ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria ed Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

SVILUPPO DI SISTEMI MOBILE AD ALTA DENSITÀ BASATI SU COMUNICAZIONI OPPORTUNISTICHE WIRELESS

Elaborata nel corso di: Ingegneria dei Sistemi Software Adattativi Complessi

Tesi di Laurea di:
MATTIA BALDANI

Relatore:
Prof. MIRKO VIROLI

ANNO ACCADEMICO 2014–2015
SESSIONE III

PAROLE CHIAVE

Wifi ad hoc

Android

Akka Stream

Reactive programming

Scala

Alla mia famiglia.

Indice

Introduzione

Negli ultimi anni, grazie allo sviluppo tecnologico dell'elettronica e dell'informatica, si è assistito ad una diffusione sempre maggiore dei cosiddetti mobile devices, come gli smartphone ed i tablet, basati su sistemi operativi come Android.

Questi dispositivi, diventati ormai pervasivi nella vita quotidiana, hanno ormai raggiunto livelli di capacità computazionali e di connettività tali da renderli estremamente interessanti come piattaforma per lo sviluppo di applicazioni distribuite su larga scala.

In questa tesi si parte analizzando l'architettura cloud-based dei sistemi software attualmente largamente utilizzata nel settore dei sistemi mobile, e ci si pone l'obiettivo di sviluppare sistemi distribuiti che possano operare in scenari dove i sistemi cloud incontrano difficoltà, grazie all'uso di tecnologie di comunicazioni opportunistiche wireless peer-to-peer tra dispositivi mobile.

Capitolo 1

Introduzione alle comunicazioni opportunistiche su sistemi mobile

1.1 I limiti dei sistemi mobile cloud-based

Negli ultimi anni si sta assistendo ad una diffusione sempre maggiore di applicazioni che richiedono sistemi software intrinsecamente distribuiti. Se il modo tradizionale di intendere una applicazione software era di un sistema centralizzato, che vive su una sola macchina ed eventualmente scambia informazioni con la rete, oggi ci si sta dirigendo sempre più nella direzione di intendere il singolo nodo computazionale come un semplice componente di un sistema distribuito più grande, nel quale la rete assume quindi un ruolo fondamentale.

Tra le applicazioni distribuite attualmente più utilizzate, troviamo certamente le applicazioni Web, od altre tipologie di applicazioni native che si collegano ad un backend remoto, come nel caso della maggior parte delle applicazioni mobile Android ed iOS. L'architettura dei sistemi sui quali si basano queste applicazioni, è quasi sempre del tipo client-server, dove il software di frontend eseguito nel device di proprietà dell'utente si occupa quasi esclusivamente di gestire l'interfaccia grafica, mentre il software di backend che si occupa di immagazzinare ed elaborare la maggior parte dei dati, viene eseguito su dei server remoti in un data center. L'architettura client-server appena descritta, è uno dei fondamenti del modello del cloud

computing, sul quale si basano tantissimi servizi molto popolari. Per certe tipologie di applicazioni, una architettura basata sui concetti del cloud computing è sicuramente adatta, e permette di rendere il sistema disponibile in modo scalabile ad un numero enorme di persone, quantificabile addirittura con buona parte della popolazione mondiale nei casi di servizi come Google o Facebook.

I sistemi basati sul cloud computing hanno tuttavia la forte limitazione di essere dipendenti dalla disponibilità di una rete che permetta la comunicazione tra client e server, che sono in generale molto distanti tra di loro dal punto di vista geografico. Anche se la rete Internet sta ormai raggiungendo performance molto elevate, sopratutto nei backbone e nei sistemi che possono essere raggiunti direttamente da fibre ottiche o altre tecnologie di connettività via cavo, rimane comunque un potenziale collo di bottiglia, sopratutto se si utilizza una rete di accesso wireless, come le reti cellulari LTE o UMTS. Utilizzando delle reti wireless per comunicare per esempio con dei server di un servizio cloud, si è soggetti a dei limiti intrinseci alla natura fisica del mezzo di comunicazione radio. Se infatti le reti wired risultano facilmente scalabili, dato che praticamente non esiste un limite nella quantità di cavi ed infrastrutture che si possono installare per incrementare le performance di una rete, nel caso delle reti wireless si hanno a disposizione solo una certa quantità fissa di canali o porzioni di spettro elettromagnetico in cui è possibile trasmettere. In una tipica rete cellulare, se in un certo istante di tempo un device sta trasmettendo dei dati, esso occupa tutto il canale radio in tutta la cella in cui si trova, impedendo ad altri device vicini di comunicare attraverso lo stesso canale.

Un possibile metodo per ridurre questo limite di scalabilità intrinseco nelle reti wireless, è utilizzare delle celle sempre più piccole, in modo da ridurre la quantità di dispositivi che condividono la stessa cella, e quindi la stessa quantità di banda disponibile. Questa è la strategia che è stata utilizzata finora nelle reti cellulari, e nella maggior parte delle situazioni è sufficiente per garantire delle performance accettabili. Tuttavia, in certe situazioni particolari dove si hanno delle grandi folle di persone in spazi ristretti, per esempio in eventi sportivi negli stadi o di spettacolo nei centri delle città, è possibile che le reti cellulari smettano di funzionare correttamente per la quantità troppo alta di device che utilizzano la stessa cella. È possibile ridurre ulteriormente le dimensioni delle celle, che in certi casi sono già nell'ordine delle decine di metri, ma così facendo si arriverà prima o

poi ad un punto in cui diventa difficile e poco sensato far comunicare queste piccole celle direttamente con la rete Internet, come richiesto dal modello del cloud computing.

1.2 Tecnologie wireless peer-to-peer per comunicazioni opportunistiche

Un approccio alternativo per la costruzione di sistemi distribuiti basati su reti wireless che funzionino correttamente ed in modo scalabile anche in scenari di alta densità spaziale di device connessi, è quello di utilizzare delle tecnologie di comunicazione wireless peer-to-peer per scambiarsi dati solo con i device vicini, invece di forzare ogni device a contattare direttamente un server centrale nel cloud. Le tecnologie wireless peer-to-peer permettono infatti ad un device di trasmettere una informazione direttamente ai device vicini che il segnale radio riesce a raggiungere. Queste tecnologie, grazie a vari accorgimenti come la trasmissione a bassa potenza e l'utilizzo di frequenze radio alte, riescono a formare delle "celle" di dimensioni molto piccole, all'interno delle quali le comunicazioni peer-to-peer tra i device che le compongono sono molto efficienti anche in situazioni difficili, come gli scenari di folle di persone sopra descritti.

Dato che al giorno d'oggi gli smartphone con connettività wireless sono ormai estremamente diffusi, risulterebbe molto utile in scenari di folle di persone poter utilizzare gli smartphone stessi come dispositivi che compongono un sistema distribuito che svolge una certa funzione, facendoli comunicare tra di loro tramite tecnologie wireless peer-to-peer. Nel caso per esempio di folle di persone in eventi pubblici, gli smartphone già in possesso delle singole persone, potrebbero effettuare delle comunicazioni opportunistiche con gli smartphone delle persone vicine a portata diretta di segnale radio. L'insieme degli smartphone presenti nel luogo dell'evento, formerebbe quindi un sistema distribuito che non necessita di un coordinatore centrale o di connettività Internet per raggiungere servizi cloud. Un esempio di funzioni che possono essere svolte da un sistema simile, è quello di fornire supporto alle persone in eventuali situazioni di emergenza, per esempio per trovare una via di fuga in una piazza affollata o per lanciare un allarme.

Un modello utile per la costruzione di sistemi distribuiti basati sulle comunicazioni opportunistiche, è quello dei campi computazionali. In questa

tesi, verrà successivamente descritto il modello dei campi computazionali e ne verranno analizzate delle possibili implementazioni in sistemi mobile basati su device Android e comunicazioni opportunistiche, e delle possibili applicazioni per la gestione di emergenze in uno scenario di folle in eventi pubblici.

Una categoria di sistemi distribuiti che sta diventando, anche per questo motivo, sempre più importante, è quella dei sistemi di lavoro cooperativo situato. In questi sistemi, i vari nodi cooperano tra di loro per svolgere un lavoro complessivo più complicato, dando quindi all'intero sistema un comportamento complesso che spesso emerge da un insieme di comportamenti molto più semplici. A differenza di altri sistemi distribuiti di questa categoria, questi vengono però definiti "situati" in quanto il loro comportamento dipende in modo rilevante dalla posizione dell'insieme dei nodi che li costituiscono. Generalmente per "posizione" si intende la posizione fisica dei nodi, o meglio un modello di essa che sia adatto alle caratteristiche del sistema. Un modo per intendere la posizione dei nodi molto utilizzato e conveniente nei sistemi di nostro interesse, è quello di considerare i vari nodi puntiformi, e disposti su un piano. Questo modello si presta particolarmente bene per rappresentare vari tipi di oggetti (smartphone o altri device elettronici) disposti per esempio su un terreno (una piazza, un campo, o le strade di una città). Se necessario, è possibile utilizzare uno spazio tridimensionale invece di un piano, in modo da poter rappresentare per esempio degli aerei o droni UAV. Costruire questi sistemi con un approccio di software engineering tradizionale, è generalmente estremamente difficile. In questa tesi vengono analizzate delle tecniche per la progettazione di questa tipologia di sistemi usando il modello dei campi computazionali, che in molti casi rende piuttosto semplice ed elegante la realizzazione del sistema.

Vediamo ora alcuni possibili scenari di interesse pratico, in cui possono essere utilizzati dei sistemi basati su campi computazionali per risolvere problemi reali.

1.2.1 Use case: Folle di persone in eventi pubblici

Uno scenario classico in cui possono essere utili dei sistemi basati su campi computazionali, è quello di una folla di persone, riunite in un certo luogo per esempio in occasione di un evento pubblico. Degli esempi della vita di tutti i giorni possono essere un concerto in una piazza, una partita in uno

stadio, oppure un museo o delle stazioni ferroviarie nei momenti di massimo afflusso di persone. In queste situazioni può essere necessario raccogliere o diffondere delle informazioni riguardanti per esempio la logistica o l'organizzazione della manifestazione, oppure in situazioni di emergenza può essere estremamente utile far si per esempio che ogni persona abbia chiaro quale è la via di fuga migliore per uscire da una piazza nel caso ci sia un incendio o un altro tipo di pericolo. Nel caso si debba trovare una via di fuga da una piazza piena di gente, è necessario tenere conto della direzione in cui si stanno dirigendo le altre persone, per evitare che le persone in preda al panico si concentrino in certi punti bloccando la circolazione, oppure che si dirigano in strade chiuse o già troppo piene di altre persone.

Dare questo genere di indicazioni di fuga in situazioni di emergenza è estremamente difficile con i mezzi tradizionali. Dato che la maggior parte delle persone ormai utilizza e porta sempre con se uno smartphone, si può pensare di usare gli smartphone delle singole persone per indicargli la via di fuga migliore che devono seguire. Gli smartphone di oggi hanno i sensori di posizione necessari per fare questo (bussola, accelerometro, GPS), ma hanno il grosso problema di avere connettività quasi solo esclusivamente tramite la rete cellulare. In situazioni come una piazza di una città piena di persone, le reti cellulari possono iniziare ad entrare in crisi poichè ci sono troppe persone che vogliono utilizzarle contemporaneamente in un luogo molto ristretto. In una situazione di emergenza risulterebbe quindi molto difficile inviare per esempio la posizione sempre aggiornata di ogni persona ad un server centrale che calcola le vie di fuga migliori, e poi rispedire i dati indietro a tutti gli smartphone di ogni singola persona.

Un approccio alternativo a quello "cloud" appena descritto, può essere quello di fare comunicare direttamente in modo peer-to-peer gli smartphone di ogni persona con quelli dei vicini, in modo che il sistema complessivo di tutti gli smartphone presenti nella piazza si auto-organizzi e trovi la via di fuga migliore per ogni singola persona, in modo completamente distribuito e decentralizzato. Questo è possibile realizzando un sistema basato sui campi computazionali, che usa il Wifi ad hoc (che analizzeremo in seguito nei dettagli) per avere delle comunicazioni opportunistiche con gli smartphone di altre persone nelle immediate vicinanze. Un sistema del genere può essere più scalabile e fault tolerant di un sistema basato su una infrastruttura di rete cellulare tradizionale e dei server cloud.

Capitolo 2

Tecnologie wireless peer-to-peer

Per tecnologie di comunicazione wireless peer-to-peer, si intendono delle tecnologie che permettono ad un insieme di dispositivi di comunicare tra di loro ricevendo ognuno direttamente i segnali radio trasmessi da un altro, senza la necessità di un "nodo centrale" o di altre infrastrutture di rete.

L'insieme di dispositivi che un certo nodo della rete può raggiungere, può variare in ogni istante di tempo, ed è dato dai dispositivi che riescono a ricevere con qualità sufficiente il segnale trasmesso. L'insieme di dispositivi raggiungibili è quindi dinamico, e dipendente dalla location fisica del dispositivo stesso e dei suoi vicini. Questo rende le tecnologie wireless peer-to-peer adatte per svolgere le comunicazioni opportunistiche, grazie anche al fatto che non richiedono la presenza di alcuna infrastruttura fisica se non i dispositivi stessi tra i quali la comunicazione avviene.

In questo capitolo della tesi, verranno analizzate le principali tecnologie di comunicazione wireless attualmente disponibili, ed in particolare quanto esse si prestano per un uso in modalità peer-to-peer. Verrà poi presa in considerazione la tecnologia Wifi Ad-Hoc, analizzando la fattibilità e le performance di un suo possibile utilizzo per la costruzione di sistemi basati su comunicazioni opportunistiche tra dispositivi mobile.

2.1 Survey principali tecnologie wireless

Al giorno d'oggi esistono un gran numero di tecnologie di comunicazione wireless, ognuna nata per scopi diversi e con certe caratteristiche. Una delle categorie di tecnologie wireless più di largo uso, è sicuramente quella delle reti cellulari, della quale fanno parte standard come LTE ed UMTS, utilizzati negli smartphone e altri tipi di terminali.

Le reti cellulari sono basate su una architettura incentrata su una forte dipendenza da una infrastruttura di rete, cioè di un insieme di apparati gestiti da un operatore che permettono alla rete di funzionare. L'operatore è generalmente proprietario sia dell'infrastruttura di rete, sia della licenza di utilizzare una certa porzione dello spettro elettromagnetico per far funzionare la propria rete. Nelle reti cellulari, l'utente finale della rete può quindi solo comunicare via radio con degli apparati gestiti dall'operatore, che si occuperanno poi a loro volta di metterlo in contatto con una rete più grande, come Internet o la rete telefonica. Anche se le reti cellulari rappresentano un ottimo mezzo per accedere a reti WAN come Internet, non permettono invece di effettuare comunicazioni peer-to-peer, cioè trasmissioni dirette tra due dispositivi senza aver bisogno di alcuna infrastruttura di rete.

Un'altra categoria di tecnologie di comunicazione wireless, è quella degli standard progettati per funzionare sulle unlicensed bands, cioè delle bande di frequenze che sono state allocate per essere utilizzate da chiunque senza la necessità che possieda una licenza per utilizzarle. Al contrario delle reti cellulari dove l'operatore ha pieno controllo di chi trasmette nelle proprie frequenze, nelle frequenze unlicensed ogni utilizzatore può incontrare delle interferenze provenienti da altri utilizzatori vicini. Per questo motivo la potenza di trasmissione sulle frequenze unlicensed è limitata per legge ad un valore piuttosto basso, in modo da ridurre la zona in cui ogni utilizzatore introduce interferenze. Sulle bande unlicensed è quindi permessa ad ogni utente la comunicazione senza la mediazione di apparati di rete di un operatore.

Questo viene sfruttato da alcuni standard come il Wifi Infrastructure per permettere ad ogni utente di costruirsi una propria rete con un proprio punto centrale che svolge un ruolo simile agli apparati di rete degli operatori cellulari, oppure altri standard permettono di effettuare comunicazioni peer-to-peer vere e proprie senza alcun nodo centrale. Gli standard appartenenti a questa ultima categoria, sono evidentemente adatti per essere impiegati

nella costruzione di sistemi basati su comunicazioni opportunistiche.

Verrà ora effettuato un survey delle principali tecnologie wireless attualmente disponibili operanti su frequenze unlicensed, analizzandone le caratteristiche principali e quanto siano adeguate per essere utilizzate in sistemi basati su comunicazioni opportunistiche.

Per ogni tecnologia è generalmente stabilito un raggio di copertura approssimativo, entro il quale viene solitamente ricevuto il segnale trasmesso. Tuttavia va sottolineato che questo raggio di copertura è estremamente approssimativo, e serve solo per avere un indice utile per un primo confronto tra tecnologie diverse. La reale distanza tra cui due device possono comunicare può variare di molto in base a tanti fattori, per esempio alla quantità di rumore ed interferenze radio, alla presenza di ostacoli e più in generale in base a tutti gli oggetti presenti nello spazio raggiunto dal segnale radio.

