POLITECNICO DI MILANO
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

SVILUPPO DI UN MIDDLEWARE PER GIOCHI MULTIPIATTAFORMA A PIÙ GIOCATORI SU SUPPORTI MOBILI

Relatore: Prof. Pier Luca Lanzi
Correlatori: Ing. Gaetano Peligra

Tesi di Laurea di:
Mauro Ronchi, matricola 749802

Anno Accademico 2011-2012

A chiunque abbia il fegato di leggerla...

Indice

Sommario

Questa tesi si inquadra nell'ambito dei videogiochi su dispositivi mobili, un settore in grande crescita negli ultimi anni, soprattutto per l'aumento dei videogiocatori dovuto alla rapida diffusione e facilità di utilizzo degli smartphone.

Lo scopo di questa tesi è il gioco multi piattaforma a più giocatori su piattaforme mobili, un settore in cui sono state individuate diverse possibilità di business.

Per gioco multigiocatore si intende un videogioco in cui due o più giocatori umani giocano alla stessa partita, sullo stesso dispositivo oppure su dispositivi diversi che comunicano tramite Internet o rete locale. Questo si contrappone al gioco a giocatore singolo, in cui il giocatore gioca senza interazione con altri giocatori umani.

Ringraziamenti

Ringrazio innanzitutto il prof. Lanzi che mi ha dato l'occasione di iniziare a sviluppare videogiochi, quindi i miei colleghi e compagni di progetto per l'esperienza maturata nel lavorare insieme.

Capitolo 1

Introduzione

Questa tesi si inquadra nell'ambito dei videogiochi, in particolare discutiamo di videogiochi su dispositivi mobili quali smartphone o altri terminali portatili. Tale settore è stato protagonista di una grande crescita negli ultimi anni, a causa soprattutto della rapida diffusione e facilità di utilizzo degli smartphone, che ha permesso a un numero sempre maggiore di persone di avvicinarsi al mondo dei videogiochi.

Lo scopo di questa tesi è il gioco multi piattaforma a più giocatori su piattaforme mobili, un settore in cui sono state individuate diverse possibilità di business, in particolare su dispositivi Nokia Symbian, per i quali non è disponibile nessun gioco multigiocatore, ad eccezione di applicazioni web visualizzate sul browser la cui logica applicativa viene eseguita su server remoti.

Per gioco multigiocatore si intende un videogioco in cui due o più giocatori umani giocano alla stessa partita, sullo stesso dispositivo oppure su dispositivi diversi che comunicano tramite Internet o rete locale.

Questo si contrappone al gioco a giocatore singolo, in cui il giocatore gioca senza interazione con altri giocatori umani, per esempio risolvendo enigni o giocando contro avversari controllati dal gioco stesso.

In questa tesi discutiamo di giochi multigiocatore in cui i giocatori sono connessi tramite Internet, ciascuno con il proprio dispositivo.

Esistono diverse tipologie di giochi multigiocatore su Internet, a seconda delle modalità di interazione tra i giocatori:

- Giochi multigiocatore in tempo reale, in cui i giocatori si connettono tra loro e ogni azione di gioco viene inviata istantaneamente agli altri. Ogni giocatore può compiere azioni di gioco in qualsiasi momento della partita.

- Giochi multigiocatore a turni, in cui solo un giocatore alla volta può compiere azioni di gioco. Il gioco stesso decide quale giocatore deve giocare in ogni momento della partita.

- Giochi social, in cui i giocatori giocano sostanzialmente da soli, ma con la possibilità di inviare notifiche o richieste ad altri giocatori tramite una piattaforma di social network di riferimento (ad esempio Facebook)

- MMO, ovvero Massive Multiplayer Online Games, in cui tutti i giocatori sono connessi ad un server che gestisce un unico mondo di gioco persistente, in cui i giocatori possono entrare e uscire in qualsiasi momento. Il server può ospitare centinaia o migliaia di giocatori connessi contemporaneamente.

Nei primi due casi la logica del gioco viene eseguita sui client, in maniera distribuita oppure, più frequentemente, eleggendo un host (che può coincidere con il terminale di uno dei giocatori oppure essere eseguito su una macchina ad hoc) ed affidandogli il compito di gestire la logica per la partita corrente. Negli ultimi due casi invece tutta la logica di gioco viene eseguita sul server, e i client si limitano ad inviare gli input dei giocatori e visualizzare i risultati.

In questa tesi ci concentriamo sui giochi multigiocatore in tempo reale e a turni. Abbiamo sviluppato un middleware in grado di connettere un certo numero di giocatori e gestire lo scambio diretto dei dati di gioco, utilizzando un server centrale solo per la ricerca di giocatori all'inizio della partita.

Lo scopo di questa tesi è quindi quello di facilitare l'implementazione di giochi multipiattaforma a più giocatori a turni o in tempo reale.

La tesi è strutturata nel modo seguente.

Nel capitolo due mostriamo lo stato dell'arte dei middleware in ambito mobile, PC e console.

Nel capitolo tre illustriamo gli obiettivi della tesi e il nostro approccio alla loro realizzazione.

Nel capitolo quattro descriviamo le caratteristiche dell'ambiente di sviluppo e delle librerie utilizzate.

Nel capitolo cinque poniamo l'accento sul conseguimento dell'obbiettivo dal punto di vista pratico e quindi l'implementazione del middleware.

Nel capitolo sei illustriamo l'applicazione sviluppata per dimostrare il corretto funzionamento del middleware.

Nel capitolo sette effettuiamo alcune valutazioni sul lavoro svolto e alcuni sviluppi futuri.

Capitolo 2

Stato dell'arte dei middleware per giochi

In questo capitolo descriviamo diverse soluzioni per il supporto delle funzionalità multigiocatore o di social gaming, disponibili per varie piattaforme hardware e software tra cui computer (Windows, Linux e Mac), console (PS3, XBox 360 e Wii) e terminali mobili (iPhone, Android, Windows Phone e Symbian), con particolare attenzione alle soluzioni che supportano piattaforme mobili.

Analizziamo in particolare la capacità di consentire connessioni peer to peer tra giocatori in modo da evitare eccessivo carico su server.

2.1 Game Center di Apple

Il *Game Center[1]* è la piattaforma sviluppata da Apple per il supporto al multiplayer su dispositivi mobili con sistema operativo iOS. Permette partite a più giocatori tra amici invitati dall'utente oppure partite tra giocatori scelti a caso dal sistema. Fornisce inoltre supporto a funzionalità come classifiche e achievement.

Il sistema richiede la registrazione del proprio account. Una volta iscritti è possibile giocare da soli o con altre persone. E' possibile consultare il profilo dei propri amici se sono iscritti al *Game Center* per vedere a quali giochi hanno giocato di recente. Per confrontare i risultati con quelli dei nostri amici possiamo consultare in ogni momento la classifica generale dove sono presenti i punteggi dei giocatori sia nei singoli giochi sia all'interno della community di Game Center.

Il *Game Center* ha il vantaggio di non richiedere lo sviluppo di un server dedicato, tuttavia può essere utilizzato solo su dispositivi iOS.

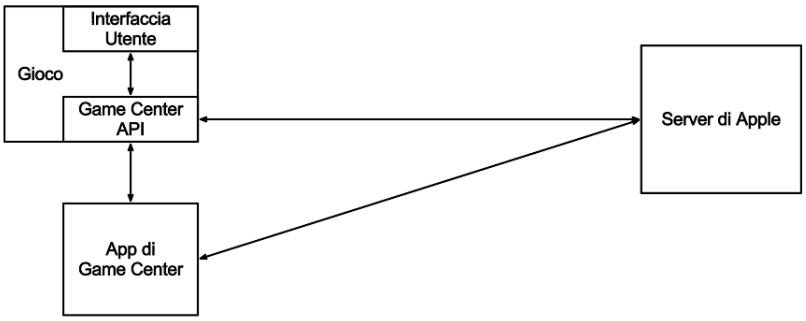

Figura 2.1: Architettura del *Game Center* di Apple

2.1.1 Architettura

Il *Game Center* (Figura 2.1) comprende un server centralizzato gestito da Apple, un'app installata sui dispositivi che gestisce il profilo dell'utente e una libreria da includere nei giochi. Per utilizzare il *Game Center* in un gioco è necessario registrare l'applicazione al server di *Game Center* utilizzando il *Bundle Identifier*, ovvero una stringa che identifica in modo univoco un'applicazione e che deve essere definita per tutte le app iOS.

Il supporto al *Game Center* in un gioco può essere obbligatorio oppure opzionale. Nel primo caso il gioco può essere installato solo sui dispositivi che supportano il *Game Center* (iOS versione 4.1 o superiore).

