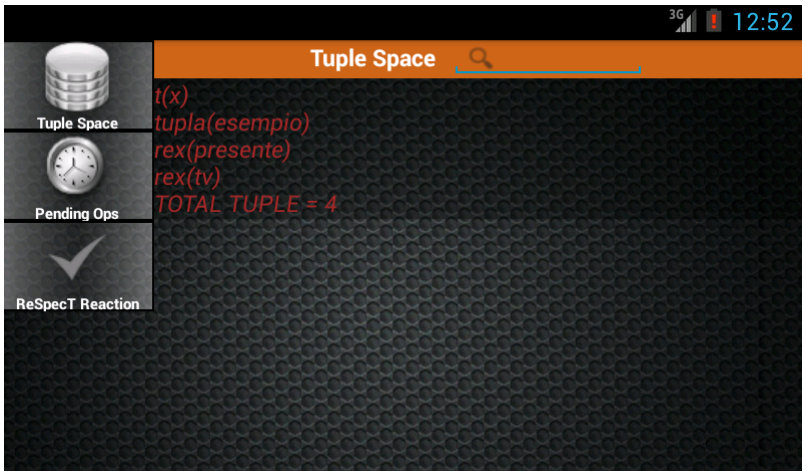

Figura 3.5: MainActivity in landscape mode

Conclusioni

In questa tesi è stata sviluppata una prima implementazione della versione mobile dell'applicazione Inspector all'interno del progetto TuCSoN. Grazie ad essa, ora, gli utenti del mondo mobile possono accedere, attraverso il proprio cellulare o tablet, ad alcune delle funzionalità del modello di coordinazione: ad esempio è possibile analizzare il contenuto sia dello spazio di tuple ordinario, sia dello specification space di tutti i centri di tuple sparsi nella rete, oppure è possibile modificare la specifica stessa e salvarla in un file all'interno del dispositivo. Questo compito è stato facilitato dall'adozione dell'ambiente di sviluppo Android che, oltre a supportare lo stesso linguaggio di programmazione (Java) dell'applicazione originale, ha fornito l'insieme degli strumenti che hanno permesso il porting dell'applicazione.
La realizzazione dell'Inspector mobile ha evidenziato la necessità di un'applicazione di adattarsi ai meccanismi che fanno parte di un ambiente di esecuzione: di questo nè beneficerà sia l'applicazione per essere eseguita correttamente, sia l'ambiente di esecuzione che offrirà al meglio le proprie prestazioni.
Le funzionalità future sono ora collegate agli sviluppi sia del modello di coordinazione TuCSoN sia a quelli del mondo mobile e, in particolare di Android. Ad esempio potrebbe essere utile implementare il supporto ai social network o ad altre applicazioni per la condivisione delle informazioni.

Bibliografia

[1] Omicini A. and Denti E.: From tuple spaces to tuple centres, 2001, *Science of Computer Programming, 41(3), 277–294.*

[2] Rowstron, A.I.T: Bulk Primitives in Linda Run–Time Systems, 1996, *PhD thesis, The University og York.*

[3] Omicini A.: Formal ReSpecT in the A&A perspective. *In Carlos Canal and Mirko Viroli, editors, 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), pages 93–115, CONCUR 2006, Bonn, Germany, 31 August 2006. University of Málaga, Spain. Proceedings.*

[4] John W. Lloyd: Fondation of Logic Programming. *Springer, 1st edition, 1984.*

[5] Paolo Ciancarini: Coordination models and languages as software integrators. *ACM Computing Surveys, 28(2):300–302, June 1996.*

[6] Omicini A.: Coordination–based Systems. *Course of Distribuited System, Academic Year 2011/2012.* http://campus.unibo.it/80694.

[7] Omicina A. and Stefano M.: The TuCSoN coordination model & tecnology: A guide. *TuCSoN v.1.10.2.0205, Guide v.1.0.1, October 4 2012.*

[8] Omicini A.: On the semantics of Tuple–Based Coordination Models- *ACM Symposium on Applied Computing, 1999, 175–182.*

[9] Gelenter D.: Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems, 7(1), 1985, 80–112.*

[10] Massimo C.: Android, guida per lo sviluppatore. *2010, APOGEO srl.*

[11] Android Developers. *API Guides. http://developer.android.com/guide.*