Bluetooth

La versione "classic" del Bluetooth (molto differente dalla versione Low Energy) è stata sviluppata a fine anni '90 principalmente per essere utilizzata per connettere degli auricolari senza fili ai telefoni cellulari, ed è stata poi estesa supportando molti altri "profili", cioè protocolli di alto livello pensati per trasportare altri tipi di informazioni.

È presente in quasi tutti gli smartphone ed in molti altri device messi sul mercato dagli anni 2000.

I profili più utili per effettuare comunicazioni peer-to-peer di informazioni arbitrarie, sono il Serial Port Profile, che permette di simulare una comunicazione seriale tra 2 soli device, oppure il Personal Area Networking Profile, che permette di creare delle piccole reti di massimo 7 device. Questo numero massimo di device rappresenta quindi un grosso limite alla scalabilità di un eventuale sistema di comunicazioni opportunistiche. Un potenziale modo per arginare il problema sarebbe connettersi a turni con i device che si hanno intorno, in stile round-robin.

Il bitrate teorico che permette di ottenere il layer fisico Bluetooth è di circa 1-3 Mbit/s, a seconda delle versioni del protocollo che vengono utilizzate. La velocità di trasferimento reale può essere ancora più bassa, nell'ordine delle centinaia di kbit/s. Inoltre ha un raggio di copertura molto limitato (circa 10 metri) e consuma relativamente molta energia, anche in "idle" mentre non vengono scambiati dati.

I dispositivi Bluetooth per poter scambiare informazioni devono necessariamente essere connessi tra di loro usando un certo profilo, e la procedura di establishment della connessione impiega una quantità di tempo nell'ordine dei secondi, quindi piuttosto alta per un sistema che deve operare con un gran numero di device.

Lo standard Bluetooth "classic" non si presta quindi per lo sviluppo di sistemi basati su comunicazioni opportunistiche su larga scala, come per esempio comunicazioni opportunistiche in una piazza piena di persone. Tuttavia può essere utile in sistemi dove c'è la necessita di far comunicare sempre solo pochi device contemporaneamente.

Bluetooth LE (Low Energy)

Questo standard è stato pensato per superare le limitazioni del Bluetooth classico ed essere usato in dispositivi che richiedono un consumo energetico estremamente basso, sopratutto in idle.

Questo standard è disponibile negli smartphone usciti sul mercato circa dal 2013. Non è stato pensato per sostituire il Bluetooth classico, ma per affiancarlo, dato che ha una velocità di trasferimento dati estremamente più lenta che non permetterebbe per esempio di trasferire audio in tempo reale ad un auricolare Bluetooth.

Il Bluetooth LE è stato sostanzialmente progettato solamente per comunicare pochissimi byte al secondo di informazione con device a basso costo ed a basso consumo. Esempi di device che utilizzano il Bluetooth LE sono: cardiofrequenzimetri da polso, beacon che segnalano la loro presenza agli smartphone, altri tipi di dispositivi e sensori che devono trasmettere (o meglio segnalare) una piccola quantità di informazione utilizzando pochissima energia. Molti device Bluetooth LE come i beacon, riescono a durare addirittura degli anni essendo alimentati solamente da una batteria a bottone.

Il bitrate con cui vengono modulati i dati trasmessi (il cosiddetto "air rate") è di 1 Mbit/s, ma in realtà le caratteristiche del protocollo MAC (di accesso al mezzo), rendono disponibile a livello applicativo un bitrate di solo qualche decina di kbit al secondo. Inoltre il protocollo MAC sostanzialmente non gestisce gli scenari di collisioni tra due trasmissioni contemporanee effettuate da due dispositivi contemporaneamente, quindi le performance

calerebbero ulteriormente in modo drastico nel caso vengano usati molti device nello stesso luogo.

In linea teorica sarebbe possibile realizzare una rete "mesh" che si appoggi al Bluetooth LE implementando un protocollo di routing, ma si otterebbero molto probabilmente delle performance pessime. L'unica implementazione commerciale nota di una rete basata su Bluetooth LE è CSR Mesh [1], una rete mesh basata però sul flooding, quindi con performance e scalabilità molto limitate. Inoltre il raggio di copertura estremamente limitato (inferiore ai 10 metri) ne limita ulteriormente l'utilità.

Il Bluetooth LE risulta quindi inadatto in scenari di comunicazioni opportunistiche dove va scambiata una quantità non minima di informazioni.

IEEE 802.15.4

Questo standard è stato pensato per realizzare delle reti tra device a basso costo ed a basse performance, come dei microcontrollori.

A differenza del Bluetooth LE, questo standard è basato su un protocollo MAC molto più efficiente nel gestire le collisioni tra trasmissioni contemporanee, grazie ad un algoritmo basato sul CSMA, simile a quello utilizzato dagli standard più performanti come il Wifi. Anche se ha un "air rate" di soli 250 kbit/s, permette quindi di ottenere performance reali effettive migliori di quelle del Bluetooth LE, sopratutto quando viene usato in una rete formata da molti device.

Per questo motivo è abbastanza diffuso in sistemi di Internet of Things o home automation, dove è fondamentale poter gestire una rete formata da un gran numero di device. È infatti il protocollo su cui si basano standard come ZigBee per l'home automation, o il più recente Thread [2] sponsorizzato da Google ed utilizzato nei termostati intelligenti Nest.

Ad oggi non è però presente in alcun smartphone, computer o altri tipi di device mobile. Questo ne limita fortemente un eventuale impiego in sistemi di comunicazioni opportunistiche basati su dispositivi mobile.

Wifi Ad-hoc (IEEE 802.11 IBSS)

Lo standard IEEE 802.11, meglio noto come Wifi, è stato sviluppato per permettere di creare delle reti LAN wireless con performance possibilmente alte come quelle delle LAN wired.

A differenza degli standard wireless precedentemente illustrati come Bluetooth LE o 802.15.4, il Wifi è stato quindi progettato considerando le performance come obiettivo principale, a scapito dei consumi energetici e dei costi di produzione dei device. La velocità delle reti Wifi dipende dalla versione dello standard 802.11 utilizzata. La più vecchia attualmente in uso è la 802.11b, che prevede un massimo teorico di 11 Mbit/s. La più veloce attualmente disponibile è invece la 802.11ac, che permette addirittura di superare 1 Gbit/s, utilizzando device di fascia alta. In situazioni normali, utilizzando device Wifi di basso costo, ci si può tranquillamente aspettare di ottenere delle velocità reali nell'ordine delle decine di Mbit/s, quindi 3 - 4 ordini di grandezza superiori rispetto alle tecnologie precedentemente descritte.

Lo standard IEEE 802.11 su cui è basato il Wifi, prevede due tipologie di reti: infrastructure ed ad hoc.

La modalità infrastructure prevede la presenza di un device centrale chiamato Access Point con il quale tutti gli altri device comunicano, e che svolge funzioni di coordinamento della rete. Questa è la modalità più comunemente usata nelle reti casalinghe, negli uffici o negli hotspot pubblici per la connessione ad Internet, ma ovviamente non è adatta per delle comunicazioni opportunistiche peer-to-peer, dato che richiede la presenza di un nodo centrale.

La modalità ad hoc, chiamata anche IBSS (Independent Basic Service Set), permette invece di creare una rete wireless totalmente decentralizzata, che non necessita di un nodo centrale. Ogni nodo può comunicare direttamente con gli altri nodi che vengono raggiunti direttamente dal suo segnale radio.

Lo standard 802.11 definisce i layer PHY e MAC dello stack ISO/OSI. Nei layer superiori viene generalmente utilizzato lo stack di protocolli IP. È quindi possibile utilizzare per esempio il TCP per scambiarsi efficientemente grandi quantità di informazioni, oppure inviare pacchetti UDP all'indirizzo di broadcast. Nel caso delle reti Wifi ad hoc, i pacchetti broadcast inviati possono essere ricevuti da tutti i device che si trovano in quel momento a portata del segnale radio, senza alcuna procedura di handshaking o di establishment di connessioni. Il Wifi in modalità ad hoc può essere quindi particolarmente adatto per sistemi basati su comunicazioni opportunistiche su larga scala, che debbano essere performanti, scalabili, e supportare un numero di device elevato concentrati nello stesso luogo.

Le reti basate sullo standard Wifi sono estremamente diffuse, e le radio 802.11 sono quindi presenti in una grande quantità di dispositivi di largo uso, come smartphone, computer, televisori, ed anche dispositivi Internet of Things come termostati, lampade, elettrodomestici. Questi dispositivi vengono però generalmente utilizzati esclusivamente per reti di tipo Infrastructure, poichè sono la tipologia più diffusa in ambito casalingo o aziendale. L'hardware delle radio 802.11 di questi dispositivi è quasi sempre in grado di supportare anche la modalità Wifi ad hoc, tuttavia molto spesso i driver del sistema operativo ed il resto del software utilizzato nel dispositivo, funzionano correttamente esclusivamente in modalità Infrastruture. È possibile aggirare queste limitazioni ricorrendo per esempio a driver alternativi, ma solo in una piccola parte dei device in commercio si riescono ad avere risultati soddisfacenti. Per poter effettuare ricerca e sperimentazione sulle reti Wifi Ad-Hoc, è quindi necessario scegliere dei dispositivi con un chipset Wifi possibilmente supportato da driver open-source, e noto nella comunità per supportare correttamente la modalità ad hoc.

Wifi Mesh (IEEE 802.11s)

Lo standard 802.11s è una estensione del Wifi tradizionale che aggiunge alle classiche modalità di rete Infrastructure ed Adhoc, una nuova modalità "mesh". Per rete mesh, si intende una rete decentralizzata dove ogni nodo della rete può funzionare da router, ricevendo pacchetti da altri nodi ed instradandoli ai nodi dove sono diretti.

Lo standard 802.11s è stato principalmente pensato per creare reti mesh Wifi che gestiscono il protocollo di routing mesh a livello 2 dello stack ISO/OSI, e forniscono ai livelli superiori una astrazione di un unica grande rete, nascondendo il routing mesh. Questa potrebbe essere una caratteristica negativa vista nell'ottica dello sviluppo di sistemi peer-to-peer per comunicazioni opportunistiche, dato che non vengono esposte le comunicazioni dirette tra i due device, ma vengono invece mascherate dal protocollo di routing mesh. Per questo motivo, in certi casi la comunicazione tra due device potrebbe non essere possibile per via del protocollo di routing mesh, anche se la rete vera e propria sottostante permetterebbe di comunicare tra di loro.

Wifi Direct

Lo standard Wifi Direct rappresenta una estensione delle reti Wifi Infrastructure tradizionali, e permette di creare in modo semplice una rete tra più device senza dover configurare manualmente un access point. Grazie al Wifi Direct dei dispositivi raggiungibili l'un l'altro dal segnale radio possono accordarsi per selezionare automaticamente un nodo tra di essi che svolgerà il ruolo di Access Point. Una volta eseguito questo setup iniziale della rete, essa diventa una normale rete Wifi Infrastructure.

Nella maggior parte delle implementazioni del Wifi Direct non è prevista la possibilità per un nodo di essere associato a più di un Access Point, quindi in molti scenari di sistemi peer-to-peer ci si potrebbe trovare nella situazione in cui due nodi sarebbero in grado di comunicare tra di loro, ma non possono poichè sono già connessi ognuno ad Access Point diversi.

2.2 Reti Wifi ad hoc per comunicazioni opportunistiche

Come descritto nella sezione precedente, le reti Wifi ad hoc sono uno strumento piuttosto potente ed adatto per la realizzazione di sistemi basati su comunicazioni opportunistiche. In questa sezione della tesi, verrà analizzato come gestire una rete Wifi ad hoc dal punto di vista software per realizzare un semplice sistema di comunicazione peer-to-peer tra dispositivi mobile.

Le reti Wifi ad hoc, come le più comuni reti Wifi infrastructure, sono identificate da un SSID (Service Set Identified), un codice alfanumerico scelto arbitrariamente da chi crea la rete. Il SSID è comunemente chiamato in modo improprio "nome della rete". Come nel caso delle reti Wifi infrastructure, possono coesistere nello stesso luogo più reti ad hoc separate tra di loro. Dato che il SSID è scelto dall'utente, possono crearsi dei conflitti, creando nello stesso luogo due reti con lo stesso SSID. Questo problema viene risolto identificando univocamente una rete Wifi ad hoc tramite un BSSID, un codice di 48 bit estratto in modo random da ogni device che effettua un join alla rete in mancanza di altri device raggiungibili dal segnale radio già connessi alla rete identificata dallo stesso SSID. Nel caso in cui due gruppi di device inizialmente separati tra di loro ma entrambi connessi ad una rete ad hoc identificata dallo stesso SSID, entrino in contatto uno con

l'altro, le due reti devono essere unite in una sola, quindi deve essere scelto in modo distribuito un unico BSSID per la rete complessiva. Questa operazione di merging è molto complessa, e spesso i driver di molti chipset Wifi non la implementano correttamente, causando quindi dei malfunzionamenti. Per arginare il problema, è possibile in certi casi settare manualmente il BSSID ad un valore prestabilito in tutti i nodi della rete, evitando quindi di dover effettuare dei merge. [3]

Lo standard 802.11 prevede la suddivisione in canali dello spettro di frequenze nel quale opera. Le reti Wifi, sia di tipo ad hoc che di tipo infrastructure, possono operare su un solo canale per volta. Nel caso delle reti infrastructure, il canale sul quale opera la rete è scelto dall'Access Point, che informa automaticamente tutti i device al quale vogliono connettersi del canale che devono utilizzare. Nel caso delle reti ad hoc invece non esiste un Access Point centrale che può stabilire su che canale opera la rete, quindi ogni nodo deve decidere il canale sul quale effettuare il join alla rete ad hoc. Nella progettazione di un eventuale sistema di comunicazioni opportunistiche basato su reti Wifi ad hoc, è quindi fondamentale far si che tutti i device che fanno parte del sistema, utilizzino un canale prestabilito.

Nelle reti Wifi infrastructure, generalmente il dispositivo che svolge il ruolo di Access Point è un apparato di rete che in realtà svolge molte altre funzioni, tra le quali spesso quella di server DHCP che configura i device della rete nella quale operano. Questo per esempio permette ai router casalinghi di assegnare automaticamente un indirizzo IP a tutti i dispositivi che si connettono alla rete locale da essi gestita tramite il loro Access Point interno. Nel caso delle reti Wifi ad hoc utilizzate in contesti di comunicazioni opportunistiche, non può esistere un coordinatore centrale di tutta la rete che assegna indirizzi IP ai vari nodi tramite DHCP, quindi ogni nodo dovrà assegnarsi lui stesso un proprio indirizzo IP. Ovviamente non devono esserci conflitti di indirizzi IP, quindi ogni device della rete deve utilizzare un indirizzo IP univoco in tutta la rete. Una possibile strategia per l'assegnazione di indirizzi IP può essere stabilire un indirizzo IP fisso per ogni device fisico. Nel caso questo non fosse possibile, può essere usato un indirizzo IP estratto casualmente dal device stesso, cercando di ridurre al minimo la probabilità di conflitti. Dato che gli indirizzi IPv4 privati sono piuttosto limitati, può essere utile utilizzare il protocollo IPv6.

Buona parte dei protocolli di rete applicativi, cioè appartenenti al layer 7 dello stack ISO/OSI, si appoggiano al protocollo di trasporto TCP per

gestire la trasmissione di dati tra due endpoint. Le connessioni TCP sono necessariamente unicast, cioè permettono di scambiare dati solamente da un endpoint ad un altro endpoint. In un sistema di comunicazioni opportunistiche, potrebbe però essere utile trasmettere un dato in broadcast a tutti gli altri device che il mio segnale radio riesce a raggiungere, senza neanche conoscere il loro indirizzo IP e se sono effettivamente presenti o meno. Nelle reti Wifi ad hoc, questo può essere ottenuto inviando pacchetti UDP all'indirizzo IP di broadcast. Nel caso di una rete con indirizzi 192.168.0.0/24, l'indirizzo IP di broadcast da impostare come destinazione dei pacchetti UDP da inviare, è quindi 192.168.0.255. Ovviamente inviando pacchetti in broadcast in questo modo, verranno raggiunti solamente i nodi della rete direttamente raggiungibili dal segnale radio del trasmettitore, a meno che la rete non implementi qualche forma di routing mesh, di default non presente nelle reti Wifi ad hoc.

Il layer di Media Access Control dello standard 802.11 implementa la trasmissione di pacchetti broadcast con una trasmissione di un segnale radio senza garanzie di ricezione da parte del destinatario, al contrario delle normali trasmissioni unicast dove viene adottato un meccanismo di ACK per ritrasmettere pacchetti persi e di RTS/CTS per evitare interferenze con altri utilizzatori della banda. Nello standard 802.11, le trasmissioni unicast avvengono generalmente alla velocità (air rate) migliore consentita dalla potenza del segnale radio tra sorgente e destinatario, mentre invece le trasmissioni in broadcast avvengono ad un air rate piuttosto basso, in modo da permettere a device che ricevono un segnale molto debole di decodificare ugualmente il pacchetto trasmesso.