Per poter utilizzare le funzionalità del *Game Center* in un gioco, è necessario che il giocatore sia autenticato al servizio. La procedura di autenticazione funziona nel seguente modo: quando il gioco lo richiede (solitamente subito dopo l'avvio, ma può farlo in qualsiasi momento) *Game Center* controlla se il giocatore è già autenticato sul dispositivo, se non lo è presenta al giocatore una schermata in cui può autenticarsi o registrare un nuovo account. Se l'autenticazione fallisce, non è possibile utilizzare le funzionalità del *Game Center*.

Dopo che il giocatore si è autenticato è possibile visualizzare il proprio profilo, classifiche e achievement sul *Game Center*, giocare una partita multigiocatore, inviare al *Game Center* i punteggi ottenuti nel gioco o completare degli *achievement*.

Il profilo è un insieme di schermate visualizzabili all'interno dei giochi e nell'app del *Game Center*, contenente informazioni relative all'utente, quali il suo nome, la lista dei suoi amici e dei giochi che ha installato e facoltativamente un avatar e una breve descrizione inseriti dal giocatore stesso. Per

ogni gioco, sono presenti le classifiche e gli *achievement* relativi ad esso.

Per giocare una partita multigiocatore, il gioco effettua una richiesta al *Game Center* fornendo diversi parametri. Il *Game Center* visualizza una schermata in cui il giocatore può invitare i propri amici alla partita oppure lasciare che gli altri giocatori siano scelti a caso dal *Game Center* tra quelli che hanno richiesto partite compatibili. Una volta trovati abbastanza giocatori, il *Game Center* restituisce il controllo al gioco, che a questo punto può iniziare la partita.

Il *Game Center* fornisce il supporto a classifiche basate su criteri definiti dallo sviluppatore di ogni gioco in base alle sue esigenze. Lo sviluppatore può creare sul server di *Game Center* un numero arbitrario di classifiche, ciascuna delle quali può essere personalizzata con certi parametri ed è identificata da una categoria. All'interno del gioco è possibile inviare al *Game Center* i punteggi ottenuti, specificando la categoria di appartenenza, che verranno utilizzati per costruire le classifiche. Tali classifiche sono visualizzabili all'interno del gioco oppure nell'app di *Game Center*.

Gli *achievement* sono degli obiettivi da raggiungere all'interno di un gioco, memorizzati sul server del *Game Center* e collegati all'utente. Essi vengono definiti dallo sviluppatore allo stesso modo delle classifiche, e all'interno del gioco è possibile inviare al *Game Center* la percentuale di completamento di ciascuno di essi. Un giocatore ottiene un certo numero di punti al completamento di un *achievement*, su cui viene calcolata una classifica generale dei suoi amici.

2.2 GREE Platform

GREE Platform[2] è il nuovo nome di *OpenFeint*, la prima piattaforma che permette agli sviluppatori di aggiungere caratteristiche di social gaming ai propri giochi. Fornisce tutte le funzionalità del *Game Center* di Apple tranne il *matchmaking*. L'utilizzo di *GREE Platform* richiede l'autenticazione del giocatore.

GREE Platform è disponibile nativamente per piattaforme iOS e Android e tramite HTTP su qualunque piattaforma, tuttavia non fornisce funzionalità di comunicazione diretta tra dispositivi, necessarie ai giochi multigiocatore.

2.2.1 Architettura

Il funzionamento della piattaforma si basa su un server centralizzato che gestisce tutti gli utenti, i giochi, le classifiche gli *achievements* presenti. *GREE Platform* mette a disposizione due diverse modalità per utilizzare le sue funzionalità in un gioco: includendo le librerie native iOS o Android,

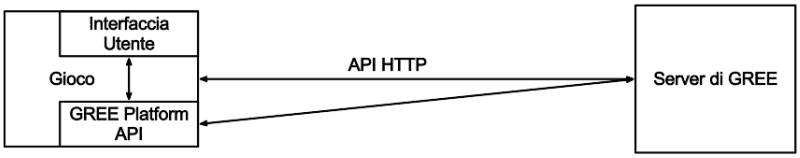

Figura 2.2: Architettura di GREE Platform

oppure utilizzando direttamente le API di rete basate su HTTP. In entrambi i casi è necessario registrare il gioco sul sito di GREE Platform, per ottenere un identificativo da utilizzare per l'utilizzo del servizio.

Prima di poter utilizzare le funzionalità di GREE Platform in un gioco, il giocatore deve autenticarsi al servizio. Per far ciò, il gioco chiede a *GREE Platform* di mostrare all'utente una schermata contenente i campi per l'autenticazione, e al termine della procedura riceve il risultato.

Dopo essersi autenticato, il giocatore può visualizzare la propria bacheca contenente classifiche, *achievements*, messaggi e richieste dei propri amici. Altre funzionalità disponibili includono la condivisione dei risultati (per esempio al termine di una partita) e la gestione di moneta virtuale acquistabile con denaro reale e spendibile all'interno dei giochi per ottenere oggetti o funzionalità *premium*.

GREE Platform supporta l'uso di *Push Notifications* tramite C2DM[1] su Android o APNS[2] su iOS per ricevere notifiche relative ai giochi anche quando essi non sono in esecuzione.

2.3 GameSpy SDK

GameSpy SDK[3] è una piattaforma che permette agli sviluppatori di aggiungere funzionalità multiplayer e di social gaming ai propri giochi. Dispone di API scritte in C e plugin per Unity3D e Unreal Engine. Tra le funzionalità più interessanti sono presenti il matchmaking e il Nat Punchthrough.

[1] Android Cloud to Device Messaging, servizio di push notification su Android `http://en.wikipedia.org/wiki/Android_Cloud_to_Device_Messaging_Service`

[2] Apple Push Notification Service, servizio di Push Notification su iOS `http://en.wikipedia.org/wiki/Apple_Push_Notification_Service`

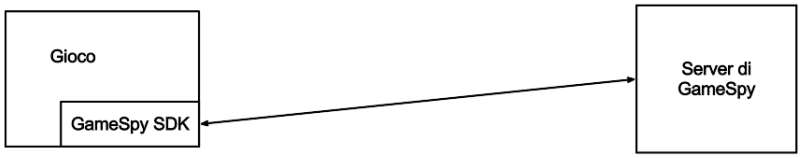

Figura 2.3: Architettura di GameSpy SDK

Le piattaforme ufficialmente supportate sono: PC (Windows, Mac OS X), Console (PS3, Wii, DS, PS Vita) e iPhone.

GameSpy SDK è una piattaforma flessibile ed efficiente adatta a grossi progetti, tuttavia richiede molto lavoro riuscire ad utilizzarla, rendendola non adatta a progetti più piccoli. I servizi di *GameSpy* sono a pagamento.

2.3.1 Architettura

Le componenti principali di *GameSpy SDK* sono il master server (fornito da *GameSpy*) e una libreria scritta in C. Per poter utilizzare il framework bisogna innanzitutto registrare il proprio gioco sul server di *GameSpy* e ottenere un *Game ID* e delle chiavi segrete per consentire la comunicazione sicura client-server.

Un gioco che utilizza *GameSpy SDK* deve per prima cosa inizializzare la piattaforma eseguendo un controllo di disponibilità del backend. Se ha successo, è possibile autenticare il giocatore utilizzando le credenziali da lui fornite. A seconda del tipo di autenticazione effettuato è possibile accedere a determinate funzionalità.

GameSpy SDK fornisce tutte le funzionalità del *Game Center* di Apple, più il *Nat Punchthrough* e servizi *cloud*.

Il *Nat Punchthrough* permette a dispositivi connessi a Internet tramite dietro firewall o NAT (Network Address Translation)[4] che bloccano le connessioni in entrata, di aggirare tale blocco e poter agire da server. La sua efficacia dipende dai router connessi.

GameSpy SDK permette inoltre ai giocatori di inviare ai propri server dati arbitrari e condividerli con altri giocatori.

2.4 Master Server di Unity

Unity[7] è un motore grafico 3d per creare giochi, che dispone di un editor avanzato per la creazione delle scene di gioco e supporta i linguaggi Java-

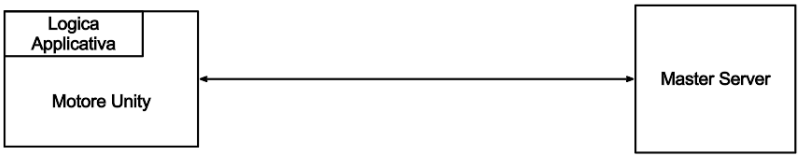

Figura 2.4: Architettura del Master Server di Unity

script, C# e Boo per l'implementazione della logica applicativa. Fornisce inoltre numerosi tool per integrare funzionalità avanzate.

Il *Master Server* di Unity è un componente che permette l'utilizzo di funzionalità di matchmaking e *Nat Punchthrough* per poter giocare partite multigiocatore. Il suo utilizzo è immediato, tuttavia richiede l'utilizzo di un server dedicato per poter funzionare.