Alcuni driver di certi chipset 802.11 permettono di impostare manualmente la velocità con cui inviare i pacchetti di broadcast. In sistemi di comunicazioni opportunistiche basati su reti Wifi che effettuano molte comunicazioni broadcast, la velocità di trasmissione dei pacchetti broadcast rappresenta quindi un importante parametro su cui intervenire per ottimizzare le performance del sistema.

2.3 Simulazione performance di reti Wifi ad hoc

Nelle sezioni precedenti della tesi, sono state analizzate le varie caratteristiche delle reti Wifi ad hoc utili per esempio in sistemi di comunicazioni opportunistiche con dispositivi mobile. Buona parte di questi sistemi hanno però il requisito di essere scalabili e performanti anche con un numero di device partecipanti alla rete piuttosto elevato. Risulterebbe quindi molto interessante poter valutare le performance di un potenziale sistema basato su reti Wifi ad hoc, al variare del numero dei device che fanno parte di esso.

Per ottenere dati sulle performance di qualità più alta possibile, la soluzione ideale sarebbe effettuare un test con dei device reali, disposti in un luogo reale, in modo da riprodurre il più possibile il funzionamento che avrebbero nel sistema vero e proprio. Questa soluzione però incontra presto dei limiti pratici molto forti all'aumentare dei device che dovrebbero far parte del sistema. Se si vuole testare per esempio un eventuale sistema formato dagli smartphone delle persone che compongono una folla in un evento pubblico, si può arrivare facilmente a dover effettuare un test con centinaia di device, cosa difficilmente attuabile sia dal punto di vista economico che logistico.

Una soluzione alternativa per ottenere una prima stima delle performance di un sistema come quello sopra descritto, può essere invece quella di effettuare una simulazione al computer del sistema che si vuole analizzare, cercando di modellare in modo più rigoroso possibile gli aspetti del sistema più legati alle performance. In una rete Wifi ad hoc di grandi dimensioni, come quella necessaria in un ipotetico sistema di comunicazioni opportunistiche formato da centinaia di device vicini tra loro, le performance possono essere limitate in modo significativo dalle interferenze provenienti dal segnale radio di altri dispositivi, che si sovrappone al segnale che si sta tentanto di trasmettere o ricevere, abbassando quindi il Signal to Noise Ratio (SNR) del segnale a cui si è interessati. Le performance sono di conseguenza anche dipendenti da come il layer di Media Access Control del Wifi gestisce la trasmissione di informazioni in un ambiente con interferenze.

Per poter effettuare una simulazione significativa sotto l'aspetto delle performance, è necessario quindi simulare il comportamento dei device che compongono il sistema in modo accurato dal layer 2 dello stack ISO/OSI in giù, cioè i layer MAC e PHY di un device 802.11. I layer superiori possono

essere invece eventualmente rimpiazzati con una semplice logica fake che per esempio invia periodicamente un pacchetto UDP in broadcast ai vicini.

Simulazioni di questo tipo, possono essere effettuate da alcuni simulatori di rete, software utilizzati tradizionalmente nell'Ingegneria delle Telecomunicazioni per simulare il comportamento di reti di vario tipo. Alcuni di questi simulatori, per esempio ns-2 ed ns-3, sono nati in ambito accademico e sono sviluppati da una comunità open source. Altri come Omnet++ sono sviluppati da società commerciali e sono closed source.

Buona parte dei simulatori di rete, sono nati prima del rilascio dello standard 802.11 e delle reti wireless in generale, quindi sono stati spesso originariamente pensati per le reti wired, e solo successivamente estesi per le reti wireless come quelle Wifi. Molti simulatori che ho analizzato modellano infatti in modo abbastanza impreciso le reti di dispositivi Wifi, e non supportano per esempio gli standard più recenti come 802.11n oppure alcune modalità come il Wifi ad hoc. Sotto questo punto di vista, il simulatore più completo che ho trovato, dopo aver analizzato quelli più noti, è ns-3.

In questa sezione della tesi, verrà quindi descritta una sperimentazione di simulazione di un sistema di comunicazioni opportunistiche di esempio tramite il simulatore di rete ns-3.

2.3.1 Introduzione al simulatore di rete ns-3

ns-3 è un simulatore di rete ad eventi discreti, sviluppato da una comunità open source gestita da un consorzio fondato da INRIA ed University of Washington [4] . ns-3 è nato come una riscrittura completa di ns-2, un simulatore sviluppato negli anni 90 come progetto di ricerca. A differenza di ns-2, è stato sviluppato in modo più modulare e cerca di rendere più semplice la creazione di modelli di simulazioni.

ns-3 si presenta come una libreria C++, sulla quale l'utente può appoggiarsi scrivendo un proprio software finalizzato ad eseguire una certa tipologia di simulazione. Il metodo classico per effettuare una simulazione con ns-3 consiste nel sviluppare una procedura main in C++ che istanzia i modelli dei componenti di rete simulati, esegue la simulazione ed infine visualizza in output i risultati della simulazione che interessano.

Tra le feature che offre ns-3 utili per la simulazione di un sistema di comunicazioni opportunistiche basato su reti Wifi ad hoc, le principali sono:

- Simulazione della propagazione fisica delle onde radio, tramite un modello di propagation loss e propagation delay basato su vari modelli di luoghi fisici. [5]
- Simulazione del layer PHY di 802.11, inclusi gli standard più recenti e veloci come 802.11n. [6]
- Simulazione del layer MAC di 802.11, inclusa la modalità ad hoc. Può essere utilizzato un modello di controllo del bitrate che utilizza lo stesso algoritmo usato di default dal kernel Linux, in modo da simulare il più possibile il comportamento di un device Linux-based, per esempio uno smartphone Android.

2.3.2 Metodologia di simulazione di comunicazioni opportunistiche con Wifi ad hoc

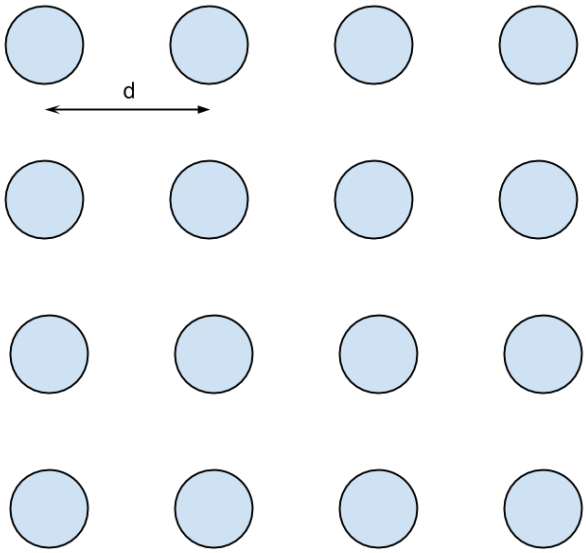
L'obiettivo della simulazione che si vuole effettuare tramite ns-3, è quello di misurare le performance di un ipotetico sistema basato sulle comunicazioni opportunistiche tra device mobile effettuate tramite reti Wifi ad hoc, al variare del numero dei device, in modo da valutare la scalabilità del sistema in oggetto.

Uno scenario di uso reale utile da essere simulato al fine della stima delle performance, può essere quello di una folla di persone che esegue sui propri smartphone un software che effettua comunicazioni opportunistiche con gli smartphone delle persone vicine. Quello appena descritto può essere uno scenario d'uso molto rappresentativo di un sistema finalizzato per esempio alla gestione di situazioni di emergenza in eventi pubblici come un concerto in una piazza o una partita in uno stadio.

Per poter simulare lo scenario sopra descritto, è necessario definire un modello del comportamento del software applicativo eseguito sui device che effettua le comunicazioni opportunistiche. Ipotizzando che l'obiettivo del software sia quello di propagare una informazione a tutti i nodi vicini che riescono a ricevere il suo segnale, il suo comportamento può essere rappresentato da un invio periodico di pacchetti UDP all'indirizzo IP di broadcast.

Per poter effettuare la simulazione è anche necessario fornire ad ns-3 un modello della disposizione fisica dei device che compongono il sistema e dell'ambiente fisico attorno ad essi, in modo da permettere ad ns-3 di simulare

la propagazione delle onde radio. Rappresentare in modo accurato uno scenario come una folla di persone in una piazza con uno smartphone in tasca, è estremamente difficile utilizzando dei modelli di propagazione radio come quelli integrati in ns-3. Ho quindi deciso di utilizzare un modello semplificato dello scenario fisico, considerando i device disposti a griglia in un piano come rappresentato in figura, ed impostando il modello di propagazione LogDistancePropagationLossModel fornito da ns-3.



Il valore di distanza d rappresentato in figura, è stato impostato a 0,96 metri, valore ricavato dalla densità standard delle folle di 1,08 persone al metro quadro, già utilizzata in letteratura scientifica.

Anche se questo rappresenta un modello molto semplificato del sistema reale che si vuole simulare, dovrebbe comunque essere sufficiente al fine di fornire al layer PHY del 802.11 una astrazione della propagazione fisica delle onde radio tale che esso si comporti in modo simile a come si sarebbe comportato nel corrispondente sistema reale.

Altri parametri importanti da fornire ad ns-3 riguardano di quale versione dello standard 802.11 devono essere i layer MAC e PHY dei device simulati, e come devono essere configurati. Avendo come obiettivo quello di simulare un ipotetico sistema composto da device mobili, ho scelto la versione dello standard Wifi più diffusa tra gli smartphone e tablet attualmente in commercio, cioè 802.11n, che offre tra l'altro un vantaggio prestazionale

notevole rispetto agli standard precedenti 11g e 11b. I layer PHY e MAC sono poi stati configurati per funzionare in modalità di rete ad hoc, ed impostando la velocità di broadcast a 6,5 Mbit/s. Ho scelto questo particolare valore per la velocità di broadcast perchè è la velocità più bassa tra quelle consentite dallo standard 802.11n [7], e che beneficia quindi di alcuni vantaggi prestazionali dati da funzionalità come MIMO e frame aggregation [8] . Utilizzando una velocità bassa di tramissione dei pacchetti broadcast, si permette ai nodi più lontani di riceverli, poichè il segnale diventa ricevibile anche con un Signal to Noise Ratio (SNR) inferiore rispetto agli air rate più alti. Parallelamente però si riduce l'efficienza dell'uso dello spettro, e si aumentano le collisioni a parità di quantità di dati trasmessi globalmente.

Lo standard 802.11n prevede anche la possibilità di trasmettere su canali di larghezza di banda di 40 MHz invece dei tradizionali 20 MHz, ottenendo in certi casi delle performance più elevate. Nella simulazione ho tuttavia utilizzato dei canali a 20 MHz, poichè in un ambiente caratterizzato da numerose interferenze, come quello per esempio di una piazza di persone dove potrebbe trovarsi il sistema che si sta studiando, è consigliato non utilizzare canali di banda troppo larga, poichè potrebbero portare a performance addirittura più basse, per via del maggior numero di collisioni a cui andrebbero incontro.

Di seguito è riportato il codice C++ del software che ho realizzato, che si appoggia ad ns-3 per eseguire la simulazione sopra descritta:

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/config-store-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/random-variable-stream.h"

#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace ns3;


NS_LOG_COMPONENT_DEFINE ("OpportunisticWifiAdHoc");


uint32_t receivedPackets = 0;

Ptr<UniformRandomVariable> randVar;


// Metodo invocato come callback ogni volta che viene ricevuto un pacchetto
void ReceivePacket (Ptr<Socket> socket)
{
  while (socket->Recv ())
    {
      NS_LOG_INFO ("Received one packet!");
    receivedPackets++;
```

```
32      }
33  }
34
35
36  static void GenerateTraffic (
37      Ptr<Socket> socket, uint32_t pktSize,
38          uint32_t pktCount, Time basePktInterval
39    )
40  {
41    if (pktCount > 0)
42    {
43        socket->Send (Create<Packet> (pktSize));
44        NS_LOG_INFO ("inviato un pacchetto");
45
46        double randVal = 0;
47        Time pktInterval = Seconds (randVal + basePktInterval.GetSeconds ());
48        Simulator::Schedule (pktInterval, &GenerateTraffic,
49           socket, pktSize,pktCount-1, basePktInterval);
50    }
51    else
52    {
53        socket->Close ();
54    }
55  }
56
57
58
59  int main (int argc, char *argv[])
60  {
61
62    uint32_t latoQuadrato = 5;
63    uint32_t numDevices;
64    uint32_t packetSizeBytes = 1000;
65    double interval = 1.0; // seconds
66    uint32_t numPackets = 5;
67    double distance = 2; // metri
68
69    // generatore casuale di un "pezzo" dell'intervallo di tempo tra un pacchetto e l'
        altro, in secondi
70    randVar = CreateObject<UniformRandomVariable> ();
71    randVar->SetAttribute ("Min", DoubleValue (0.0));
72    randVar->SetAttribute ("Max", DoubleValue (1.5));
73
74    CommandLine cmd;
75    cmd.AddValue("latoQuadrato", "numero di device che ci sono lungo il lato del quadrato
        in cui sono disposti", latoQuadrato);
76    cmd.AddValue("interval", "intervallo di tempo in secondi tra l'invio di un pacchetto
        e l'altro", interval);
77    cmd.Parse(argc, argv);
78
79    numDevices = latoQuadrato * latoQuadrato;
80
81
82    NodeContainer c;
83    c.Create (numDevices);
84
85    // Configuro il modello di propagazione delle onde radio
86    YansWifiChannelHelper wifiChannel;
87    wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
88    wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel",
89      "Exponent", DoubleValue (3.0)); // questo sarebbe per i 5 GHz
90
91    // Configuro il layer 1 (physical) della radio Wifi
92    YansWifiPhyHelper wifiPhyHelper = YansWifiPhyHelper::Default ();
93    wifiPhyHelper.SetChannel (wifiChannel.Create ());
94    wifiPhyHelper.Set ("ChannelBonding",BooleanValue(false)); // canali 20 MHz
95    wifiPhyHelper.Set ("ShortGuardEnabled",BooleanValue(false));
96    wifiPhyHelper.Set ("GreenfieldEnabled",BooleanValue(true)); // 11n only
97    wifiPhyHelper.SetChannel (wifiChannel.Create ());
98
99    WifiHelper wifi = WifiHelper::Default ();
100   wifi.SetStandard (WIFI_PHY_STANDARD_80211n_5GHZ);
101   std::string phyMode ("OfdmRate6_5MbpsBW20MHz");
102   wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
103     "DataMode", StringValue(phyMode),
104     "ControlMode", StringValue(phyMode),
```