2.4.1 Architettura

Il *Master Server* di Unity è composto da un componente client integrato nel motore Unity, e da un programma server da installare su un server con IP pubblico e noto, a cui le istanze del gioco faranno riferimento per trovare e connettersi ad altre istanze dello stesso gioco e quindi per poter giocare partite a più giocatori. Si occupa inoltre della gestione delle connessioni e di tutte le operazioni che sono necessarie durante l'inizializzazione della connessione di rete, come la gestione dei firewall e il *NAT Punchthrough*.

Per iniziare una partita a più giocatori, le modalità sono diverse a seconda chesi voglia fare il server di gioco o il semplice client. Il server di gioco si registra al *Master Server* utilizzando una stringa predefinita, quindi aspetta le connessioni da parte dei client. I client interrogano il *Master Server* per una lista di server di gioco, quindi ne scelgono uno a cui connettersi.

Se un client non riesce a connettersi direttamente al server, è possibile effettuare il *Nat Punchthrough*. La procedura è eseguita automaticamente quando serve, se abilitata.

Il master server si occupa del *Nat Punchthrough* come impostazione predefinita, ma ciò non è necessario. Le operazioni di *Nat Punchthrough* sono effettuate da un processo separato chiamato *Facilitator*. Se due client sono connessi allo stesso *Facilitator*, allora sono potenzialmente in grado di connettersi fintanto che conoscono i rispettivi indirizzi IP e porte esterni. Il master server viene utilizzato per fornire tali indirizzi, che altrimenti sarebbero difficili da determinare.

Capitolo 3

Un nuovo middleware multi piattaforma per giochi

L'obiettivo del nostro lavoro è quello di sviluppare un middleware in grado di fornire il supporto ai giochi multigiocatore su dispositivi di fascia bassa, ad esempio quelli forniti di sistema operativo Symbian.

Tra tutte le piattaforme esaminate nel capitolo 2, la maggior parte si basano su server proprietari, sono disponibili per una singola piattaforma hardware e software, oppure richiedono costi di licenza eccessivi.

Per questo motivo, abbiamo scelto di implementare una nostra piattaforma basandoci su librerie il più possibili standard, in modo da poterla sviluppare considerando quante più piattaforme possibili, nello specifico Symbian, iOS e Android. L'obiettivo finale è quello di permettere la comunicazione tra tutti i dispositivi.

Facendo ciò, abbiamo ottenuto una piattaforma flessibile e facilmente estendibile, in quanto abbiamo il controllo completo sull'architettura del sistema, sia lato client che lato server.

La libreria sviluppata fornisce funzionalità multiplayer, occupandosi della risoluzione di problematiche relative alla gestione degli utenti, connessione tra i giocatori e gestione delle partite e fornendo un'interfaccia semplice e progettata su misura per le nostre esigenze.

Il linguaggio scelto per l'implementazione è il C++, in quanto permette di sviluppare applicazioni per tutte le piattaforme menzionate e fornisce un controllo di basso livello sulle risorse utilizzate, caratteristica ideale per lo sviluppo su dispositivi mobili con risorse limitate.

La libreria è stata sviluppata utilizzando *RakNet[8]* per la gestione della rete. *RakNet* è un framework che permette connessioni rapide e affidabili tra due o più sistemi remoti, e presenta un'interfaccia semplificata per l'invio e la ricezione di messaggi. Inoltre sono disponibili diversi plugin che si occupano di funzionalità più avanzate, tra cui *Nat Punchthrough*, la gestione di un database centralizzato per la gestione degli utenti, un sistema di stanze per

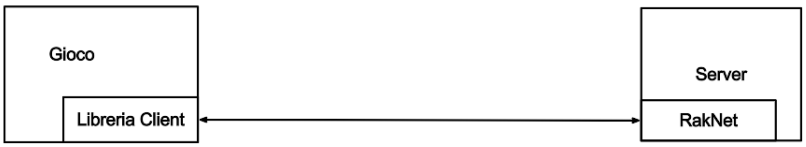

Figura 3.1: Architettura del nostro middleware

la gestione delle partite e un proxy in grado di inoltrare la comunicazione tra client attraverso un server.

3.1 Architettura

L'architettura del sistema è suddivisa tra client e master server. Il master server si occupa della gestione del database e delle stanze, nonché del supporto alla connessione tra gli utenti tramite *Nat Punchthrough* o proxy. Il client si occupa della connessione al master server, della scelta dell'host per ogni partita e delle operazioni di *Nat Punchthrough*.

È stato inoltre implementato un sistema di classifiche potente e flessibile, in cui è possibile memorizzare punteggi o statistiche arbitrarie per ogni utente e calcolare un numero arbitrario di classifiche basate su una qualsiasi combinazione di essi, come ad esempio la percentuale di vittorie ottenute.

Capitolo 4

Ambiente di sviluppo

In questo capitolo illustriamo le caratteristiche dell'ambiente utilizzato per lo sviluppo del nostro middleware multigiocatore e delle applicazioni che la utilizzano. La piattaforma di riferimento lato client è Symbian, tuttavia abbiamo scritto la libreria in C++ standard in modo da essere compatibile con qualunque piattaforma.

4.1 RakNet

La gestione del protocollo di rete è affidata alla libreria *RakNet*[8, 9]. *RakNet* fornisce un protocollo basato su *UDP* a cui aggiunge diverse funzionalità, come l'ordinamento dei pacchetti e la ritrasmissione dei pacchetti persi, mantenendo tuttavia il protocollo leggero e veloce.

Le funzionalità base di *RakNet* sono accessibili attraverso l'interfaccia *RakPeerInterface*, che fornisce metodi per la connessione ad altri sistemi e l'invio dei dati tra essi.

La prima cosa da fare all'avvio della libreria è chiamare la funzione *Startup()*. Questa funzione genera un identificatore univoco che identifica globalmente l'istanza corrente, apre una o più *socket* che verranno usate per connettersi agli altri sistemi e alloca le risorse per consentire il numero specificato di connessioni, sia in entrata che in uscita.

A questo punto è possibile connettersi ad altri sistemi, utilizzando il loro indirizzo IP e porta. La funzione *Connect()* inizia un tentativo asincrono di connessione con il sistema remoto e ritorna se il tentativo ha avuto successo.

Per ricevere i risultati delle chiamate asincrone effettuate, nonché notifiche e dati dai sistemi remoti, è necessario chiamare la funzione *Receive()* periodicamente. *Receive()* ritorna NULL se non sono stati ricevuti pacchetti dall'ultima chiamata, altrimenti restituisce un puntatore al primo pacchetto ricevuto, contenente informazioni sul mittente, la lungheza complessiva e i dati sotto forma di array di byte. Il primo byte identifica il tipo di pacchetto. *RakNet* utilizza diversi valori per identificare i propri pacchetti

(es. ID_CONNECTION_REQUEST_ACCEPTED identifica le connessioni in uscita accettate) ed è possibile definire ulteriori identificativi da utilizzare per i messaggi generati dall'applicazione.

Dopo aver processato il pacchetto, è necessario deallocarlo utilizzando la funzione *DeallocatePacket()* per liberare le risorse utilizzate.

Per mandare dei dati tramite RakNet, bisogna innnazitutto definire un identificativo da utilizzare per quel tipo di dati. Fatto ciò è possibile utilizzare la funzione *Send()* specificando i dati da inviare in cui il primo byte deve corrispondere all'identificativo scelto, il destinatario, e altri parametri come la priorità e l'affidabilità desiderata del pacchetto da inviare. È possibile inviare strutture arbitrarie tramite la funzione *Send()*, oppure utilizzare la classe *BitStream* per inviare in modo più efficiente dati eterogenei e di lunghezza variabile.

I dati inviati vengono ricevuti tramite la funzione *Receive()*. Ricevuto un pacchetto, è necessario controllare il primo byte dei dati per verificare se corrisponde a un messaggio che deve essere gestito dall'applicazione. In seguito, la struttura dati inviata può essere immediatamente recuperata tramite semplice casting. Nel caso un *BitStream* venga creato dai dati del pacchetto, i dati contenuti in esso devono essere letti nello stesso ordine in cui sono stati scritti.

I pacchetti possono essere inviati con diverse priorità: IMMEDIATE_PRIORITY, HIGH_PRIORITY, MEDIUM_PRIORITY e LOW_PRIORITY. IMMEDIATE_PRIORITY è la più elevata e causa l'invio istantaneo del pacchetto, gli altri livelli vengono processati ogni 10 millisecondi e inviati insieme per risparmiare banda.