```
105     "NonUnicastMode", StringValue(phyMode)); // dovrebbe settare quella modalita' anche
          per i pacchetti multicast e broadcast
106
107   // Configuro il layer 2 (MAC) del Wifi
108   HtWifiMacHelper wifiMacHelper = HtWifiMacHelper::Default ();
109   wifiMacHelper.SetType ("ns3::AdhocWifiMac");
110   // non so se questi sotto funzionano, sono ottimizzazioni dell'802.11n
111   wifiMacHelper.SetMsduAggregatorForAc (AC_VO, "ns3::MsduStandardAggregator", "
        MaxAmsduSize", UintegerValue (3839));
112   wifiMacHelper.SetBlockAckThresholdForAc (AC_BE, 10);
113   wifiMacHelper.SetBlockAckInactivityTimeoutForAc (AC_BE, 5);
114
115   NetDeviceContainer devices;
116   devices = wifi.Install (wifiPhyHelper, wifiMacHelper, c);
117
118   // Configuro la posizione fisica nello spazio dei device
119   MobilityHelper mobility;
120   mobility.SetPositionAllocator (
121     "ns3::GridPositionAllocator",
122     "MinX", DoubleValue (0.0),
123     "MinY", DoubleValue (0.0),
124     "DeltaX", DoubleValue (distance),
125     "DeltaY", DoubleValue (distance),
126     "GridWidth", UintegerValue (latoQuadrato),
127     "LayoutType", StringValue ("RowFirst"));
128
129   // i device stanno fermi, non si muovono
130   mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
131   mobility.Install (c);
132
133
134   // Inizializzo lo stack IP ed UDP
135   InternetStackHelper internet;
136   internet.Install (c);
137
138   Ipv4AddressHelper ipv4;
139   NS_LOG_INFO ("Assign IP Addresses.");
140   ipv4.SetBase ("192.168.0.0", "255.255.0.0");
141   Ipv4InterfaceContainer interf = ipv4.Assign (devices);
142
143   TypeId udpTid = TypeId::LookupByName ("ns3::UdpSocketFactory");
144   int i;
145   for(i = 0; i < numDevices; i++){
146
147     // Socket che invia i pacchetti UDP broadcast
148     Ptr<Socket> sender = Socket::CreateSocket (c.Get (i), udpTid);
149     InetSocketAddress destinationAddr = InetSocketAddress (Ipv4Address ("
          255.255.255.255"), 10000);
150     sender->SetAllowBroadcast (true);
151     sender->Connect (destinationAddr);
152
153     // Socket che riceve i pacchetti UDP broadcast
154     Ptr<Socket> recvSink = Socket::CreateSocket (c.Get (i), udpTid);
155     InetSocketAddress local = InetSocketAddress (Ipv4Address::GetAny (), 10000);
156     recvSink->Bind (local);
157     recvSink->SetRecvCallback (MakeCallback (&ReceivePacket));
158
159     Simulator::ScheduleWithContext (sender->GetNode ()->GetId (),  Seconds (randVar->
          GetValue ()), &GenerateTraffic, sender, packetSizeBytes, numPackets, Seconds(
          interval));
160   }
161
162   std::cout << "Inizio simulazione\n";
163
164   Simulator::Run ();
165   Simulator::Destroy ();
166
167   std::cout << "Fine! Pacchetti ricevuti: " << receivedPackets << ", " << (double(
        receivedPackets)/double(numPackets*numDevices*(numDevices-1)))*100 << "% di quelli
         previsti\n";
168
169   return 0;
170 }
```

Il programma sopra riportato, accetta i seguenti parametri da linea di

comando:

- latoQuadrato: nel modello fisico della simulazione, i device sono disposti in una griglia che forma un quadrato nel quale il numero di device disposti lungo un suo lato, è dato da questo parametro.

- interval: l'intervallo di tempo medio tra un invio di un pacchetto di broadcast ed un altro, da parte di un qualsiasi device, misurato in secondi.

Al termine della simulazione, il programma visualizza in output la quantità di pacchetti ricevuti, calcolata sommando il numero di pacchetti ricevuti correttamente da ogni device, e la relativa percentuale rispetto al numero di pacchetti inviati complessivamente da tutti i device. Questa percentuale rappresenta l'efficienza del sistema e può essere utilizzata come indice di performance per confrontare diverse configurazioni del modello di simulazione.

2.3.3 Analisi risultati simulazioni

Utilizzando il software di simulazione descritto al punto precedente, ho effettuato vari esperimenti per valutare le performance del sistema simulato al variare di alcuni parametri.

L'obiettivo principale degli esperimenti, è quello di analizzare quanto un ipotetico sistema di comunicazioni opportunistiche basato su una rete Wifi ad hoc, modellato come descritto al punto precedente, sia scalabile rispetto al numero di device che ne fanno parte. I dati più importanti da ricavare, saranno quindi quelli delle performance in funzione del numero di device presenti nella rete simulata. Come indice per valutare le performance di ogni esperimento, ho considerato la percentuale di pacchetti ricevuti globalmente, descritta nel punto precedente.

Un altro parametro importante della simulazione, consiste nel valore dell'intervallo di tempo tra una trasmissione di un pacchetto broadcast e la successiva, per ogni device. Per massimizzare l'efficienza del sistema, occorre minimizzate le collisioni tra pacchetti, dato che i pacchetti trasmessi contemporaneamente vengono persi. Il comportamento di ogni singolo device del sistema, dovrebbe quindi essere progettato in modo da minimizzare la probabilità di generare collisioni. I sistemi nei quali i pacchetti vengono

per esempio trasmessi periodicamente con la stessa frequenza di tempo da tutti i device, potrebbero essere più soggetti a fenomeni di "sincronizzazione" delle trasmissioni, con conseguenti collisioni, rispetto a sistemi dove i pacchetti vengono trasmessi con un intervallo di tempo dato per esempio da una distribuzione di probabilità uniforme da 0 ad n.

Per verificare questa ipotesi ho eseguito un set di simulazioni configurate in questo modo:

- Numero di device: 25
- Tempo dopo il quale un device trasmette il primo pacchetto dall'istante di tempo iniziale della simulazione: valore estratto casualmente per ogni device, estratto con densità di probabilità uniforme da 0 ad 1,5 secondi
- Intervallo di tempo tra la trasmissione di un pacchetto ed il successivo: valore sempre costante, uguale per tutti i device, e dato dal parametro T

Ho eseguito varie simulazioni variando il parametro T e registrando i valori di output della simulazione. Questi sono alcuni risultati:

Parametro T	Indice di performance
3,5	100
4	73,115
1,5	0,9583

Dove l'indice di performance consiste nella percentuale di pacchetti ricevuti con successo, in rapporto a quelli trasmessi.

Osservando i risultati, si nota una riduzione di performance altissima in corrispondenza di certi valori del parametro T, in corrispondenza dei quali probabilmente molti device del sistema trasmettono contemporaneamente.

Analizziamo invece ora come varia l'indice di performance al variare del numero di device che compongono la rete. Per ottenere dei risultati il più possibile indipendenti da fenomeni di "sincronizzazione" della trasmissione di pacchetti, come osservati nelle simulazioni precedenti, in questo caso ho adottato un intervallo di tempo prima della trasmissione di ogni pacchetto estratto casualmente ad ogni trasmissione su ogni device, con una densità

di probabilità uniforme tra 0 e 1,5 secondi. L'intervallo di tempo medio tra trasmissioni dei pacchetti da parte di ogni device, è quindi di 0,75 secondi.

Questi sono i risultati di alcune simulazioni, al variare del numero di device:

Numero di device	Indice di performance
25	100
400	94,0353
625	82,3

Le percentuali di pacchetti ricevuti con successo, risultano piuttosto basse se confrontate con le percentuali di perdita di pacchetti tipiche delle comunicazioni digitali wireless, anche in presenza di un numero di device piuttosto alto. Ipotizzando che il sistema di comunicazioni opportunistiche in oggetto riesca a tollerare una perdita di pacchetti di qualche punto percentuale, esso risulta quindi scalabile almeno fino a poche centinaia di device.

Occorre comunque tenere in considerazione che i risultati sopra descritti possono essere molto differenti da un potenziale corrispondente sistema reale. Le simulazioni effettuate rappresentano infatti un modello molto approssimato della realtà, non vengono infatti considerati fattori quali eventuali bug nello stack Wifi dei dispositivi utilizzati, interferenze da altri dispositivi e differente propagazione delle onde radio in un ambiente complesso come per esempio una piazza piena di persone.

Tuttavia ritengo che i risultati ottenuti risultino incoraggianti per potenziali ulteriori ricerche sull'impiego di reti Wifi ad hoc in sistemi di comunicazioni opportunistiche ad alta densità tra device mobile.

Capitolo 3

Comunicazioni opportunistiche su dispositivi mobile

In questa sezione della tesi, ci si pone l'obiettivo di sviluppare un sistema basato sulle comunicazioni opportunistiche tra dispositivi mobile come smartphone e tablet, facendo uso delle reti Wifi ad hoc come tecnologia di comunicazione wireless peer-to-peer.

Il mercato dei dispositivi mobile attualmente in commercio è sostanzialmente diviso tra dispositivi Apple basati sul sistema operativo iOS, e dispositivi di vari produttori hardware basati sul sistema operativo Android sviluppato principalmente da Google. Anche se recentemente Apple ha introdotto in iOS il MultipeerConnectivityFramework [9], delle API per la comunicazione wireless peer-to-peer tra device, il suo utilizzo nel campo della ricerca risulta tuttavia limitato dalla natura proprietaria del sistema e dall'assenza di specifiche sul suo funzionamento interno.

Il sistema operativo Android invece, grazie alla sua natura open source dei vari componenti del sistema operativo, incluso il kernel Linux sul quale si basa, può risultare adatto allo sviluppo di un sistema di comunicazioni opportunistiche basato su reti Wifi ad hoc, anche se ufficialmente in teoria non le supporta.

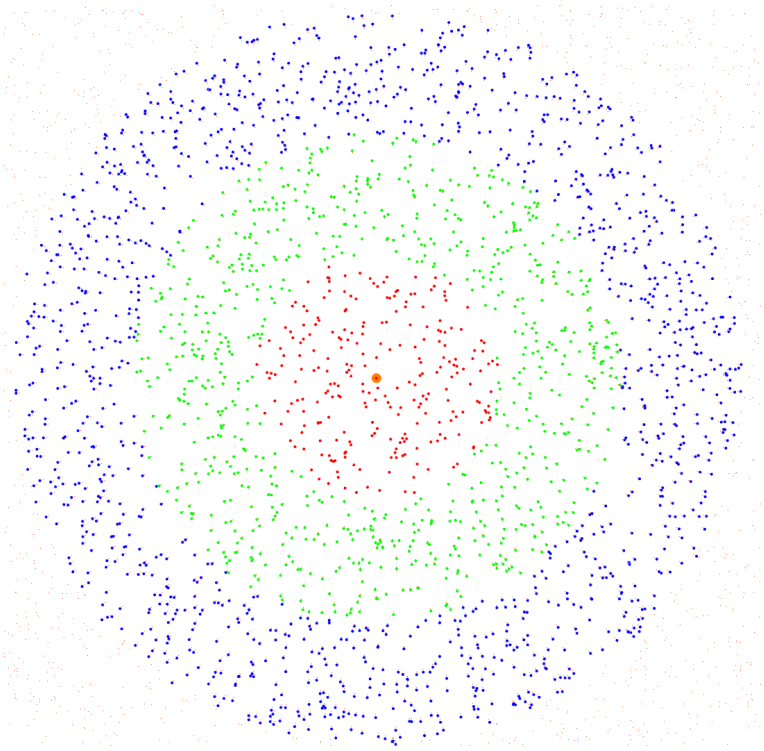
In questa tesi verrà quindi proposto un metodo per sviluppare sistemi di comunicazioni opportunistiche basati su device Android, e verrà sviluppato un software proof of concept per sistemi Android che permette di sviluppare applicazioni basate su campi computazionali costruiti tramite comunicazioni opportunistiche, utilizzando astrazioni fornite dal framework

di programmazione ad eventi Akka Stream.

3.1 Use case: campi computazionali basati su comunicazioni opportunistiche

Come caso d'uso di un possibile sistema mobile basato su comunicazioni opportunistiche, verrà considerato quello della costruzione di campi computazionali su device mobile tramite scambio di messaggi tra di essi, facendo uso di comunicazioni wireless peer-to-peer. Per campo computazionale si intende una struttura dati distribuita che mappa ogni device facente parte di un certo sistema, ad un certo valore di un certo tipo. Uno strumento utile per lo studio dei campi computazionali è Proto [10], un software che permette di descrivere campi computazionali tramite un linguaggio special-purpose e di simularli su un ipotetico sistema di device che possono comunicare con i loro "vicini". Proto fornisce anche delle immagini delle simulazioni, che sono state utilizzate di seguito in questa tesi come rappresentazioni grafiche di campi computazionali.

Un esempio di sistema distribuito basato su campi computazionali, può essere per esempio composto dall'insieme di smartphone delle persone presenti in una piazza affollata, dove ad ogni smartphone viene fatto corrispondere un valore numerico che determina un certo campo. Un esempio di campo computazionale è quello dato dalla distanza del device sul quale vuole essere calcolato il valore del campo, da un altro device fissato, misurata in numero di hop necessari per comunicare tra i due device tramite una rete peer-to-peer che permetta ai device "vicini" di comunicare tra di loro. Il campo formato nel modo appena descritto, viene chiamato "gradiente". [11] [12] [13]



Questo campo può essere utilizzato in applicazioni reali per fornire ai device una misura approssimata di una distanza fisica da un certo luogo, generata in modo distribuito da un semplice scambio di messaggi tra device vicini, effettuato per esempio con tecnologie di comunicazioni opportunistiche basate su reti Wifi ad hoc, come descritto nei capitoli precedenti della tesi.

Nelle prossime sezioni della tesi, verrà analizzato come sviluppare un sistema basato su device mobile Android, che permetta di realizzare applicazioni distribuite che utilizzano campi computazionali come quelli appena descritti, tramite comunicazioni opportunistiche basate su reti Wifi ad hoc.

3.2 Wifi ad hoc sui device Android

In questa sezione della tesi, ci si pone come obiettivo la connessione di device Android ad una rete Wifi ad hoc, con il fine di poter sviluppare applicazioni Android che sfruttano la rete per effettuare comunicazioni opportunistiche con device vicini.

3.2.1 Workaround al mancato supporto ufficiale della modalità Wifi ad hoc su Android

Come già accennato precedentemente, il sistema operativo Android non supporta ufficialmente le reti Wifi in modalità ad hoc. Le reti Wifi più comunemente utilizzate nelle abitazioni e nelle aziende per esempio per accedere ad Internet, sono infatti quelle in modalità infrastructure, e sono perfettamente supportate praticamente da tutti i dispositivi Android in commercio. Le reti Wifi in modalità ad hoc invece hanno molte meno applicazioni pratiche rispetto a quelle di tipo infrastructure, al punto da avere una richiesta di mercato talmente bassa da non giustificarne il supporto ufficiale da parte di Android.

Tuttavia i chipset Wifi comunemente utilizzati in molti device Android in commercio, dispongono in certi casi di driver Linux che supportano anche la modalità Wifi ad hoc. Dato che il sistema operativo Android è basato sul kernel Linux, potrebbe essere quindi possibile per certi device attivare la modalità Wifi ad hoc intervenendo direttamente sullo stesso kernel Linux incluso nel sistema Android. Questo è possibile solo a patto che il kernel incluso nella distribuzione Android utilizzata dal device, includa un driver del chipset Wifi montato nel device che supporti il Wifi ad hoc, e che si riesca in qualche modo ad eseguire comandi privilegiati sul kernel Linux, azione generalmente bloccata dai vari sistemi di sicurezza di Android.

Lo stack Wifi 802.11 interno al kernel Linux, viene generalmente configurato tramite delle applicazioni eseguite in userspace con i privilegi adeguati per poter eseguire l'operazione. Nella maggior parte dei sistemi operativi Linux basati su interfacce grafiche, il compito di configurare le interfacce Wifi viene svolto da un tool grafico che fornisce una comoda GUI all'utente per configurare le impostazioni di rete. Per esempio in molte distribuzioni GNU/Linux viene usato NetworkManager, un servizio integrato con i principali desktop environment come GNOME e KDE. Anche in Android, il sistema fornisce una interfaccia grafica all'utente per configurare le impostazioni Wifi, ed un servizio interno del sistema operativo si occupa di configurare le interfacce di rete nel kernel Linux sulla base dei dati inseriti dall'utente.

Un modo alternativo per configurare lo stack Wifi nei sistemi Linux, è di usare le utility da linea di comando `iw` ed `ifconfig`.

Queste utility agiscono, come i tool precedentemente descritti, sull'in-

terfaccia di configurazione standard dei dispositivi di rete del kernel Linux, fornendo tra l'altro molte funzionalità e parametri di configurazione generalmente non accessibili tramite i tool grafici di configurazione delle reti Wifi. Queste utility da linea di comando, funzionano in generale in tutti i sistemi basati su kernel Linux con uno stack Wifi standard. Dato che Android è basato sul kernel Linux, è quindi probabile che sia possibile configurare il driver Wifi di certi dispositivi Android utilizzando utility come `iw` ed `ifconfig`.

Nelle distribuzioni Android installate nei device comunemente in commercio, il sistema operativo dispone di numerose protezioni che impediscono di far eseguire operazioni privilegiate al codice applicativo installabile dall'utente, allo scopo di impedire ad eventuale malware installato inconsapevolmente di eseguire operazioni dannose. Tra le varie operazioni che sono impedite alle applicazioni Android installabili dall'utente, rientra l'esecuzione di un processo Unix come utente root, dato che ciò permetterebbe ovviamente ad una applicazione malevola di assumere il controllo di tutto il sistema.

Le utility `iw` ed `ifconfig` necessitano però di permessi di root per poter configurare lo stack Wifi del kernel Linux, dato che si tratta di una operazione con un impatto piuttosto forte sul funzionamento di tutto il sistema. Il loro utilizzo è dunque impossibile nelle normali distribuzioni di Android presenti nella maggior parte dei device in commercio. Fortunatamente però per alcuni device Android esistono dei metodi per poter aggirare queste protezioni, effettuando una procedura comunemente chiamata rooting, che ha come obiettivo l'installazione di un software analogo al comando Unix `sudo`, che permette di eseguire processi con i permessi di root con l'autorizzazione da parte dell'utente.

Le procedure di rooting, pur essendo legali, sono spesso scoraggiate ed ostacolate da alcuni produttori di device, ed in molti casi per effettuarle è necessario ricorrere ad exploit che sfruttano vulnerabilità di sicurezza del sistema operativo distribuito con il device. Altri produttori invece prevedono ufficialmente la possibilità di effettuare il rooting dei propri dispositivi, tramite dei tool forniti dai produttori stessi. È questo il caso dei dispositivi Nexus distribuiti da Google, per i quali tramite l'utility `fastboot` è possibile sbloccare il bootloader ed installare un software di gestione dell'accesso root come SuperSU [14] .

Dopo aver effettuato il rooting di un dispositivo, è quindi possibile ese-

guire dei processi Unix come utente root utilizzando l'eseguibile `sudo` installato nel sistema Android dal software di gestione dell'accesso root utilizzato, per esempio SuperSU. Sarà dunque possibile eseguire utility come `iw` ed `ifconfig` con permessi di root.

3.2.2 Test di attivazione della modalità ad hoc su dispositivi Nexus

Avendo a disposizione degli smartphone Nexus 5 e tablet Nexus 7 2013, entrambi con sistema operativo Android Lollipop 5.0, ho effettuato alcune sperimentazioni al fine di attivare su di essi la modalità Wifi ad hoc, ed effettuare uno scambio di dati di prova tra di essi attraverso la rete ad hoc creata.

Per ogni device ho quindi tentato di eseguire la stessa procedura volta all'attivazione della modalità Wifi ad hoc, che ora descriverò. Dopo aver eseguito la procedura di rooting del dispositivo, è ora possibile eseguire una shell con permessi di root, per esempio con un emulatore di terminale sotto forma di applicazione Android come ConnectBot [15], oppure utilizzando il tool di debugging `adb` incluso nell'SDK Android ufficiale. Dopo aver aperto una shell come utente non privilegiato, è infatti possibile eseguire una shell di root semplicemente eseguendo `su`.

Una volta ottenuta una shell di root, è ora possibile tentare di utilizzare utility come `iw` ed `ifconfig` per controllare a basso livello lo stack Wifi del dispositivo Android. Queste utility però non sono generalmente incluse nei sistemi Android tra i file di sistema, quindi è necessario reperire i file eseguibili delle utility `iw` ed `ifconfig` e copiarli nel device.

I file eseguibili possono essere ottenuti compilando i rispettivi sorgenti C con il toolchain fornito nell'Android NDK (Native Development Kit), per l'architettura utilizzata dal device, `armeabi` nel caso dei Nexus in mio possesso. Per semplicità, è anche possibile reperire gli eseguibili `iw` ed `ifconfig` già compilati dalla comunità di alcuni progetti open source, come Serval Project [16] che li include nella propria applicazione Android [17]. Una volta copiati nel filesystem del device Android gli eseguibili `iw` ed `ifconfig` , è ora possibile lanciarli dalla shell di root per tentare di attivare la modalità Wifi ad hoc.

Prima di tutto è però necessario disattivare dall'interfaccia utente Android le funzionalità Wifi del device, in modo da non far gestire lo stack

802.11 del kernel da parte dei servizi di configurazione Wifi del sistema Android, che potrebbero interferire con le impostazioni effettuate manualmente tramite l'utility `iw`.

Ora è possibile lanciare i seguenti comandi:

```
ifconfig wlan0 up
ifconfig wlan0 192.168.0.1 netmask 255.255.255.0
```

in questo modo viene configurato il protocollo IP per l'interfaccia di rete `wlan0` del kernel con un IP statico.

Come spiegato in precedenza, in sistemi di comunicazioni opportunistiche basati su reti Wifi ad hoc, non è possibile far assegnare gli indirizzi IP dei device ad un server DHCP centrale, quindi in questa sperimentazione ho scelto di utilizzare indirizzi IP assegnati staticamente ad ogni device.

Ora è possibile eseguire l'utility `iw` per impostare lo stack 802.11 dell'interfaccia `wlan0` in modalità ad hoc:

```
iw dev wlan0 set type ibss
iw dev wlan0 ibss join <ESSID> <frequenza in Hz> <BSSID>
```

Il parametro `ibss` identifica la modalità IBSS (Independent Basic Service Set), un altro nome con cui viene chiamata la modalità Wifi ad hoc nelle specifiche dello standard 802.11.

Nel secondo comando, vanno specificati i seguenti parametri:

- ESSID: il "nome" della rete Wifi ad hoc che si vuole creare
- frequenza in Hz: la frequenza centrale del canale Wifi da utilizzare. Se si vuole utilizzare per esempio il canale 6 dello standard 802.11, occorre inserire 2437.
- BSSID: parametro opzionale che forza l'uso di un BSSID manuale, utile per evitare alcuni problemi dati dal merge di reti ad hoc precedentemente separate, descritti in precedenza. [3] Per i test che ho effettuato, ho scelto di utilizzare questo BSSID: `02:ca:ff:ee:ba:be`

Utilizzando dei parametri di esempio, possiamo lanciare il secondo comando in questo modo:

```
iw dev wlan0 ibss join adhoctest 2437 02:ca:ff:ee:ba:be
```

A questo punto, se tutto ha funzionato correttamente, il device Android dovrebbe essere connesso alla rete Wifi adhoc "adhoctest", con l'indirizzo IP 192.168.0.1. È ora possibile testare il corretto funzionamento della rete per esempio connettendosi con un altro device alla stessa rete, e provando a scambiare dati per esempio tramite il comando ping.

Dai test che ho effettuato su device come il Nexus 5, ho notato che negli smartphone che possiedono connettività ad Internet tramite rete cellulare, il sistema Android si connette alla rete LTE/3G dato che pensa che il Wifi sia disattivato, e di conseguenza configura lo stack IP del kernel Linux per funzionare con la connessione ad Internet tramite rete cellulare, in modo da bloccare il transito dei pacchetti con l'interfaccia di rete Wifi.

Per risolvere questo problema, è possibile eseguire questo comando:

```
ip rule add table main
```

tramite questo comando viene modificata la configurazione delle regole IP del kernel Linux in modo da aggirare alcune limitazioni configurate dal sistema Android probabilmente per motivi di sicurezza. Dopo aver eseguito questo comando, il device Android è in grado di far coesistere la rete Wifi ad hoc e la connessione ad Internet tramite LTE/3G, permettendo quindi alle applicazioni di comunicare contemporaneamente sia tramite la rete ad hoc che tramite Internet.

Per effettuare dei test sul corretto funzionamento della rete Wifi ad hoc appena configurata, è possibile utilizzare il comando ping, già incluso nel sistema Android ufficiale dei dispositivi Nexus, specificando come argomento l'indirizzo IP di un altro device con il quale si vuole testare la possibilità di comunicare:

```
ping 192.168.0.2
```

Dai risultati di alcuni test che ho effettuato, due device Nexus 5 riescono a comunicare correttamente con la procedura sopra descritta, mentre invece ho riscrontrato alcuni problemi in una rete formata da un Nexus 7 ed un Nexus 5, in particolare ho riscontrato un alto tasso di perdita dei pacchetti inviati.

I problemi che ho riscontrato potrebbero essere dovuti a dei bug o limitazioni nei driver del chipset 802.11 utilizzati nei rispettivi device. I Nexus 5 infatti montano il chipset Wifi Broadcom BCM4339 [18] ed utilizzano un driver Broadcom, mentre i Nexus 7 2013 usano un Qualcomm Atheros WCN3660 tramite un driver Qualcomm.

3.2.3 Sviluppo di un'applicazione Android per l'attivazione automatizzata di una rete ad hoc

Nel punto precedente è stata descritta una procedura manuale per connettere un dispositivo Android ad una rete Wifi ad hoc, lanciando manualmente dei file eseguibili da una shell di root. Se si vuole realizzare un sistema reale che fa uso delle reti Wifi ad hoc, occorre però automatizzare la procedura vista precedentemente, in modo da renderla semplice da usare per l'utente finale del sistema.

Analizziamo ora come sviluppare un semplice prototipo di applicazione Android che esegue l'attivazione della modalità Wifi ad hoc, e successivamente invia e riceve pacchetti UDP di broadcast tramite dei socket, in modo da testare il corretto funzionamento della rete. Essendo solamente una semplice applicazione di prova, per semplicità essa sarà composta da una sola Activity con all'interno un Fragment, che contiene tutta la logica di attivazione della modalità ad hoc e dello scambio di pacchetti UDP.

Dato che l'applicazione deve eseguire come root le utility `iw` ed `ifconfig`, occorre:

- Includere i binari di `iw` ed `ifconfig` nel file apk finale dell'applicazione.

  Uno dei metodi per includere eseguibili Unix all'interno di una applicazione Android, consiste nell'inserirli nel progetto Gradle di Android Studio nei seguenti path:

  ```
  lib/armeabi/lib_iw_.so
  lib/armeabi/lib_ifconfig_.so
  ```

  In questo modo vengono inclusi nell'apk dai build script, e dopo l'installazione nel device Android, essi sono accessibili con permessi di esecuzione dall'applicazione al path:

  ```
  /data/data/<nome-package-java>/lib/
  ```

- Includere tra le dipendenze Gradle una libreria per facilitare l'esecuzione di comandi come utente root. Una di queste librerie è `libsuperuser`, utilizzabile aggiungendo la seguente dipendenza nel file build.gradle:

```
compile 'eu.chainfire:libsuperuser:1.0.0.+'
```

È ora possibile realizzare un semplice metodo Java che esegue i comandi descritti precedentemente per la connessione ad una rete Wifi ad hoc:

```
private void startWifiAdhoc(String myIpAddr, String netmask, String essid, int
    frequency){

  // Disable Android Wifi management, I need to control the Linux Wifi stack by myself!
  WifiManager wifiManager = (WifiManager) getActivity().getSystemService(Context.
    WIFI_SERVICE);
  wifiManager.setWifiEnabled(false);

  try {
    Thread.sleep(4000);
  } catch (InterruptedException e) {}

  // I file "lib_NOME_.so" contenuti in questa directory in realta' sono degli
    eseguibili Unix (ELF).
  // Li ho chiamati in quel modo perche' cosi' vengono trattati "correttamente" dal
    build system di Android
  String exeDir = "/data/data/" + getActivity().getPackageName() + "/lib/";

  // Execute as root the commands to enable adhoc wifi
  List<String> execResults = Shell.SU.run(new String[]{
        exeDir + "lib_ifconfig_.so wlan0 up",
        exeDir + "lib_ifconfig_.so wlan0 " + myIpAddr + " netmask " + netmask,
        exeDir + "lib_iw_.so dev wlan0 set type ibss",
        exeDir + "lib_iw_.so dev wlan0 ibss join " + essid + " " + frequency + " " +
    BSSID,
                "ip rule add table main" // this is needed to make the adhoc network
    working simultaneously with LTE/3G/2G connection.
                                          // This probably make everything bypass some
    Android network access limitations/permissions,
                                          // but I think it's sufficiently secure anyway
     for use case.
  });

  for(String msg: execResults){
    Log.i("adhoc", msg);
  }
}
```

Per effettuare un testing veloce del funzionamento della rete Wifi ad hoc appena creata, è possibile includere nell'applicazione una funzionalità per inviare dei messaggi periodici o manuali tramite un socket UDP all'indirizzo di broadcast, e contemporaneamente riceverli con un altro socket UDP e mostrarli a video.

Questa è una semplice implementazione che fa uso di AsyncTask e Thread, da utilizzare dentro una Activity o un Fragment:

```
private class UdpBroadcastReceiverTask extends AsyncTask<Void, String, Void> {

  @Override
  protected Void doInBackground(Void... params) {

    // Non so se il MulticastLock riguarda anche i pacchetti di broadcast... nel dubbio
      lo acquisisco
    WifiManager wifi = (WifiManager) getActivity().getSystemService(Context.
     WIFI_SERVICE);
    WifiManager.MulticastLock multicastLock = wifi.createMulticastLock("
     opportunisticnet");
    multicastLock.acquire();

    try {
```

```
12        publishProgress("starting udp packets receiver!");
13
14        DatagramSocket recvSock = new DatagramSocket(UDP_PORT, InetAddress.getByName("
      0.0.0.0"));
15        recvSock.setBroadcast(true);
16
17        while (true) {
18          byte[] data = new byte[15000];
19          DatagramPacket packet = new DatagramPacket(data, data.length);
20          recvSock.receive(packet);
21          String msg = new String(packet.getData());
22
23          publishProgress(msg);
24          }
25
26        } catch (Exception e){
27           e.printStackTrace();
28        }
29
30        multicastLock.release();
31
32        return null;
33      }
34
35      @Override
36      protected void onProgressUpdate(String... values) {
37        super.onProgressUpdate(values);
38
39        logTextView.setText(values[0]+"\n" + logTextView.getText().toString());
40      }
41    }
42
43
44
45    // Questo thread sarebbe stato da fare unito al task sotto...
46    private class BroadcastSenderThread extends Thread {
47
48      private DatagramSocket sendSock;
49
50      public BroadcastSenderThread() throws SocketException {
51        sendSock = new DatagramSocket();
52        sendSock.setBroadcast(true);
53      }
54
55      @Override
56      public void run() {
57        super.run();
58
59        while(!isInterrupted()) {
60          try {
61            Thread.sleep(10000);
62
63            byte[] data = ("ciao sono " + myIpEditText.getText().toString()).getBytes();
64            DatagramPacket packet = new DatagramPacket(data, data.length, InetAddress.
      getByName(BROADCAST_ADDRESS), UDP_PORT);
65            sendSock.send(packet);
66
67          } catch (Exception e){
68            e.printStackTrace();
69          }
70        }
71      }
72    }
73
74
75
76    private class BroadcastSenderTask extends AsyncTask<Void, Void, Void> {
77
78      private String msg;
79
80      public BroadcastSenderTask(String msg){
81        this.msg = msg;
82      }
83
84      @Override
85      protected Void doInBackground(Void... params) {
```

```
86        try {
87          // Invio in broadcast il messaggio scritto dall'utente
88          DatagramSocket sendSock = new DatagramSocket();
89          sendSock.setBroadcast(true);
90          byte[] data = msg.getBytes();
91          DatagramPacket packet = new DatagramPacket(data, data.length, InetAddress.
      getByName(BROADCAST_ADDRESS), UDP_PORT);
92          sendSock.send(packet);
93        } catch(Exception e){
94          e.printStackTrace();
95        }
96        return null;
97      }
98  }
```

Le precedenti due parti di codice, possono essere inserite in una classe chiamata NetStatusFragment:

```
1   package it.unibo.opportunisticnet.android;
2
3   import android.content.Context;
4   import android.net.wifi.WifiManager;
5   import android.os.AsyncTask;
6   import android.os.Bundle;
7   import android.support.v4.app.Fragment;
8   import android.text.method.ScrollingMovementMethod;
9   import android.util.Log;
10  import android.view.LayoutInflater;
11  import android.view.View;
12  import android.view.ViewGroup;
13  import android.widget.Button;
14  import android.widget.EditText;
15  import android.widget.TextView;
16  import eu.chainfire.libsuperuser.Shell;
17
18  import java.net.*;
19  import java.util.List;
20
21
22  public class NetStatusFragment extends Fragment {
23
24    private final static int UDP_PORT = 10000;
25    private final static String BROADCAST_ADDRESS = "192.168.0.255";
26
27    private final static String BSSID = "02:ca:ff:ee:ba:be"; // un qualsiasi MAC address
        che inizi con 02:
28
29    private TextView logTextView;
30    private EditText myIpEditText;
31    private EditText msgEditText;
32
33    public NetStatusFragment(){
34    }
35
36    @Override
37    public View onCreateView(LayoutInflater inflater, ViewGroup container,
38                             Bundle savedInstanceState) {
39      return inflater.inflate(R.layout.fragment_net_status, container, false);
40    }
41
42
43    @Override
44    public void onViewCreated(View view, Bundle savedInstanceState) {
45      super.onViewCreated(view, savedInstanceState);
46
47      myIpEditText = (EditText) getActivity().findViewById(R.id.myIpEditText);
48
49      Button adhocStartButton = (Button) getActivity().findViewById(R.id.adhocStartButton
        );
50      adhocStartButton.setOnClickListener(new View.OnClickListener() {
51        @Override
52        public void onClick(View v) {
53
54          startWifiAdhoc(myIpEditText.getText().toString(), "255.255.255.0", "mesh",
      2437); // canale 6 = 2437, canale 40 = 5200
```

```
55
56          // Faccio partire la demo di scambio dei messaggi
57          try {
58            new BroadcastSenderThread().start();
59          } catch (Exception e){
60            e.printStackTrace();
61          }
62
63          new UdpBroadcastReceiverTask().executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR
        );
64        }
65      });
66
67      logTextView = (TextView) getActivity().findViewById(R.id.logTextView);
68      logTextView.setMovementMethod(new ScrollingMovementMethod());
69
70      msgEditText = (EditText) getActivity().findViewById(R.id.msgEditText);
71
72      Button sendMsgButton = (Button) getActivity().findViewById(R.id.sendMsgButton);
73      sendMsgButton.setOnClickListener(new View.OnClickListener() {
74        @Override
75        public void onClick(View v) {
76          new BroadcastSenderTask(msgEditText.getText().toString()).executeOnExecutor(
        AsyncTask.THREAD_POOL_EXECUTOR);
77        }
78      });
79    }
80
81    private void startWifiAdhoc(String myIpAddr, String netmask, String essid, int
        frequency){
82    ...
83    }
84
85    private class UdpBroadcastReceiverTask extends AsyncTask<Void, String, Void> {
86    ...
87    }
88
89    private class BroadcastSenderThread extends Thread {
90    ...
91    }
92
93    private class BroadcastSenderTask extends AsyncTask<Void, Void, Void> {
94    ...
95    }
96  }
```

Di seguito anche la risorsa XML dell'interfaccia utente del NetStatusFragment sopra descritto:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:orientation="vertical">

<LinearLayout
```

```
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">


  <TextView
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:textAppearance="?android:attr/textAppearanceMedium"
     android:text="My IP addr:"
     android:id="@+id/textView"/>

  <EditText
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:id="@+id/myIpEditText"
     android:text="192.168.0.1"
     android:layout_weight="1"/>

  <Button
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Start Wifi Ad-Hoc"
     android:id="@+id/adhocStartButton" />

</LinearLayout>


<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/msgEditText"
```

```
        android:hint="messaggio da inviare"
        android:text="prova"
        android:layout_weight="1"/>

    <Button
        android:id="@+id/sendMsgButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send msg"/>

</LinearLayout>

<TextView
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:textAppearance="?android:attr/textAppearanceSmall"
   android:text="Log:"
   android:id="@+id/logTextView"
   android:layout_marginTop="20dp"
   android:scrollbars="vertical" />

</LinearLayout>
```

Tramite l'Activity dell'applicazione che contiene il NetStatusFragment appena descritto, è possibile inserire nella `myIpEditText` l'indirizzo IP che verrà assegnato al device nel quale è eseguita l'applicazione, e successivamente premere il bottone "Start Wifi Ad-Hoc".