Oltre alla priorità, è possibile definire l'affidabilità di un pacchetto,che può avere uno dei seguenti valori: UNRELIABLE, UNRELIABLE_SEQUENCED, RELIABLE, RELIABLE_ORDERED e RELIABLE_SEQUENCED. UNRELIABLE invia i pacchetti direttamente tramite UDP, cosa che può portare a perdita di pacchetti ma è più veloce e richiede meno banda. RELIABLE gestisce internamente il rinvio dei pacchetti andati persi. SEQUENCED scarta i pacchetti inviati prima dell'ultimo pacchetto ricevuto, infine ORDERED bufferizza i pacchetti e li consegna in ordine.

4.1.1 I plugin di RakNet

RakNet dispone di un sistema di plugin in grado di intercettare ed eventualmente modificare i pacchetti ricevuti prima che vengano ritornati all'utente. è possibile creare un plugin estendendo la classe *PluginInterface2* ed implementando i metodi virtuali *OnReceive()*, *OnClosedConnection()* e *OnNewConnection()*, che vengono chiamati rispettivamente alla ricezione di un pacchetto, alla disconnessione di un sistema remoto e alla connessione di un nuovo sistema. Inoltre RakNet dispone di molti plugin già pronti in

grado di svolgere diverse funzionalità utili, tra cui il *Nat Punchthrough* e il *matchmaking*.

Per utilizzare un plugin di *RakNet* è sufficente istanziarlo e collegarlo all'istanza di *RakNet* tramite la funzione *AttachPlugin()*. A questo punto (dopo aver chiamato *Startup()*) è possibile utilizzare le funzionalità del plugin, implementate tramite funzioni specifiche da chiamare o tramite callback alla ricezione di nuovi pacchetti.

4.1.2 NAT Punchthrough

Il NAT (Network Address Translation)[4, 17, 18, 19, 20, 21, 22, 5, 6] è una tecnica utilizzata per permettere a più dispositivi di connettersi ad Internet utilizzando un solo indirizzo IP. Un router che implementa questa tecnica contiene una tabella di corrispodenze tra le coppie indirizzo IP/porta utilizzate all'interno della rete locale con le coppie corrispondenti da utilizzare sulla rete pubblica.

I dispositivi all'interno di un NAT hanno un indirizzo IP *privato* visibile solo all'interno della rete locale in cui si trovano ed un indirizzo IP *pubblico* che corrisponde a quello del router, che viene utilizzato per le comunicazioni con l'esterno.

Il funzionamento è il seguente: ipotizziamo che il client, con indirizzo IP privato 192.168.1.1, voglia connettersi ad un server esterno con indirizzo 1.2.3.4. Il router NAT ha indirizzo pubblico 2.3.4.5.

Quindi, il client invia un pacchetto con indirizzo sorgente 192.168.1.1:1111 e destinazione 1.2.3.4:2222. Il router, prima di instradare il pacchetto, sostituisce l'indirizzo sorgente con il proprio indirizzo, utilizzando una porta ancora libera. il pacchetto uscente ha indirizzo sorgente 2.3.4.5:3333 e destinazione 1.2.3.4:2222. Il pacchetto arriva al server.

Il server risponde, inviando un pacchetto con indirizzo sorgente 1.2.3.4:2222 e destinazione 2.3.4.5:3333. Il router riceve il pacchetto e controlla se l'indirizzo 2.3.4.5:3333 corrisponde a qualche dispositivo nella rete interna. Quindi procede a sostituire l'indirizzo di destinazione. Il pacchetto inoltrato avrà così indirizzo sorgente 1.2.3.4:2222 e destinazione 192.168.1.1:1111. il client riceve la risposta del server.

Se invece il primo pacchetto proviene dall'esterno, il router non ha modo di sapere a chi deve inoltrarlo in quanto potrebbero esserci più dispositivi collegati alla rete interna, mentre gli indirizzi privati sono validi solo all'interno della rete interna e non possono essere utilizzati fuori da essa.

In questo modo è possibile per un client all'interno della rete privata iniziare uno scambio di dati bidirezionale con un server esterno, ma non è possibile per un client esterno connettersi ad un server interno senza modificare opportunamente la configurazione del router. Allo stesso modo non è possibile per due peer dietro NAT diversi tra loro di comunicare direttamente, in quanto i rispettivi router bloccano le connessioni in entrata provenienti

dall'esterno e quindi le connessioni in uscita vengono bloccate dal router di destinazione.

Si possono distinguere diversi tipi di NAT, a seconda del grado di permissività dell'algoritmo di traduzione del router e quindi della facilità con cui si riesce ad effettuare il *Nat Punchthrough*:

Full Cone NAT: accetta ogni pacchetto diretto verso una porta che è già stata usata.

Address-Restricted Cone NAT: accetta un pacchetto verso una porta se l'indirizzo IP sorgente corrisponde ad un indirizzo verso il quale abbiamo già inviato un pacchetto.

Port-Restricted Cone NAT: accetta un pacchetto verso una porta se l'indirizzo IP e la porta sorgente corrisponde ad un indirizzo e porta verso il quale abbiamo già inviato un pacchetto. La stessa coppia indirizo IP/porta interni verso una destinazione diversa utilizza la stessa coppia indirizzo IP/porta esterni.

Symmetric NAT: accetta i pacchetti come in Port-restricted. Una porta diversa viene allocata per ogni diversa destinazione.

La tecnica del *Nat Punchthrough* permette, con l'ausilio di un server che possiede un indirizzo IP pubblico, ad un client esterno di connettersi ad un server interno ad una rete dotata di NAT, oppure a due peer dietro due diversi NAT di comunicare tra loro. È basata su tentativi simultanei di connessione, in modo che se A tenta di connettersi a B e contemporaneamente B tenta di connettersi ad A, se il tipo di NAT lo permette allora è possibile che almeno uno dei pacchetti di connessione riesca a raggiungere la destinazione, in quanto è stato inviato in precedenza un pacchetto verso il sistema remoto da cui proviene.

RakNet implementa la funzionalità di Nat Punchthrough tramite i plugin *NatPunchthroughClient* e *NatPunchthroughServer*. L'algoritmo utilizzato è il seguente:

1. Il sistema P1 vuole connettersi al sistema P2, entrambi sono già connessi ad un server F con IP pubblico.

2. P1 chiama *OpenNAT()* passando come parametri il *RakNetGUID* (identificatore univoco) di P2 e l'indirizzo di F.

3. la procedura fallisce se P2 non è connesso ad F, o sta già tentando di effettuare il NAT Punchthrough verso P1.

4. F memorizza lo stato di P1 e P2. Se P1 o P2 sono occupati, la richiesta viene messa in coda. Altrimenti F richiede la porta esterna utilizzata più di recente da P1 e da P2. P1 e P2 vengono marcati come occupati.

5. Se P1 o P2 non rispondono il Nat Punchthrough fallisce con ID_NAT_TARGET_UNRESPONSIVE e lo stato di P1 e P2 viene impostato su libero. Altrimenti, F invia un messaggio di connessione fornito di timestamp a P1 e P2 contemporaneamente.

6. P1 e P2 si comportanoallo stesso modo da questo pounto in avanti. Per prima cosa, si inviano a vicenda diversi datagrammi UDP verso l'indirizzo privato. Quindi provano l'indirizzo IP/porta esterni visti da F. Le porte vengono provate sequenzialmente, fino al valore di MAX_PREDICTIVE_PORT_RANGE.

7. Se ad un certo punto un datagramma arriva dal sistema remoto, il sistema entra nello stato PUNCHING_FIXED_PORT. I datagrammi vengono quindi inviati soltanto a quella combinazione IP/porta per il resto dell'algoritmo. Se la nostra risposta arriva al sistema remoto, il NAT Punchthrough viene considerato bidirezionale e ID_NAT_PUNCHTHROUGH_SUCCEEDED viene restituito all'utente.

8. Quando il NAT è aperto, oppure se abbiamo esarito le porte, P1 e P2 segnalano a F che sono pronti per un nuovo tentativo di Nat Punchthrough.

L'efficacia dell'algoritmo dipende dal tipo di NAT coinvolti. Il router da considerare è il più permissivo.

Full Cone NAT: Il router accetta i pacchetti in entrata in quanto la porta utilizzata è la stessa della connessione al server, quindi il Nat Punchthrough ha successo al primo tentativo.

Address-Restricted Cone NAT: Se i due sistemi trasmettono contemporaneamente, il Nat Punchthrough ha successo al primo tentativo. altrimenti il pacchetto inviato per ultimo riesce a passare.

Port-Restricted Cone NAT: Il comportamento è identico al precedente, tranne se l'altro sistema ha un *Symmetric Nat.*

Symmetric NAT: Dato che la porta scelta dal router è diversa ad ogni connessione, il tentativo di Nat Punchthrough fallisce. Per avere una possibilità di successo, è necessaria la *port-prediction* e che il router scelga le porte in modo sequenziale.

Una volta che il *Nat Punchthrough* ha avuto successo, i sistemi sono pronti per connettersi.

Su alcune configurazioni di router o firewall, il *Nat Punchthrough* non funziona. Per esempio, nel caso di un router che sceglie una porta casuale per ogni nuova connessione in uscita, e accetta pacchetti in entrata solo dalla coppia ip/porta registrata, non è possibile far passare nessun pacchetto. Per far fronte a questi casi, *RakNet* mette a disposizione dei metodi alternativi per inoltrare i pacchetti tra i dispositivi client.

Noi abbiamo utilizzato i plugin *UDP Proxy* per inoltrare i pacchetti tra i client tramite il server.

In breve, tale plugin utilizza un server con IP pubblico controllato da noi per inoltrare i messaggi tra il client mittente e destinatario in modo trasparente. Questo sistema funziona anche per inoltrare messaggi UDP non generati da *RakNet*, anche se è necessario *RakNet* per inizializzare la connessione.

Tale sistema di basa su tre plugin di *RakNet*: *UDPProxyClient* da utilizzare sui client di gioco, *UDPProxyServer* e *UDPProxyCoordinator* da utilizzare sul server.

Utilizzo sul client

UDPProxyClient: questo plugin si occupa di effettuare le richieste verso *UDPProxyCoordinator* per inizializzare l'inoltro dei pacchetti.

UDPProxyClientResultHandler: è necessario estendere questa classe per poter ricevere le notifiche degli eventi generati dal sistema.

L'utilizzo del plugin è abbastanza semplice: si crea un'istanza di UDPProxyClient, si attacca a *RakNet* e si registra l'estensione di *UDPProxyClientResultHandler* che implementa le callback necessarie. Quindi si chiama la funzione *UDPProxyClient::RequestForwarding()* quando necessario. se la procedura ha successo, viene chiamata la callback *UDPProxyClientResultHandler::OnForwardingSuccess()* al completamento dell'operazione. a questo punto è possibile connettersi al client remoto utilizzando l'indirizzo e la porta del proxy nella funzione *Connect()*.

Utilizzo sul server

UDPProxyCoordinator: questo plugin viene implementato sul server e gestisce le richieste provenienti da *UDPProxyClient*. si occupa anche del controllo dei vari *UDPProxyServer*.

UDPProxyServer: questo plugin è quello che effettivamente esegue l'inoltro dei pacchetti, utilizzando la classe UDPForwarder, che è indipendente dal protocollo di *RakNet*. Grazie a ciò è possibile utilizzare *UDPProxyServer* per inoltrare una qualsiasi comunicazione UDP, posto che l'inizializzazione deve essere effettuata tramite *RakNet*.

Il plugin *UDPProxyCoordinator* va installato sul master server, mentre su un numero arbitrario di macchine è possibile installare un'istanza di *RakNet* con il plugin *UDPProxyServer* attaccato. *UDPProxyServer* si connette a *UDPProxyCoordinator* ed effettua il login utilizzando una password preimpostata.

Quando un client effettua la richiesta di inoltro, se sono presenti più *UDPProxyServer* collegati a *UDPProxyCoordinator* ed entrambi i client utilizzano *RakNet*, allora entrambi effettuano un ping verso tutti i proxy disponibili, quindi provano a connettersi a partire dal server che ha la somma dei ping più bassa.

La combinazione di *Nat Punchthrough* e *UDP Proxy* permette ai client di potersi connettere tra di loro, supponendo che i nostri server siano in grado di sostenere tutto il traffico generato.

4.1.3 Lobby2 e Rooms

Per implementare le funzionalità multiplayer richieste, sono stati utilizzati i plugin di *RakNet Lobby2* e *Rooms*. *Lobby2* è un plugin che gestisce un database contenente informazioni su utenti, giochi e partite effettuate, mentre *Rooms* serve a permettere agli utenti connessi di trovarsi e di iniziare partite multiplayer.

Il funzionamento di *Lobby2* si basa sull'invio di messaggi rappresentati da strutture dati contenenti i parametri in ingresso e in uscita dal sistema. Tali messaggi vengono allocati sul client da una *message factory* a partire da un identificativo univoco, quindi vengono inseriti i parametri di ingresso ed infine la struttura viene serializzata ed inviata al server. Sul server è presente una *message factory* corrispondente che alloca i messaggi, li processa invocando delle funzioni virtuali che eseguono le query sul database ed inseriscono i parametri di uscita, infine la struttura viene serializzata ed inviata di nuovo al client. Tale sistema permette di implementare facilmente messaggi personalizzati modificando le strutture già esistenti, oppure creandone di nuove.

Il funzionamento di *Rooms* si basa anch'esso sull'invio di strutture dati predefinite rappresentanti le varie funzionalità implementate dal plugin. Tali funzionalità includono la gestione delle stanze di gioco, l'automatching e l'invio di messaggi di chat in-game. Mentre *Lobby2* è un sistema flessibile e

facilmente estendibile, *Rooms* offre la specifica funzionalità per cui è stato implementato, ovvero permettere ai giocatori di connettersi per giocare.

Architettura del plugin

Lobby2Client: questa è la classe principale da utilizzare sul client. Dopo averla inizializzata chiamando SetServerAddress() dopo essersi connessi al server e SetCallbackInterface() per ricevere i risultati, è possibile inviare i messaggi al server utilizzando la funzione *SendMsg(Lobby2Message *)*. Questa classe in sé non fa altro che serializzare i messaggi ed inviarli al server.

Lobby2Server: questa è la classe principale da utilizzare sul server. Si occupa di procassare i messaggi in arrivo dai client allocando un certo numero di thread e chiamando determinate funzioni virtuali implementate dai messaggi stessi.

Lobby2Callbacks: questa classe definisce le funzioni di callback che verranno chiamate da Lobby2Client quando vengono ricevuti i messaggi inviati al server, processati e rispediti indietro. Le callbacks sono definite sotto forma di overload di *MessageResult()*. È possibile registrare più di una istanza di Lobby2Callbacks in un client, in questo caso ogni messaggio può indicare quale di esse deve essere chiamata, o se devono essere chiamate tutte.

Lobby2MessageFactory: questa classe dispone del metodo *Alloc(Lobby2MessageID)* che dato un identficativo univoco per un tipo di messaggio, genera un'istanza della classe corrispondente. è possibile estendere questa classe per includere nuovi tipi di messaggi o per utilizzare una struttura diversa per un certo tipo di messaggio predefinito.

Per definire nuovi tipi di messaggi è inoltre necessario estendere la classe *Lobby2Callbacks* per includere tali messaggi.

Lobby2Message: questa struttura è la base di tutti i messaggi scambiabili attraverso *Lobby2*. Definisce alcune funzioni virtuali che verranno chiamate da *Lobby2Server* durante il processamento di ogni messaggio.

Ogni messaggio viene allocato (e deallocato) da un'istanza di *Lobby2MessageFactory* attraverso un ID specifico per il tipo di messaggio, definito da un enumeratore chiamato per convenzione L2MID_<Nome del messaggio>, quindi vengono assegnati i parametri di ingresso e il messaggio viene serializzato ed inviato al server. Il server processa il messaggio chiamando le funzioni virtuali

PrevalidateInput(), *ServerPreDBMemoryImpl()*, *ServerDBImpl()* e *ServerPostDBMemoryImpl()*. Tali funzioni hanno il compito di assegnare i parametri di uscita del messaggio e impostarne il risultato (*L2RC_SUCCESS*, o uno specifico codice d'errore). A questo punti il messaggio viene reinviato al client o ignorato. Il client riceve il messaggio e chiama la callback registrata corrispondente.

RoomsPlugin: questo plugin si occupa di gestire gli utenti attivi e di tenere traccia delle richieste di nuove partite multiplayer, filtrate in base a certe caratteristiche, in modo da poter organizzare i giocatori che hanno effettuato richieste compatibili e dare l'avvio alla partita. Questa classe può essere configurata per operare sia da client che da server. Il funzionamento del plugin si basa sulla funzione *ExecuteFunc()*, che permette di inviare ad un sistema remoto un descrittore di una specifica funzione da eseguire.

RoomsPluginCallbacks: questa classe definisce le funzioni di callback da chiamare al ricevimento dei messaggi inviati da un sistema remoto. Tali messaggi si distinguono in funzioni chiamate dal client o notifiche generate dal server. *RoomsPlugin* stesso estende questa classe, e ne implementa le funzioni in modo da operare da server.

RoomsPluginFunc: questa è la base da cui derivano tutte le funzioni di *RoomsPlugin* che possono essere invocate dal client. Il funzionamento è molto simile a *Lobby2Message*, con la differenza che tutte le operazioni sul server vengono svolte in un unico thread e senza accedere a dati persistenti.