A questo punto l'applicazione, dopo aver ottenuto l'accesso root, si occuperà di attivare la modalità Wifi ad hoc, eventualmente in parallelo alla connessione ad Internet tramite la rete cellulare, se abilitata.

Inoltre inizierà ad inviare alla rete periodicamente dei messaggi di debug tramite pacchetti UDP in broadcast, che possono essere ricevuti e visualizzati nella `logTextView`. È anche possibile inviare dei messaggi manualmente con del testo a piacere, inserendoli nella `msgEditText`.

3.3 Costruzione di campi computazionali tramite Akka Stream

Nel capitolo precedente della tesi, è stato spiegato come realizzare una applicazione Android che riesca ad attivare la modalità Wifi ad hoc sui dispositivi che la supportano.

In questo capitolo vedremo invece come sfruttare i risultati appena descritti per sviluppare un sistema che riesca a costruire campi computazionali tramite comunicazioni opportunistiche tra dispositivi Android.

Una volta ottenuta l'attivazione della rete Wifi ad hoc, l'applicazione che si dovrà costruire consisterà in un semplice software basato su tradizionali comunicazioni di rete. Esso sarà infatti basato sull'invio e la ricezione di dati attraverso pacchetti UDP di broadcast, attraverso i quali l'applicazione dovrà essere in grado di costruire dei campi computazionali.

Pur utilizzando a basso livello dei semplici socket UDP, la logica sovrastante dell'applicazione può risultare piuttosto complessa, in quanto deve occuparsi di gestire la ricezione e l'invio possibilmente concorrente di pacchetti da diversi device, tenere aggiornata una struttura dati in memoria dello stato della logica di costruzione dei campi computazionali, aggiornamento dell'interfaccia grafica, e gestione di varie situazioni di errore come la perdita di pacchetti.

Nello sviluppo di applicazioni di questa tipologia basate sostanzialmente sull'elaborazione di flussi di eventi, può risultare molto utile l'impiego di framework ad eventi, per rendere lo sviluppo di software intrinsecamente concorrente più semplice, efficiente e privo di errori.

Nella prossima sezione della tesi, verranno analizzati i framework Rx ed Akka Stream, e successivamente verrà descritto lo sviluppo di un applicazione Android basata su Akka Stream, scritta in linguaggio Scala, in grado di costruire campi computazionali sulla base di comunicazioni di rete tramite pacchetti UDP.

3.3.1 Introduzione ai framework di reactive programming

Negli ultimi anni si sta assistendo ad un trend di sempre maggiore diffusione dei cosiddetti framework di reactive programming, o programmazione orientata agli eventi, utilizzati sopratutto negli ambiti dell'industria del

software legati al paradigma della programmazione funzionale. Questi framework semplificano lo sviluppo di software basato su computazioni svolte come reazione a degli eventi, generati generalmente da operazioni di IO. Gli eventi a cui una applicazione deve poter reagire possono essere per esempio l'evento di click di un bottone della GUI, oppure l'arrivo di un pacchetto dalla rete.

Le operazioni di IO all'interno di un software, sono state tradizionalmente gestite in due modi:

- **IO sincrono** - l'operazione di IO viene effettuata invocando per esempio dei metodi come send() e receive(), che bloccano il flusso di esecuzione finché l'operazione non è terminata.

  Questa soluzione ha il vantaggio di essere molto familiare per chi è abituato al paradigma di programmazione imperativo tradizionale, ma causa problemi dati dalla necessità di usare molti thread per gestire tutte le computazioni necessarie. Questo oltre a causare potenziali problemi di performance, rende necessario l'uso di tecniche di locking nel caso in cui più di un thread debba operare concorrentemente su uno stato condiviso.

  Nello sviluppo di software complesso che fa uso di lock espliciti, è estremamente facile commettere errori che possono causare bug difficili da essere scoperti, a causa del non determinismo dell'esecuzione di software concorrenti.

- **IO asincrono tramite callback** - nel caso delle operazioni di IO asincrone, metodi come per esempio receive() o send() terminano immediatamente, restituendo il flusso di controllo al chiamante, e l'operazione di IO vera e propria avviene in background. Quando l'operazione termina, l'applicazione viene notificata del risultato con l'esecuzione di un callback, del codice che svolge la funzione di handler dell'evento.

  Questa è la soluzione tradizionalmente usata nelle GUI. Pur permettendo di sviluppare software effettivamente asincrono, ha la grossa limitazione di rendere il codice difficile da comprendere e scarsamente manutenibile, appena si cerca di sviluppare una applicazione non triviale.

L'utilizzo di operazioni asincrone tramite callback non permette infatti di fare a meno di dover accedere concorrentemente ad uno stato in memoria condivisa. Si ricade quindi negli stessi problemi di gestione della shared memory tramite tecniche di locking, già visti nel caso dell'IO sincrono.

I framework di reactive programming, cercano di risolvere i problemi e le limitazioni delle tecniche di programmazione asincrona basata sui callback, utilizzando concetti ed astrazioni provenienti dal mondo della programmazione funzionale. I framework che analizzerò sono Rx (Reactive Extensions) ed Akka, che sono infatti nati in ambiti di ricerca legata a linguaggi di programmazione funzionale, come Scala ed Haskell.

Il concetto derivante dalla programmazione funzionale di cui questi framework si avvalgono maggiormente, è sicuramente quello degli oggetti immutabili, cioè dell'utilizzo di strutture dati in memoria che non possono essere modificate, tranne che creandone una copia. Le strutture dati immutabili permettono di far mantenere ad un flusso di esecuzione uno snapshot sempre consistente di una certa struttura dati presente in memoria. Le strutture dati immutabili possono quindi essere accedute tranquillamente in modo concorrente da più thread, senza dover utilizzare alcuna tecnica di locking. In molti framework basati sul message-passing, le strutture dati immutabili vengono utilizzate per i messaggi scambiati tra le varie entità del sistema, sfruttando la loro immutabilità per evitare side-effect. Le strutture dati immutabili più comuni, come liste, set e map, sono fornite nativamente da linguaggi funzionali come Scala.

I framework di reactive programming principalmente utilizzati, possono essere divisi in due categorie, a seconda dell'astrazione di base che usano:

- **Attori**: il framework più rappresentativo di questa categoria è sicuramente la versione tradizionale di Akka, che permette di rappresentare le entità che compongono un certo sistema software come degli attori, una sorta di oggetti che mantengono uno stato e che reagiscono a dei messaggi che possono essergli inviati da altri attori.

  Il modello ad attori permette di risolvere alcune problematiche della programmazione basata su callback: invece di notificare un evento tramite un callback che potrebbe incorrere in corse critiche accedendo a della memoria condivisa utilizzata concorrentemente da altri thread,

in un sistema basato sul modello ad attori si può semplicemente notificare un evento inviando un messaggio ad un certo attore. Dato che i messaggi scambiati tra gli attori sono immutabili, e le computazioni all'interno dei singoli attori avvengono in modo non concorrente, il modello ad attori permette di sviluppare software asincrono e concorrente senza dover utilizzare direttamente memoria condivisa da più thread, con tutta la complessità ed i rischi che ne derivano.

Uno dei più grossi problemi dei framework basati sul modello ad attori è dato dal fatto che un attore può inviare continuamente messaggi ad un altro attore, anche se l'attore ricevente non è in grado di processarli ad una velocità maggiore o uguale a quella con cui gli vengono inviati. I messaggi inviati ad un attore ed in attesa di essere processati da esso, rimangono nella sua mailbox, che può quindi crescere in modo illimitato, occupando tutta la memoria del computer su cui viene eseguito il software.

Questa problematica può essere in teoria risolta elaborando dei protocolli di comunicazione tra gli attori che notificano all'attore sorgente quando deve smettere di inviare messaggi all'attore destinazione. Tuttavia questi protocolli sono notevolmente complessi, e non è efficiente doverli reimplementare in tutte le situazioni in cui sono necessari. Per evitare questo, nelle versioni più recenti di Akka, è stata introdotta l'astrazione di stream, come descritto nel punto seguente.

- **Stream**: tramite questa astrazione, le computazioni in un sistema software vengono modellate come delle trasformazioni applicate in modo asincrono ad un flusso (stream) di messaggi immutabili.

  Il primo framework che ha reso popolare l'astrazione di stream è sicuramente Rx (Reactive Extensions), nato inizialmente alla Microsoft come funzionalità del linguaggio C# e successivamente portato sulla piattaforma Java da Netflix.

  In Rx, l'astrazione di stream viene rappresentata con la classe Observable, che rappresenta appunto uno stream di cui è possibile osservare l'arrivo di nuovi messaggi di tipo T. Ad un Observable è possibile sottoscrivere un Subscriber, che eseguirà una azione all'arrivo di un nuovo messaggio. La particolarità che differenzia il modello di programmazione di Rx da quello ad eventi tradizionale basato sui

callback, consiste nel fatto che gli Observable di Rx godono di certe proprietà derivanti dai monad, una tipologia di struttura utilizzata nella programmazione funzionale. Una delle proprietà più importanti degli Observable è la loro composizionalità, cioè la possibilità di poter comporre degli operatori che trasformano degli Observable stream ottenendo altri Observable.

Il concetto di stream osservabile su cui è basato Rx, è stato poi preso come fonte di ispirazione per sviluppare altri framework, tra cui Akka Stream, una estensione del framework Akka basato sugli attori precedentemente descritto.

In Akka Stream, viene fornita agli utilizzatori del framework una API concettualmente simile a quella offerta da Rx, ma viene implementata internamente al framework utilizzando degli attori, a differenza di Rx dove gli operatori vengono eseguiti in thread pool e sono implementati tramite tecniche di accesso concorrente a della memoria condivisa.

3.3.2 Comunicazioni opportunistiche su Android tramite Scala ed Akka Stream

Analizzeremo ora come sviluppare una applicazione Android basata su comunicazioni opportunistiche effettuate tramite scambio di pacchetti UDP di broadcast, facendo uso di framework di reactive programming basato su stream per gestire la comunicazione via rete peer-to-peer con altri device.

Tra Rx ed Akka Stream, i due principali framework per la gestione di stream, ho deciso di utilizzare Akka Stream, per via della sua maggiore espressività nella costruzione di stream di eventi, e delle funzionalità di serializzazione e comunicazione via rete integrate. Akka Stream è un framework basato su Scala, un linguaggio di programmazione funzionale interoperabile con la piattaforma Java. Il compilatore Scala converte infatti il codice sorgente Scala in bytecode Java, eseguibile da una qualsiasi JVM (Java Virtual Machine) standard. Inoltre all'interno del codice Scala è possibile utilizzare in modo elegante delle classi Java, permettendo quindi al software scritto in Scala di interoperare con software realizzati in Java, molto diffusi per esempio in ambito enterprise.

Dato che i sorgenti Scala vengono compilati in bytecode Java standard, esso può essere in teoria eseguito da ogni implementazione di Java Virtual

Machine, incluse quelle utilizzate dal sistema Android: Dalvik VM e la più recente ART (Android RunTime). È quindi in teoria possibile sviluppare delle applicazioni Android utilizzando codice Scala, che al suo interno utilizza classi Java delle API di Android. La compilazione di codice Scala non è tuttavia supportata dal sistema di build ufficiale di Android basato su Gradle, e dagli ambienti di sviluppo come Android Studio supportati direttamente da Google.

Per utilizzare codice Scala in applicazioni Android, è quindi necessario utilizzare un altro sistema di build, per esempio SBT (Scala Build Tool) unito al plugin `android-sdk-plugin` [19]. Tramite questi strumenti, è possibile sviluppare applicazioni Android in linguaggio Scala utilizzando degli IDE con funzionalità avanzate, come IntelliJ IDEA.

L'applicazione Android che ci si pone come obiettivo di sviluppare, consiste sostanzialmente in una Activity che alla sua creazione si occupa di istanziare un backend, basato sul framework Akka Stream, che gestisce il sistema di comunicazioni opportunistiche vero e proprio. Questo backend avrà la responsabilità di gestire la ricezione di pacchetti UDP da altri device, per esempio aggiornando un suo stato interno, e di inviare pacchetti UDP quando richiesto dalla sua logica interna. Il frontend dell'applicazione Android, rappresentato dal codice di gestione dell'interfaccia utente racchiuso dall'Activity, dovrà interagire con il backend, per esempio ricevendo da esso degli aggiornamenti di stato da visualizzare all'utente.

Essendo il backend basato su Akka Stream, esso è rappresentabile come un componente che accetta in ingresso uno stream di eventi, li elabora internamente, ed emette in output un altro stream di eventi. Lo stream di ingresso corrisponde dati ricevuti dalla rete, mentre in uscita si avrà uno stream di dati da inviare alla rete, ed eventualmente uno stream di aggiornamenti di stato destinati alla GUI dell'applicazione. La business logic dell'applicazione che si sta sviluppando, cioè le funzionalità di alto livello che dovrà svolgere l'applicazione appoggiandosi alle comunicazioni opportunistiche, potrà quindi essere inserita all'interno di questo stream.

Dato che qualunque sia la business logic che si vuole utilizzare, il sistema necessiterà comunque di gestire l'invio e ricezione di dati con comunicazioni opportunistiche tramite pacchetti UDP di broadcast, è possibile sviluppare un componente dello stream completamente indipendente da una specifica logica di dominio. Utilizzando le astrazioni fornite dal framework Akka Stream, questo componente dovrà fornire agli utilizzatori una Source di

messaggi ricevuti dalla rete, ed una Sink di messaggi da inviare in broadcast alla rete.

I messaggi potranno essere rappresentati come delle case class, una feature del linguaggio Scala che permette di rappresentare strutture dati immutabili, che implementano un unico trait che svolge la funzione di interfaccia:

```
1 package it.unibo.akkaoppnet
2
3 /**
4  * A message that can be sent over the Ad-Hoc network
5  */
6 trait OppNetMessage extends java.io.Serializable
7
8 case class TestMsg(msg: String) extends OppNetMessage
```

Dato che i messaggi devono poter essere inviati via rete, essi devono essere in qualche modo serializzabili. Estendendo l'interfaccia java.io.Serializable, è possibile serializzarli tramite la feature di serializzazione integrata in Java.

Il TestMsg è una implementazione di messaggio OppNetMessage, che ha come "payload" una stringa di testo.