RoomsPluginNotification: da questa classe derivano tutte le notifiche che il server invia ai client in risposta ad eventi generati da altri giocatori o dal server stesso.

4.2 Symbian

Symbian[10] è un sistema operativo per dispositivi mobili, progettato per permettere il funzionamento di smartphone di fascia bassa dotati di un singolo core che esegue sia le applicazioni utente che le operazioni relative ai protocolli di telefonia.

Symbian supporta il multitasking preventivo e la protezione della memoria, come avviene anche in altri sistemi operativi moderni.

Le API di Symbian sono tutte asincrone e basate su callback per il ritrovamento dei risultati. Questo permette di ottenere buone prestazioni utilizzando poche risorse.

Esistono due piattaforme basate su Symbian attualmente in commercio: *Series 60* (*S60*) e *Series 40* (*S40*). *S40* permette l'installazione soltanto di applicazioni Java ME, quindi non è compatibile con il nostro lavoro.

Le versioni di S60 attualmente sul mercato sono denominate *Symbianˆ3*, *Symbian Anna* e *Nokia Belle*.

4.2.1 P.I.P.S. & Open C

P.I.P.S[1][12, 11] è una libreria per *Symbian OS* che implementa diverse API presenti nello standard C. Open C[14] è un'estensione di P.I.P.S. per piattaforme *S60* che implementa un maggior numero di API.

Le librerie implementate includono:

libc: La libreria standard C, che include funzioni standard di input/output, operatori sui bit, sulle stringhe e sui caratteri, funzioni di cifratura DES, allocazione di spazio sul file system, gestione del tempo e gestione dei segnali.

libcrypt: Libreria crittografica che contiene funzioni per cifrare blocchi di dati e messaggi, e per generare l'hash delle password.

libcrypto: Questa libreria fornisce servizi utilizzati dalle implementazioni OpenSSL dei protocolli SSL, TLS e S/MIME, e che sono utilizzati anche per l'implementazione di SSH, OpenPGP, ed altri standard crittografici.

libdl: Implementa il linking dinamico delle librerie stile POSIX, il quale estende il modello di caricamento dinamico di *Symbian OS*.

libglib: Libreria di funzioni general-purpose che fornisce, tra le altre cose, diversi tipi di dato, macro, conversioni tra tipi, funzioni per manipolare stringhe e file, e un ciclo di eventi.

libm: Libreria di funzioni matematiche operanti in accordo con le librerie C standard.

libpthread: Implementa l'interfaccia POSIX standard per implementare l'esecuzione di thread multipli all'interno di un singolo processo utente. Include la creazione e distruzione di thread, un'interfaccia verso lo scheduler per stabilire vari parametri di esecuzione dei thread, e meccanismi per fornire al programmatore l'accesso alle risorse condivise e la sincronizzazione dei thread.

libssl: La libreria OpenSSL implementa i protocolli SSL v2/v3 e TLS v1.

[1] Acronimo ricorsivo che sta per P.I.P.S. Is POSIX on Symbian

libz: Questa libreria fornisce funzioni per la compressione e decompressione dei dati in memoria, e per il controllo dell'integrità dei dati non compressi.

4.3 Le librerie Qt

Qt[15, 16] è un framework per lo sviluppo di applicazioni multipiattaforma con interfaccia grafica. Le piattaforme supportate da Qt includono: Windows, GNU/Linux, Mac OS X, Solaris, Symbian e altri.

Qt include un ricco insieme di *Widget*, ovvero i componenti base per la creazione di GUI standard. Fornisce inoltre un meccanismo di segnali e slot per la comunicazione tra differenti oggetti, oltre ad un modello basato su eventi per la gestione dell'input dell'utente (mouse, tastiera o altri).

La base delle funzionalità di Qt è il *meta-object system*. Ogni classe in cui lo sviluppatore vuole utilizzare le funzionalità di Qt deve ereditare dalla classe QObject e includere la macro Q_OBJECT nella definizione della classe. questo permette al *moc*[2] di generare il codice relativo alle funzionalità utilizzate da Qt, tra cui la gestione di segnali e slot.

4.3.1 Segnali e slot

Il meccanismo più importante che fornisce Qt sono i segnali e gli slots, che consentono la comunicazione tra oggetti di tipo diverso senza che questi conoscano la struttura reciproca. È possibile connettere a runtime un segnale a uno slot e quando un oggetto emette quel segnale tutti gli slot ad esso connessi verranno invocati.

Un segnale è un metodo di una classe derivata da QObject definito tramite la parola chiave *signals*, la cui implementazione è a carico del *moc*. All'interno dell'implementazione della classe, è possibile emettere un segnale utilizzando la parola chiave *emit*.

Uno slot è un metodo di una classe derivata da QObject definito tramite la parola chiave *slots*. Può essere utilizzato come una normale funzione, oppure è possibile connettere uno o più segnali ad esso, quindi Qt si occuperà di chiamare tutti gli slot connessi ad un segnale quando esso viene emesso.

La funzione statica *QObject::connect()* viene utilizzata per connettere un segnale ad uno slot. Accetta 4 o 5 parametri: l'oggetto che dovrà emettere il segnale, il segnale da connettere, l'oggetto a cui connettere il segnale e il metodo da chiamare quando il segnale verrà emesso. È possibile connettere un segnale ad uno slot oppure ad un altro segnale.

Segnali e slot vengono specificati rispettivamente tramite le macro SIGNAL() e SLOT(), che prendono la firma di un metodo (il nome del metodo

[2] meta object compiler, un tool che genera il codice necessario al corretto funzionamento delle classi scritte dallo sviluppatore con Qt

e la lista dei parametri) e la rendono visibile a runtime a Qt. Un quinto parametro opzionale permette di specificare alcune modalità avanzate con cui dovrà avvenire la connessione. Per esempio, con *Qt::UniqueConnection* viene eliminata la possibilità di effettuare connessioni duplicate, ovvero connettere due volte lo stesso segnale allo stesso slot per la stessa coppia di oggetti.

Un segnale può essere connesso a qualunque slot che abbia una firma compatibile, ovvero deve accettare gli stessi tipi di parametri nello stesso ordine, oppure un prefisso di essi. Ad esempio un segnale *f(int, float, int)* può essere connesso ad uno slot *g(int, float)*.

4.3.2 GUI

Qt fornisce diverse modalità con cui è possibile implementare l'interfaccia grafica delle applicazioni: *Widgets*, *Graphics View Framework* e *QML*.

Widgets: i Widgets sono componenti modulari e riutilizzabili attraverso cui si possono implementare interfaccie grafiche complesse. Sono particolarmente ottimizzati per sistemi desktop, e forniscono funzionalità sia di base come la creazione e la gestione di finestre e layout, che più sofisticate, come la personalizzazione in base a temi scelti dall'utente.

Graphics View Framework: fornisce una superficie per gestire e interagire con un numero elevato di oggetti grafici 2D personalizzati, ed un widget in grado di visualizare tali oggetti. Include un sistema di propagazione degli eventi in grado di gestire in maniera precisa le interazioni tra gli elementi della scena. gli elementi gestiscono eventi di tastiera, movimenti e click del mouse.

QML: è un linguaggio dichiarativo progettato per descrivere l'interfaccia grafica di un'applicazione. Un'interfaccia QML è specificata come un albero di oggetti con proprietà. QML utilizza JavaScript come linguaggio per specificare il comportamento dell'interfaccia. È possibile scrivere interfacce QML senza conoscere il C++, ma è anche possibile estendere QML con classi scritte in C++ che espongono proprietà, segnali e slot.

4.3.2.1 Widget

I Widget sono i componenti principali utilizzati per implementare interfacce statiche per applicazioni desktop. Esistono Widget predefiniti per la maggior parte delle funzionalità di cui un'applicazione potrebbe aver bisogno per comporre la sua interfaccia, come finestre, bottoni e scrollbar. Inoltre è possibile scrivere dei widget personalizzati estendendo la classe *QWidget* e implementando le funzionalità desiderate. *QWidget* deriva da *QObject*, quindi

è possibile utilizzare i segnali e gli slot come meccanismo di comunicazione tra Widget.

I segnali e gli slot sono il metodo principale con cui i *widget* comunicano tra loro e gestiscono l'input e l'output. Per esempio, connettendo il segnale *clicked()* di un *QPushButton* ad un corrispondente slot di una classe contenente la logica applicativa, quando l'utente clicca quel particolare pulsante l'applicazione effettua l'operazione desiderata.

È possibile connettere lo stesso segnale di diversi widget a diversi slot della logica applicativa, in modo da ottenere diversi comportamenti dai diversi componenti senza modificare né estendere nulla all'interno dei componenti stessi. Allo stesso modo è possibile connettere dei segnali della logica applicativa a degli slot di qualche *widget* per visualizzare l'output dell'applicazione senza che l'applicazione stessa sappia dell'esistenza di tali *widget*.

È possibile inoltre connettere segnali di certi *widget* a slot (o segnali) di altri *widget* per effettuare operazioni che non richiedono logica applicativa (per esempio per visualizzare un valore che l'utente sta selezionando tramite uno slider).

Un *widget* può essere visualizzato come una finestra (*top-level widget*), oppure come componente di un altro *widget*, assegnandogli come *parent* il *widget* in cui deve essere contenuto. In tal caso, è possibile impostarne la posizione in cui dovrà stare utilizzando coordinate assolute, oppure tramite *layout*.

Un *layout* si occupa di posizionare i *widget* o eventuali altri *layout* contenuti in esso secondo delle regole specifiche per quel *layout*, per esempio *QVBoxLayout* posiziona i propri figli in verticale, nell'ordine in cui sono stati inseriti.

4.3.2.2 Il Graphics View Framework

Con il *Graphics View framework* è possibile gestire e visualizzare efficentemente un grande numero di elementi grafici. Si basa sui seguenti componenti principali:

QGraphicsView: è un *widget* in grado di visualizzare gli elementi contenuti in una scena o in una parte di essa. Può essere usato da solo oppure in combinazione con altri *widget* per creare un'interfaccia esterna alla scena. Più *QGraphicsView* possono visualizzare la stesa scena. Questa classe si occupa inoltre di ricevere gli eventi di mouse e tastiera, e convertirli in eventi nella scena, trasformandone all'occorrenza le coordinate.

È possibile utilizzare una matrice di trasformazione per trasformare le coordinate della scena, e permettere alla *QGraphicsView* di effettuare operazioni di navigazione avanzate come zoom e rotazioni. *QGraphicsView*

fornisce inoltre delle funzioni di convenienza per mappare le coordinate tra vista e scena.

QGraphicsScene: è una classe in grado di contenere e gestire un numero arbitrario di semplici elementi grafici, che possono essere posizionati e animati in modo efficiente all'interno della scena. La scena si occupa di operazioni quali fornire un accesso efficente agli elementi contenuti, propagare gli eventi attraverso gli elementi, e gestire lo stato degli elementi, ovvero la selezione e il focus.

La scena agisce come un contenitore di elementi di tipo *QGraphicsItem*, che vengono aggiunti alla scena tramite la funzione *addItem()* e possono essere ritrovati tramite una delle molte funzioni disponibili. la funzione *items()* permette di ritrovare gli elementi contenuti o che intersecano un rettangolo, un punto, un poligono o un *vector path* generico.

La scena si occupa di pianificare la consegna degli eventi agli elementi, e gestisce la propagazione degli eventi tra gli elementi stessi. Se la scena riceve un evento del mouse a una certa posizione, lo inoltra all'elemento visibile in quella posizione, se esiste.

QGraphicsItem: è la classe base da cui derivano tutti gli elementi grafici della scena. Il *Graphics View Framework* fornisce elementi standard come rettangoli, ellissi e testo, ma le funzionalità più potenti sono disponibili estendendo questa classe e implementando un elemento personalizzato. *QGraphicsItem* supporta le seguenti funzionalità: eventi del mouse, eventi di tastiera e focus, drag and drop, raggruppamento degli elementi, e *collision detection*.

Gli elementi hanno un proprio sistema di coordinate locali, e forniscono funzioni in grado di mappare le coordinate tra l'elemento e la scena, o tra l'elemento ed altri elementi. È possibile inoltre trasformare le coordinate degli elementi tramite la matrice di trasformazione, permettendo così di gestire rotazioni e zoom dei singoli elementi.

Gli elementi possono contenere elementi figli, che ereditano le trasformazioni del parent. Le funzioni di un elemento operano in coordinate locali.

Il rilevamento delle collisioni è gestito in due modi: tramite la funzione *shape()* e tramite la funzione *collidesWith()*, che sono entrambe virtuali. sovrascrivendo la funzione *shape()* in modo che ritorni la sagoma dell'elemento sotto forma di *QPainterPath*, la scena si occupa del rilevamento delle collisioni. Oppure è possibile reimplementare la funzione *collidesWith()* con il proprio algoritmo.

Coordinate: Il *Graphics View Framework* è basato su un sistema di coordinate cartesiane: la posizione e la geometria degli elementi sono rap-

presentate da due coordinate x e y. Quando la scena è visualizzata da una vista senza trasformazioni, una unità nella scena è rappresentata da un pixel a schermo. Esistono tre sistemi di coordinate: relative all'elemento, alla scena e alla vista. Il *Graphics View Framework* formisce funzioni per mappare le coordinate tra i vari sistemi.

Gli elementi operano in un sistema di coordinate locali. Queste coordinate sono solitamente centrate nel centro dell'elemento (0, 0), che è anche il centro di tutte le trasformazioni. Quando viene implementato un elemento personalizzato, tutto viene gestito in coordinate locali: la scena effettua le conversioni necessarie automaticamente. Ciò semplifica l'implementazione di elementi personalizzati. La posizione di un elemento è definita come le coordinate del punto centrale dell'elemento nel sistema di coordinate del *parent*. Se un elemento è privo di *parent* la sua posizione è relativa al sistema di coordinate della scena.

Le coordinate degli elementi sono relative al *parent*, ovvero se un elemento è privo di trasformazioni la differenza tra le coordinate di un punto relative all'elemento e al suo *parent* è pari alla distanza tra i due oggetti nel sistema di coordinate del padre. Grazie a questo le coordinate di un elemento sono indipendenti dalle trasformazioni del padre, anche se tali trasformazioni si riflettono implicitamente nell'elemento.

Le coordinate relative alla scena rappresentano la base per le coordinate di tutti i suoi elementi. Questo sistema descrive la posizione di ogni elemento privo di parent e forma le base per tutti gli eventi che la scena riceve dalla vista.

Nelle coordinate relative alla vista, ogni unità corrisponde a un pixel. Queste coordinate sono relative al widget della vista, e sono indipendenti dalla scena.

Il *Graphics View Framework* fornisce funzioni per mappare tra i vari sistemi di coordinate: da un elemento alla scena, al parent o a un altro elemento, oppure dalla scena alla vista e viceversa. Queste funzioni accettano come parametri punti, rettangoli, poligoni e path.

4.3.2.3 Qt quick

Qt Quick è una collezione di tecnologie progettate per consentire agli sviluppatori di creare interfaccie grafiche fluide, moderne, intuitive e adatte alle applicazioni per dispositivi mobili. *Qt Quick* consiste in un ampio insieme di elementi grafici, un linguaggio dichiarativo per descrivere le interfaccie e un ambiente di esecuzione per tale linguaggio. Una collezione di API C++ è utilizzata per integrare tali funzionalità di alto livello con le applicazioni Qt classiche. Dalla versione 2.1 di Qt Creator sono disponibili strumenti per facilitare lo sviluppo di applicazioni *Qt Quick*.

Il linguaggio QML: *QML* è un linguaggio interpretato di alto livello basato su *JavaScript* e *CSS*. I suoi comandi, detti elementi, fanno uso delle librerie Qt per permettere l'utilizzo intuitivo delle funzionalità del linguaggio. È possibile disegnare elementi grafici, visualizzare immagini o gestire gli eventi dell'applicazione in maniera dichiarativa. Inoltre è possibile utilizzare *JavaScript* per implementare funzionalità di alto livello relative all'interfaccia.

Un elemento *QML* ha varie proprietà che definiscono quell'elemento. Poter costruire interfaccie mettendo insieme diversi elementi è una delle principali funzionalità di *QML*.

La sintassi di *QML* è abbastanza semplice e intuitiva:

```
import QtQuick 1.0
 Rectangle {
     width: 200
     height: 200
     color: "blue"
     Image {
         source: "pics/logo.png"
         anchors.centerIn: parent
     }
 }
```

Questo codice definisce due oggetti, di cui il primo di tipo RECTANGLE e uno di tipo IMAGE dichiarato avente come parent il primo elemento, e ne definisce alcune proprietà quali il colore del rettangolo o il file immagine da visualizzare.

Gli oggetti sono specificati dal loro tipo, seguito da parentesi graffe. I nomi dei tipi iniziano sempre con una lettera maiuscola. All'interno delle parentesi è possibile specificare le proprietà degli oggetti.

Le proprietà sono specificate come NOMEPROPRIETÀ: VALORE. Ogni oggetto ha delle determinate proprietà. Il nome della proprietà è separato dal valore dal carattere ":". Le proprietà possono essere specificate una per riga oppure è possibile specificare più proprietà in una riga separandole con il carattere ";".

Il comando IMPORT importa il modulo *QtQuick*, che contiene tutti gli elementi *QML* standard, tra cui gli elementi RECTANGLE e IMAGE.