Una volta definita l'interfaccia che rappresenta i messaggi, è possibile implementare un componente, in questo caso chiamato BroadcastPacketsTransceiver, che fornisce delle implementazioni di una Source ed una Sink con le quali poter costruire un flusso di invio e ricezione di messaggi dalla rete:

```
1 package it.unibo.akkaoppnet
2
3 import java.net.{DatagramPacket, InetAddress, InetSocketAddress}
4 import java.nio.channels.DatagramChannel
5 import java.util
6
7 import akka.actor.ActorSystem
8 import akka.serialization.SerializationExtension
9 import akka.stream.{Supervision, ActorOperationAttributes}
10 import akka.stream.scaladsl.{Flow, Sink, Source}
11 import it.unibo.akkaoppnet.BroadcastPacketsTransceiver.ReceivedMessage
12
13 import scala.concurrent.{ExecutionContext, Future}
14
15
16 /**
17  * This stage of the stream must get a serialized message ready to be sent, and
      transmit it as an UDP packet to the broadcast IP address,
18  * using a random delay between each packet and another, in order to maximize the
      efficiency of an ad-hoc network.
19  */
20 class BroadcastPacketsTransceiver
21   (broadcastIpAddr: String, port: Int)
22   (implicit system: ActorSystem) {
23
24   val addr = InetAddress.getByName(broadcastIpAddr)
25
26   val channel = DatagramChannel.open()
27   val socket = channel.socket()
28   socket.setBroadcast(true)
29   socket.bind(new InetSocketAddress(port))
30
31   val serialization = SerializationExtension(system)
32
```

```
33   implicit val context: ExecutionContext = system.dispatcher
34
35
36   val receiveSource: Source[ReceivedMessage, Unit] = {
37     Source.repeat(Unit)
38     .map[DatagramPacket] ( boh => {
39       val receiveBuf = new Array[Byte](15000)
40       val receivePacket =
41     new DatagramPacket(receiveBuf, receiveBuf.length)
42       socket.receive(receivePacket)
43       receivePacket
44     })
45     .mapAsync(4)(packet => {
46       Future {
47         try{
48           val msg = serialization.deserialize(
49         util.Arrays.copyOfRange(
50         packet.getData, 0, packet.getLength
51       ),
52       classOf[OppNetMessage]).get
53           println("pacchetto recv: " + msg)
54           ReceivedMessage(packet.getAddress, msg)
55         }catch{
56           case e: Exception =>
57         e.printStackTrace()
58       throw new Exception
59         }
60       }
61     })
62     .withAttributes(
63     ActorOperationAttributes
64     .supervisionStrategy(Supervision.stoppingDecider)
65   )
66   }
67
68   val transmitSink = {
69     Flow[OppNetMessage]
70     .mapAsync(4)(msg => Future {
71       println("pacchetto send: " + msg)
72       val bytes = serialization.serialize(msg).get
73       new DatagramPacket(bytes, bytes.length, addr, port)
74     })
75     .to(Sink.foreach[DatagramPacket] { packet =>
76       try{
77         socket.send(packet)
78       }catch{
79         case e: Exception => e.printStackTrace()
80       }
81     })
82   }
83
84
85 }
86
87
88 object BroadcastPacketsTransceiver {
89
90   case class ReceivedMessage(
91     srcIpAddr: InetAddress,
92   message: OppNetMessage
93   )
94
95 }
```

Nelle Source e Sink definite in questa classe, la ricezione e l'invio dei dati dalla rete, avviene tramite un socket UDP, fornito dalle API Java. Il contenuto di un pacchetto UDP è rappresentato da un semplice array di byte, che dovrà quindi essere serializzato in una case class per essere utilizzato dal resto dello stream, e viceversa per la deserializzazione. Per effettuare questa operazione, ho scelto di utilizzare le funzionalità di serializzazione

integrate in Akka, le stesse utilizzate dalla funzionalità di remoting di Akka per comunicare con attori presenti su un computer remoto.

Akka permette di utilizzare varie librerie per la serializzazione di oggetti Java presenti in memoria nella JVM locale, ed ogni libreria di serializzazione utilizza in generale un formato di dati serializzati differente. È necessario quindi scegliere una libreria di serializzazione utilizzata in tutto il proprio sistema.

I due serializzatori più utilizzati in Akka sono Protobuf di Google, e la funzionalità di serializzazione integrata nelle librerie Java. Anche se Protobuf è solitamente preferita perchè è più veloce ed efficiente come dimensione dei messaggi serializzati, ho deciso per semplicità di utilizzare il serializzatore di Java, poichè è meno macchinoso da utilizzare durante lo sviluppo di un software prototipale.

Il serializzatore da utilizzare va specificato nel file di configurazione di Akka, che può essere posto all'interno di un progetto SBT in:

`resources/application.conf`

Queste sono le chiavi di configurazione da impostare per utilizzare la feature di serializzazione di Java:

```
# Entries for pluggable serializers and their bindings.
serializers {
  java = "akka.serialization.JavaSerializer"
  bytes = "akka.serialization.ByteArraySerializer"
}

# Class to Serializer binding.
# You only need to specify the name of an
# interface or abstract base class of the messages.
# In case of ambiguity it is using the most
# specific configured class, or giving a warning and
# choosing the \first" one.
#
# To disable one of the default serializers,
# assign its class to "none", like
# "java.io.Serializable" = none
serialization-bindings {
    "[B" = bytes
```

```
    "java.io.Serializable" = java
}
```

Il codice descritto finora, come è possibile notare dalle classi presenti negli `import`, è completamente indipendente da Android. Esso può quindi essere utilizzato in qualsiasi sistema che disponga di una piattaforma Java.

Per utilizzare il codice sopra descritto in una applicazione Android, occorre creare una Activity od un Service che internamente istanzi ed esegua un stream di Akka Stream che includa il codice in oggetto. Come primo prototipo di applicazione, ho scelto di creare una semplice Activity, chiamata MainActivity, che istanzia lo stream nel metodo di callback `onCreate`, invocato dal sistema operativo Android in seguito alla creazione dell'Activity. Per uniformità con il resto del progetto, ho scritto la MainActivity in linguaggio Scala, utilizzando la libreria Scaloid per disporre di un wrapper Scala idiomatico delle API Java di Android.

Questo è il codice in linguaggio Scala della MainActivity:

```
1  package it.unibo.akkaoppnet.android
2
3  import akka.actor.ActorSystem
4  import akka.stream.{Supervision, ActorFlowMaterializerSettings, OverflowStrategy,
       ActorFlowMaterializer}
5  import it.unibo.akkaoppnet.BroadcastPacketsTransceiver.ReceivedMessage
6  import it.unibo.akkaoppnet.models.Channel.{Normal, ChannelMsg}
7  import it.unibo.akkaoppnet.models.{Channel, Transmission}
8  import it.unibo.akkaoppnet.models.Transmission.Signal
9  import it.unibo.akkaoppnet.{OppNetMessage, TestMsg, BroadcastPacketsTransceiver}
10 import org.scaloid.common._
11 import akka.stream.scaladsl._
12 import scala.concurrent.duration._
13
14 class MainActivity extends SActivity {
15
16   lazy val logTextView = new STextView("Log:\n")
17
18   implicit lazy val system = ActorSystem("OppNetSystem")
19   implicit lazy val materializer = ActorFlowMaterializer(ActorFlowMaterializerSettings(
       system).withSupervisionStrategy(Supervision.getStoppingDecider))
20
21   lazy val transceiver = new BroadcastPacketsTransceiver("192.168.1.255", 10001)
22
23   var currNodeType: Channel.NodeType = Channel.Src
24
25
26
27   onCreate {
28
29     // Initialize the streams
30
31     val nodeTypeSource = Source.actorRef(1, OverflowStrategy.dropTail)
32
33     val channelGraph = FlowGraph.closed(nodeTypeSource) { implicit builder =>
34       nodeTypeSrc => {
35         import FlowGraph.Implicits._
36
37         val channel = builder.add( Channel(10, 3.seconds) )
38
39         val networkSource = builder.add(transceiver.receiveSource)
40         val networkSink = builder.add(transceiver.transmitSink)
41
```

```
42          val channelMsgFilter = builder.add( Flow[ReceivedMessage].collect{ case
        ReceivedMessage(a, m: ChannelMsg) => m } )
43          networkSource ~> channelMsgFilter ~> channel.networkRecv
44          channel.networkSend ~> networkSink
45
46          val channelLocationVisualizer = builder.add(
47              Sink.foreach[Channel.NodeChannelLocation] { location =>
48                println(location)
49                runOnUiThread(logTextView.append(location + "\n"))
50              }
51            )
52
53          nodeTypeSrc ~> channel.nodeType
54          channel.nodeChannelLocation ~> channelLocationVisualizer
55        }
56      }
57
58      val nodeTypeActor = channelGraph.run()
59
60      nodeTypeActor ! currNodeType // a first message is needed to make the Channel start
          working
61
62
63      // Initialize the Android UI
64
65      contentView = new SVerticalLayout {
66        SButton("Toggle node type", {
67          currNodeType = currNodeType match {
68            case Channel.Normal => Channel.Src
69            case Channel.Src => Channel.Dest
70            case Channel.Dest => Channel.Normal
71          }
72          nodeTypeActor ! currNodeType
73        })
74        this += logTextView.<<.fill.>>
75
76      }.padding(20 dip)
77
78    }
79
80  }
```

La MainActivity, come si evince dal codice, istanzia in onCreate uno stream formato dalle Source e Sink del componente BroadcastPacketTransceiver precedentemente descritto, e da un componente Channel che ha lo scopo di costruire un "canale", una tipologia di campo computazionale che verrà descritta nella prossima sezione.

La MainActivity include anche del codice che aggiorna l'interfaccia utente dell'Activity stessa al variare dello stato del componente Channel, in modo da permettere all'utente di monitorare il funzionamento del sistema.

3.3.3 Componenti Akka Stream per la costruzione di campi computazionali

Analizziamo ora come realizzare dei componenti software, basati sulle astrazioni fornite da Akka Stream, che eseguiti su un device in grado di effettuare comunicazioni oppurtunistiche con altri peer, riescano a costruire dei campi computazionali sul sistema composto dai device sui quali vengono eseguiti.

Questi componenti dovranno poter scambiare informazioni con i corrispondenti componenti eseguiti su altri device, quindi dovranno necessariamente accettare uno stream di messaggi provenienti dagli altri peer, e fornire in output uno stream di messaggi da inviare in broadcast agli altri peer direttamente raggiungibili. Inoltre possono per esempio fornire in output uno stream di aggiornamenti del loro stato interno, ed in input uno stream di impostazioni di configurazione del loro comportamento.

Gradiente

Consideriamo come esempio lo sviluppo di un componente per la costruzione di un semplice campo computazionale a "gradiente", che possa trasmettere informazioni "trasmesse" dalla sorgente più vicina. Per "gradiente", nell'ambito dei campi computazionali, si intende un campo che fa corrispondere ad ogni device un valore numerico che consiste nella distanza tra il device stesso ed un certo device sorgente fissato.

Ci poniamo ora l'obiettivo di costruire un campo simile a quello descritto che però faccia corrispondere ad ogni device anche un dato "trasmesso" dal device sorgente più vicino, oltre al valore di distanza dallo stesso device sorgente. Nel caso esistano più nodi sorgenti, il campo farà quindi corrispondere ogni device al dato trasmesso ed alla distanza dalla sorgente a distanza minima da esso. I dati provienienti dalle altre sorgenti verranno scartati.

Di seguito questo campo verrà chiamato "Transmission", sfruttando l'analogia della trasmissione di un segnale radio contenente informazioni, che si attenua in modo proporzionale alla distanza dalla sorgente.

Utilizzando le astrazioni fornite da Akka Stream, l'"interfaccia" di un componente che sia in grado di costruire il campo computazionale in oggetto, può essere definita in questo modo:

```
1 case class TrasmissionShape(
2     dataToTx: Inlet[Option[Data]],  // if None, this node doesn't transmit a signal
3     networkRecv: Inlet[TransmissionMsg],
4     receivedSignal: Outlet[Option[Signal]],   // the strongest signal currently
      received by this node
5     networkSend: Outlet[TransmissionMsg]
6   ) extends Shape
```

Lo stream in ingresso `networkRecv` consiste nei messaggi ricevuti da altri peer, mentre lo stream in uscita `networkSend` consiste nei messaggi che devono essere inviati in broadcast ai peer che il device corrente riesce a raggiungere.

Il tipo dei messaggi dei due stream appena descritti, è `TransmissionMsg`, definito in questo modo:

```
/**
   * These are all the messages exchanged over network by Transmission instances
   */
trait TransmissionMsg extends OppNetMessage
```

Il tipo TransmissionMsg rappresenta quindi i messaggi scambiati tra vari componenti Transmission, eseguiti su diversi device che compongono il sistema basato su comunicazioni opportunistiche. In una eventuale applicazione basata sul componente Transmission che stiamo sviluppando, occorrerà quindi connettere i suoi stream networkRecv e networkSend ad un altro componente in grado di scambiare i TransmissionMsg con altri peer del sistema.

Un componente adatto allo scopo può essere il BroadcastPacketsTransceiver definito nella sezione precedente della tesi, che può infatti serializzare i TransmissionMsg provenienti dallo stream networkSend ed inviarli sotto forma di pacchetti UDP in broadcast alla rete, e deserializzare i TransmissionMsg ricevuti da altri device della rete ed inviarli allo stream networkRecv.

Analizziamo ora il codice completo di una possibile implementazione del componente Transmission, basato su Akka Stream:

```
package it.unibo.akkaoppnet.models

import akka.stream._
import akka.stream.scaladsl._
import it.unibo.akkaoppnet.OppNetMessage

import scala.concurrent.duration._

import scala.collection._


/**
 * This model represent the transmission of a (radio?) signal from one or more transmitters, and the progressive weakening
 * of the signal with distance and interference with stronger ones.
 *
 * Every transmitting node periodically broadcasts a signal that contains some data.
 * Every non-transmitting node, receives the strongest signal from the neighbours, and rebroadcast a weaker copy of the original signal.
 */
object Transmission {


  def apply(listeningTimeInterval: FiniteDuration, debugName: String) : Graph[TrasmissionShape, Unit] = {

    def debug(msg: String) = println(debugName + ": " + msg)

    FlowGraph.partial() { implicit builder =>

      import FlowGraph.Implicits._

      val repeater = builder.add(
          Flow[Option[Data]]
            .map {
```

```
33              case None => None
34              case Some(data) => Some(Signal(0, data))
35            }
36            .expand(a => a)(a => (a, a)) // this repeatedly emits the last element received
37        )
38
39
40        val bestSignalFinder = builder.add(
41          Flow[TransmissionMsg]
42          // this groupes all the messages received in the listeningTimeInterval in a List. If none is received, it emits an empty List
43            .groupedWithin(1000, listeningTimeInterval)
44            .map(list => list
45              .foldLeft(None: Option[Signal])( (bestSignal, recvMsg) =>
46                recvMsg match {
47                  case recvSignal: Signal => bestSignal match {
48                    case None => Some(recvSignal)
49                    case Some(bestSign) =>
50                      if (bestSign.distanceToTransmitter <= recvSignal.distanceToTransmitter) {
51                        Some(bestSign)
52                      } else {
53                        Some(recvSignal)
54                      }
55                  }
56                  case _ => bestSignal
57                }
58              )
59            )
60        )
61
62        val zipper = builder.add(
63          ZipWith[Option[Signal], Option[Signal], Option[Signal]] { (received, myTx) =>
64            myTx match {
65              case None => received
66              case Some(s) => Some(s)
67            }
68          }
69          .withAttributes(OperationAttributes.inputBuffer(initial = 1, max = 1))
70        )
71
72        val finalDistanceBroadcaster = builder.add(Broadcast[Option[Signal]](2))
73
74        val toMsgToTransmit = builder.add(
75          Flow[Option[Signal]]
76            .collect{ case Some(signal) => signal }
77            .map(s => Signal(s.distanceToTransmitter + 1, s.data) )
78        )
79
80        repeater.outlet ~> zipper.in1
81        bestSignalFinder.outlet ~> zipper.in0
82        zipper.out ~> finalDistanceBroadcaster.in
83        finalDistanceBroadcaster.out(0) ~> toMsgToTransmit.inlet
84
85        val initialSrc = builder.add(Source(0.seconds, 1.seconds, DummySignal))
86        val merge = builder.add( Merge[TransmissionMsg](2) )
87        initialSrc.outlet ~> merge.in(0)
88        merge.out ~> bestSignalFinder.inlet
89
90        TrasmissionShape(
91          dataToTx = repeater.inlet,
92          networkRecv = merge.in(1),
93          receivedSignal = finalDistanceBroadcaster.out(1),
94          networkSend = toMsgToTransmit.outlet
95        )
96      }
97
98    }
99
100
101   /**
102    * These are all the messages exchanged over network by Transmission instances
103    */
104   trait TransmissionMsg extends OppNetMessage
105
```

```
106   /**
107    * This is a message that represent the signal broadcasted by a transmitter.
108    *
109    * @param distanceToTransmitter the distance (in hops) between the sending node and
         the nearest transmitter
110    */
111   case class Signal(distanceToTransmitter: Int, data: Data) extends TransmissionMsg
112
113   /**
114    * The data contained in a transmitted signal
115    */
116   trait Data extends OppNetMessage
117
118
119   // This is just an hack used internally
120   private case object DummySignal extends TransmissionMsg
121
122
123
124   case class TrasmissionShape(
125        // if None, this node doesn't transmit a signal
126        dataToTx: Inlet[Option[Data]],
127        networkRecv: Inlet[TransmissionMsg],
128      // the strongest signal currently received by this node
129        receivedSignal: Outlet[Option[Signal]],
130        networkSend: Outlet[TransmissionMsg]
131      ) extends Shape {
132
133     // Boilerplate code, nothing of interesting
134
135     override val inlets: immutable.Seq[Inlet[_]] =
136       dataToTx :: networkRecv :: Nil
137
138     override val outlets: immutable.Seq[Outlet[_]] =
139       receivedSignal :: networkSend :: Nil
140
141     override def deepCopy() = TrasmissionShape(
142       new Inlet[Option[Data]](dataToTx.toString),
143       new Inlet[TransmissionMsg](networkRecv.toString),
144       new Outlet[Option[Signal]](receivedSignal.toString),
145       new Outlet[TransmissionMsg](networkSend.toString)
146     )
147
148     override def copyFromPorts(
149          inlets: immutable.Seq[Inlet[_]],
150          outlets: immutable.Seq[Outlet[_]]) = {
151       assert(inlets.size == this.inlets.size)
152       assert(outlets.size == this.outlets.size)
153       TrasmissionShape(inlets(0).asInstanceOf[Inlet[Option[Data]]],
154         inlets(1).asInstanceOf[Inlet[TransmissionMsg]],
155         outlets(0).asInstanceOf[Outlet[Option[Signal]]],
156         outlets(1).asInstanceOf[Outlet[TransmissionMsg]]
157     )
158     }
159
160   }
161
162 }
```

Il metodo `apply` del object Transmission ritorna un FlowGraph che consiste sostanzialmente in una implementazione dell'"interfaccia" TransmissionShape precedentemente descritta. Questo FlowGraph internamente utilizza vari operatori di Akka Stream che elaborano i messaggi provenienti dagli stream in entrata, mantengono eventualmente uno stato interno, ed emettono dei messaggi destinati agli stream di uscita, senza produrre alcun side-effect.

Il valore del campo computazionale costruito da un sistema di device che

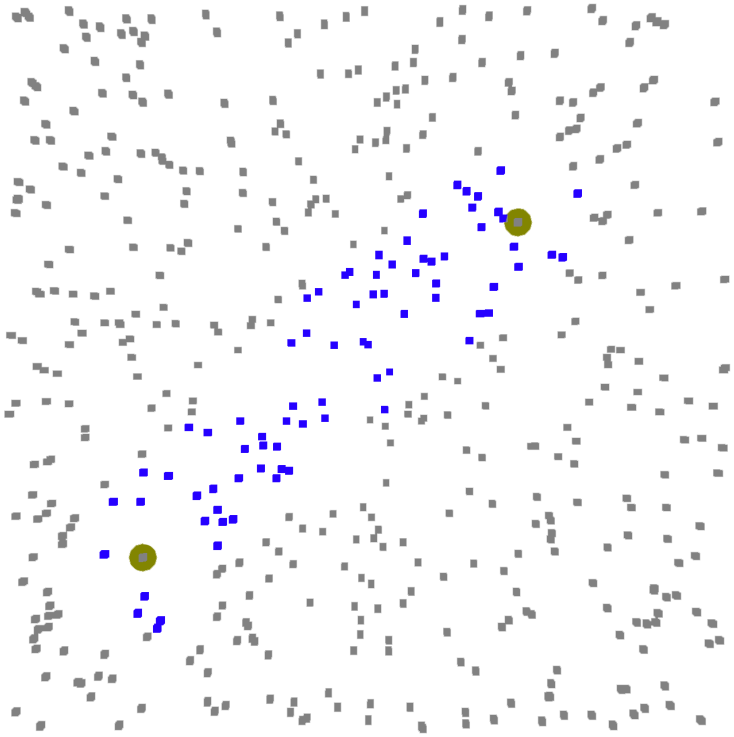
eseguono questo componente Transmission, può essere letto in ogni device ricevendo messaggi dallo stream in uscita receivedSignal. Il componente infatti emetterà in quello stream dei messaggi di tipo Signal in corrispondenza di ogni cambiamento del valore corrispondente al device corrente nel campo computazionale.

I messaggi Signal contengono la distanceToTransmitter che consiste nella lunghezza del percorso minimo di device attraverso i quali occorre effettuare comunicazioni opportunistiche per raggiungere il peer sorgente, misurata in numero di peer, ed un oggetto di tipo Data, che rappresenta l'informazione trasmessa dalla sorgente più vicina.

Canale

Un'altra struttura interessante basata su campi computazionali, consiste nel "canale" formato dai device vicini al percorso minimo tra un device sorgente ed un device destinazione. Un campo computazionale che formi un "canale" dovrà quindi far corrispondere ad ogni device del sistema un valore per esempio booleano, che indica se il device in oggetto è all'interno o all'esterno del canale.

In questa immagine è visibile una rappresentazione di un sistema simulato con Proto, dove ogni device corrisponde ad un punto, ed il colore del punto indica il valore del campo computazionale per quel device:



I device di colore blu sono quelli all'interno del canale, mentre quelli di colore grigio sono fuori dal canale. I due device colorati di giallo, sono uno il device sorgente ed uno il device destinazione del canale. Come è possibile notare dalla figura, i device che fanno parte del canale sono quelli la cui distanza dalla linea del percorso minimo tra sorgente e destinazione è minore di una certa soglia.

Un campo computazionale che rappresenta un canale, può essere costruito a partire da dei "gradienti", come dalla tipologia di campo computazionale vista nella sezione precedente della tesi, di cui è stato anche fornito un software in grado di costruirlo sulla base di comunicazioni opportunistiche tra device mobili. Ipotizzando che il device sorgente generi un gradiente formato dal campo Transmission visto nella sezione precedente, esso verrà "ricevuto" dal device destinazione, il quale a sua volta genererà un altro campo Transmission, il quale "trasmette" però un valore numerico della distanza tra il device sorgente ed il device destinazione, ottenuta dal campo emesso dal device sorgente. Per ogni device, utilizzando le informazioni

fornite dai due campi computazionali, è possibile calcolare se esso si trova dentro o fuori dal canale.

Analizziamo ora una possibile implementazione di un componente software, basato su Akka Stream, che permetta la costruzione di un campo computazionale di una struttura a canale in un sistema di device mobile utilizzando comunicazioni opportunistiche.

Il componente esposto di seguito è chiamato Channel, ed utilizza internamente due istanze dei componenti Transmission visti in precedenza:

```
1  package it.unibo.akkaoppnet.models
2
3  import akka.stream.FanOutShape.{Name, Init}
4  import akka.stream._
5  import akka.stream.scaladsl._
6  import it.unibo.akkaoppnet.OppNetMessage
7  import it.unibo.akkaoppnet.models.Transmission.{TransmissionMsg, Signal}
8  import scala.collection.immutable
9  import scala.concurrent.duration._
10
11
12 /**
13  * This model represents a set of channels going from a set of sources nodes to a set
       of destination nodes.
14  * For each source, there must be a channel going to each destination.
15  */
16 object Channel {
17
18   def apply(width: Int, listeningTimeInterval: FiniteDuration) : Graph[ChannelShape,
       Unit] = {
19
20     FlowGraph.partial() { implicit builder =>
21       import FlowGraph.Implicits._
22
23       val srcTx = builder.add(Transmission(listeningTimeInterval, debugName = "srcTx"))
24       val destTx = builder.add(Transmission(listeningTimeInterval, debugName = "destTx"
       ))
25
26
27       // Send the data to transmitters
28       val nodeTypeBroadcaster = builder.add( Broadcast[NodeType](2) )
29       val srcTxDataSetter = builder.add( Flow[NodeType].map{
30           case Src => Some(DummyData)
31           case _ => None
32         }
33       )
34       val destTxDataSetter = builder.add( Flow[Option[Int]].map {
35           case None => None
36           case Some(dist) => Some(SrcDestDistanceData(dist))
37         }
38       )
39       nodeTypeBroadcaster ~> srcTxDataSetter ~> srcTx.dataToTx
40       destTxDataSetter ~> destTx.dataToTx
41
42
43       val srcReceivedSignalBroadcaster = builder.add( Broadcast[Option[Signal]](2) )
44       srcTx.receivedSignal ~> srcReceivedSignalBroadcaster
45
46       val nodeTypeRepeater = builder.add( Flow[NodeType].expand(a => a)(a => (a, a)) )
47       val srcDestDistanceCalculator = builder.add( ZipWith[NodeType, Option[Signal],
       Option[Int]] {
48           (nodeType, srcReceivedSignal) => nodeType match {
49             case Dest => srcReceivedSignal match {
50               case None => None
51               case Some(Signal(distToTx, data)) => Some(distToTx)
52             }
53             case _ => None
54           }
55         }
56         .withAttributes(OperationAttributes.inputBuffer(initial = 1, max = 1))
```

```
57        )
58
59        nodeTypeBroadcaster ~> nodeTypeRepeater ~> srcDestDistanceCalculator.in0
60        srcReceivedSignalBroadcaster ~> srcDestDistanceCalculator.in1
61        srcDestDistanceCalculator.out ~> destTxDataSetter
62
63
64        //// Network communication
65
66        // Handle messages received from the network
67        val networkRecvBroadcaster = builder.add( Broadcast[ChannelMsg](2) )
68        val srcTxRecvFilter = builder.add( Flow[ChannelMsg].collect { case SrcTxMsg(msg)
      => msg } )
69        val destTxRecvFilter = builder.add( Flow[ChannelMsg].collect { case DestTxMsg(msg
      ) => msg } )
70        networkRecvBroadcaster ~> srcTxRecvFilter ~> srcTx.networkRecv
71        networkRecvBroadcaster ~> destTxRecvFilter ~> destTx.networkRecv
72
73        // Send messages to the network
74        val networkSendMerger = builder.add( Merge[ChannelMsg](2) )
75        val srcTxSendFilter = builder.add( Flow[TransmissionMsg].map( m => SrcTxMsg(m) )
      )
76        val destTxSendFilter = builder.add( Flow[TransmissionMsg].map( m => DestTxMsg(m)
      ) )
77        srcTx.networkSend ~> srcTxSendFilter ~> networkSendMerger.in(0)
78        destTx.networkSend ~> destTxSendFilter ~> networkSendMerger.in(1)
79
80
81        // This calculates if I'm inside the channel or not
82        val channelLocationCalculator = builder.add( ZipWith[Option[Signal],Option[Signal
      ],NodeChannelLocation]{
83          (srcTxSignal, destTxSignal) =>
84            println(srcTxSignal, destTxSignal)
85            if(srcTxSignal.isEmpty || destTxSignal.isEmpty) Outside else {
86              destTxSignal.get.data match {
87                case SrcDestDistanceData(srcDestDistance) =>
88                  val insideChannel = (srcTxSignal.get.distanceToTransmitter +
      destTxSignal.get.distanceToTransmitter) <= (srcDestDistance + width)
89                  if (insideChannel) Inside else Outside
90              }
91            }
92        })
93        srcReceivedSignalBroadcaster ~> channelLocationCalculator.in0
94        destTx.receivedSignal ~> channelLocationCalculator.in1
95
96        ChannelShape(
97          nodeType = nodeTypeBroadcaster.in,
98          networkRecv = networkRecvBroadcaster.in,
99          nodeChannelLocation = channelLocationCalculator.out,
100         networkSend = networkSendMerger.out
101       )
102     }
103
104   }
105
106
107   // Messages exchanged over the network
108   trait ChannelMsg extends OppNetMessage
109   case class SrcTxMsg(transmissionMsg: TransmissionMsg) extends ChannelMsg
110   case class DestTxMsg(gradientMsg: TransmissionMsg) extends ChannelMsg
111
112   // These messages set the status of this node to a source or destination of the
        channel, or nothing of the two.
113   trait NodeType
114   case object Src extends NodeType
115   case object Dest extends NodeType
116   case object Normal extends NodeType
117
118   // The messages are given in output to tell if this node is currently inside or
        outside the channel.
119   trait NodeChannelLocation
120   case object Inside extends NodeChannelLocation
121   case object Outside extends NodeChannelLocation
122
123   // Data to be transmitted by the Transmission models
124   case class SrcDestDistanceData(srcDestDistance: Int) extends Transmission.Data
```

```
125   case object DummyData extends Transmission.Data
126
127
128
129   case class ChannelShape(
130         nodeType: Inlet[NodeType],
131         networkRecv: Inlet[ChannelMsg],
132         nodeChannelLocation: Outlet[NodeChannelLocation],
133         networkSend: Outlet[ChannelMsg]
134       ) extends Shape {
135
136     // Boilerplate code, nothing of interesting
137
138     override val inlets: immutable.Seq[Inlet[_]] =
139       nodeType :: networkRecv :: Nil
140
141     override val outlets: immutable.Seq[Outlet[_]] =
142       nodeChannelLocation :: networkSend :: Nil
143
144     override def deepCopy() = ChannelShape(
145       new Inlet[NodeType](nodeType.toString),
146       new Inlet[ChannelMsg](networkRecv.toString),
147       new Outlet[NodeChannelLocation](nodeChannelLocation.toString),
148       new Outlet[ChannelMsg](networkSend.toString)
149     )
150
151     override def copyFromPorts(
152                                  inlets: immutable.Seq[Inlet[_]],
153                                  outlets: immutable.Seq[Outlet[_]]) = {
154       assert(inlets.size == this.inlets.size)
155       assert(outlets.size == this.outlets.size)
156       ChannelShape(
157         inlets(0).asInstanceOf[Inlet[NodeType]],
158         inlets(1).asInstanceOf[Inlet[ChannelMsg]],
159         outlets(0).asInstanceOf[Outlet[NodeChannelLocation]],
160         outlets(1).asInstanceOf[Outlet[ChannelMsg]])
161     }
162
163   }
164
165 }
```

Capitolo 4

Conclusioni

In questa tesi è stato analizzato il problema dello sviluppo di sistemi distribuiti basati su comunicazioni opportunistiche wireless tra dispositivi mobile, studiando varie tecnologie utili allo scopo, ed effettuando alcune sperimentazioni su di esse.

In particolare è stata analizzata approfonditamente la tecnologia di comunicazione wireless delle reti Wifi in modalità ad hoc, sulla quale sono state effettuate simulazioni di ipotetici sistemi di larga scala.

Dai risultati delle simulazioni, la tecnologia delle reti Wifi ad hoc risulta essere particolarmente adatta per gestire comunicazioni opportunistiche wireless in sistemi di larga scala formati da device mobile.

Inoltre sono state sviluppate delle implementazioni di componenti software in grado di costruire campi computazionali tramite comunicazioni opportunistiche tra dispositivi mobile.

Tramite il software sviluppato, è possibile creare applicazioni per device Android che basano il loro funzionamento su campi computazionali costruiti tramite reti Wifi ad hoc tra i vari dispositivi che compongono il sistema.

Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Mirko Viroli per avermi seguito nella stesura di questa tesi e le per le numerose ore che mi ha dedicato. Ringrazio poi la mia famiglia e gli amici, che mi sono stati vicini durante questi anni di studio.

Bibliografia

[1] Sito ufficiale di CSR Mesh.
http://www.csr.com/products/csrmesh-development-kit

[2] Sito ufficiale Thread Group.
http://threadgroup.org/Home.aspx

[3] Information about cell-id splitting, stuck beacons, and failed IBSS merges, VillageTelco.
http://wiki.villagetelco.org/Information_about_cell-id_splitting,_stuck_beacons,_and_failed_IBSS_merges!

[4] Sito ufficiale di ns-3.
https://www.nsnam.org/

[5] Propagation Models, sito ufficiale di ns-3.
https://www.nsnam.org/docs/models/html/propagation.html

[6] Wifi model documentation, sito ufficiale di ns-3.
https://www.nsnam.org/docs/models/html/wifi-design.html#overview-of-the-model

[7] IEEE 802.11-2009 Data Rates, Wikipedia.
https://en.wikipedia.org/wiki/IEEE_802.11n-2009#Data_rates

[8] IEEE 802.11-2009 Frame Aggregation, Wikipedia.
https://en.wikipedia.org/wiki/IEEE_802.11n-2009#Frame_aggregation

[9] MultipeerConnectivity Framework documentation, Apple.
https://developer.apple.com/library/ios/documentation/MultipeerConnectivity/Reference/MultipeerConnectivityFramework/

[10] Sito ufficiale Proto.
http://proto.bbn.com/

[11] Mamei, M., Zambonelli, F.
Programming pervasive and mobile computing applications: The tota approach. ACM Trans. on Software Engineering Methodologies 18(4), 1–56 (2009).

[12] Beal, J., Bachrach, J., Vickery, D., Tobenkin, M.
Fast self-healing gradients. In: Proceedings of ACM SAC 2008, pp. 1969–1975. ACM (2008).

[13] Viroli, M., Casadei, M., Montagna, S., Zambonelli, F.
Spatial coordination of pervasive services through chemical-inspired tuple spaces. ACM Transactions on Autonomous and Adaptive Systems 14, 14:1–14:24 (2011)

[14] Applicazione Android SuperSU, Chainfire.
https://play.google.com/store/apps/details?id=eu.chainfire.supersu

[15] Applicazione Android ConnectBot.
`https://play.google.com/store/apps/details?id=org.connectbot`

[16] Sito ufficiale Serval Project.
`http://www.servalproject.org/`

[17] Applicazione Android Batphone, Serval Project.
`https://github.com/servalproject/batphone/tree/development/jni`

[18] Nexus 5, WikiDevi.
https://wikidevi.com/wiki/Google_Nexus_5

[19] android-sdk-plugin GitHub repository.
`https://github.com/pfn/android-sdk-plugin`