Oltre a poter assegnare valori alle proprietà, è possibile assegnare espressioni *JavaScript* arbitrarie. tali espressioni possono contenere riferimenti ad altre proprietà di altri oggetti, in questo caso viene stabilito un *property binding*: ogniqualvolta il valore del'espressione cambia, automaticamente viene aggiornato il valore assegnato alla proprietà.

Una proprietà speciale è la proprietà *id*, che definisce l'identificativo dell'oggetto. Un oggetto può essere indirizzato tramite il proprio *id* da tutti gli oggetti nello stesso componente. Di conseguenza, un *id* deve essere uni-

voco all'interno del componente in cui è definito. A differenza delle normali proprietà, non è possibile accedere al valore del'*id* di un oggetto e una volta che l'oggetto è stato creato non è possibile modificarne l'*id*.

QML supporta proprietà di diversi tipi. I tipi di base includono int, real, bool, string e altri. Le proprietà sono type-safe, ovvero non è possibile assegnare un valore di un tipo a una proprietà di un altro tipo (per esempio non è possibile assegnare una stringa a una proprietà di tipo int).

Gli elementi *QML* supportano i segnali di Qt. La connessione con un segnale viene effettuata assegnando un blocco di codice JavaScript a un *signal handler*, con la stessa sintassi delle proprietà. il nome di un *signal handler* è sempre nella forma on<signal>.

```
Item {
    width: 100; height: 100
    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("mouse button clicked")
        }
    }
}
```

Alcuni *signal handlers* includono un parametro opzionale. Per esempio il *signal handler* onClicked dell'elemento MouseArea possiede un parametro mouse che contiene informazioni sull'evento del mouse, come il pulsante premuto o la posizione del mouse.

Un blocco di codice *QML* è detto *documento QML*. I documenti QML corrispondono generalmente a file salvati su disco o in rete, ma possono anche essere costruiti a runtime a partire da dati testuali. Un documento QML è sempre codificato in UTF-8.

Un documento QML definisce un singolo *componente QML top-level*. Un componente QML è un modello che viene interpretato dall'ambiente di esecuzione per creare un oggetto con comportamenti predefiniti. un songolo componente QML può essere utilizzato per creare più oggetti, chiamati *istanze* di quel componente. Qualunque blocco di codice QML può diventare un componente, è sufficiente salvarlo in un file chiamato "<Nome>.qml", dove <Nome> è il nome che si vuole dare al componente, che deve iniziare con una lettera maiuscola. Il componente così creato è automaticamente disponibile come elemento QML negli altri componenti che si trovano nella stessa cartella.

Oltre ai componenti top-level definiti dai documenti QML, è possibile definire un componente *inline*. i componenti *inline* vengono dichiarati tramite l'elemento Component. i componenti inline hanno tutte le caratteristiche dei componenti top-level.

Qt Declarative: è il modulo introdotto da Qt che fornisce l'ambiente di esecuzione necessario per interpretare il linguaggio *QML* tramite un motore basato su Qt. Dal momento che *Qt Declarative* e *QML* si basano su Qt, ereditano molte delle tecnologie di Qt, tra cui il meccanismo di segnali e slot e il *meta-object system*. Dati creati utilizzando C++ sono accessibili da *QML* e oggetti *QML* sono accessibili dal codice C++.

Insieme al linguaggio *QML*, il modulo *Qt Declarative* separa l'interfaccia utente di un'applicazione dalla sua logica applicativa.

4.3.3 Strutture Dati

Qt fornisce un'ampia gamma di strutture dati e di contenitori in grado di gestire i dati di un'applicazione in maniera sicura e meno esposta a errori rispetto ad utilizzare i puntatori e gli array del C++.

I contenitori sono basati su templates e forniscono un'interfaccia veloce e fondamentalmente sicura per iterare attraverso gli oggetti contenuti. Sono ottimizzati per essere leggeri e veloci, forniscono iteratori in stile *Java* e in stile *STL*, e usano il meccanismo di *implicit sharing* per ottimizzare la velocità e il consumo di memoria dell'applicazione. Qt fornisce i seguenti contenitori: *QList*, *QLinkedList*, *QVector*, *QStack*, e *QQueue* forniscono dei contenitori sequenziali, mentre *QMap*, *QMultiMap*, *QHash*, *QMultiHash*, e *QSet* forniscono contenitori associativi.

Qt introduce la keyword *foreach* come estensione al linguaggio C++, implementata utilizzando il preprocessore standard. è quindi possibile scrivere

```
QList<QString> list;
...
foreach (const QString &str, list)
    cout << str.ascii() << endl;
```

per accedere agli elementi di una lista, senza dover scrivere codice aggiuntivo e senza rischio di errori dovuti a disattenzione o utilizzo sbagliato di indici. Come un normale ciclo *for*, supporta le parentesi graffe e le istruzioni *break* e *continue*.

Oltre ai contenitori, Qt fornisce le classi *QByteArray* e *QString* per la gestione di stringhe e blocchi di dati binari. Tali classi forniscono funzioni avanzate per la gestione e manipolazione delle stringhe come il confronto case-insensitive, la ricerca di sottostringhe o l'inserimento di parametri al posto di speciali marcatori in una stringa.

Qt fornisce degli strumenti per gestire correttamente i puntatori:

- *QPointer<T>*, dove T deriva da Qobject, fornisce un puntatore con guardia: se e quando l'oggetto puntato da esso viene distrutto, il valore viene automaticamente settato a *NULL*.

- *QSharedPointer* e *QScopedPointer* si occupano invece di deallocare l'oggetto puntato non appena essi escono dallo scope: *QScopedPointer* distrugge l'oggetto allocato non appena il puntatore viene distrutto, mentre *QSharedPointer* utilizza un metodo di *reference counting* per distruggere l'oggetto solo quando tutte le reference ad esso sono state distrutte.

Capitolo 5

La Piattaforma Sviluppata

In questo capitolo descriviamo l'architettura del sistema multiplayer sviluppato. Esso si appoggia su *RakNet* 4.04 per la gestione del protocollo di rete, e implementa le varie funzionalità descritte di seguito.

Il sistema sviluppato è suddiviso in client e master server. Il master server si occupa di gestire il database degli utenti e dei punteggi, nonché della connessione tra i client per le partite multigiocatore. I client si connettono al master server per trovare gli avversari, in seguito la comunicazione avviene Peer to Peer.

5.1 Client

Il client è stato scritto in C++, utilizzando la libreria RakNet e i suoi plugin per il supporto alla comunicazione di rete. Fornisce un'interfaccia semplice verso le applicazioni, occupandosi di tutte le procedure complesse necessarie per la connessione tra i giocatori e l'inizio delle partite multigiocatore.

5.1.1 Porting di RakNet su Symbian

Uno degli obiettivi del nostro lavoro è stato il funzionamento della libreria su sistemi *Nokia Symbian*, in quanto non esistono implementazioni di piattaforme multiplayer per questa tipologia di dispositivi. Lo stesso *RakNet* non supporta ufficialmente *Symbian*, ma è stato possibile farlo funzionare correttamente con poche modifiche interne.

5.1.1.1 Porting: modifiche effettuate al codice

Utilizzando le librerie Open C, è stato possibile effettuare il porting della piattaforma RakNet, ad eccezione di alcune estensioni non necessarie, verso la piattaforma Symbian con un numero minimo di modifiche effettuate all'interno del codice.

Le modifiche principali effettuate riguardano:

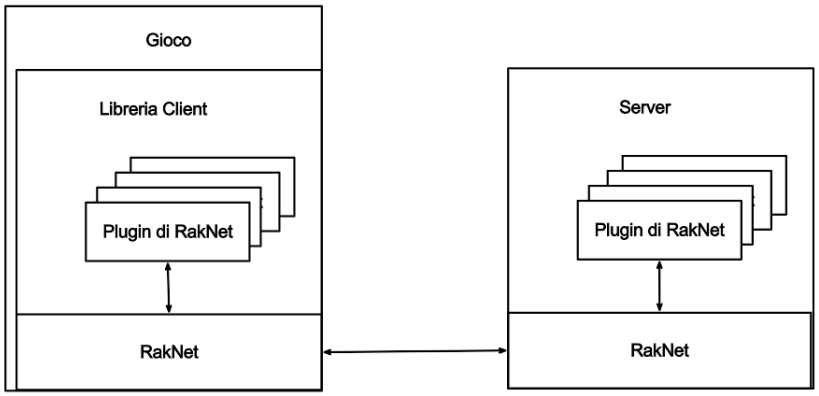

Figura 5.1: Architettura del nostro framework

- Alcune parti dipendenti dalla piattaforma hardware, principalmente relative alle operazioni atomiche sui dati, gestite in modo totalmente differente tra i diversi sistemi. Principalmente abbiamo scelto di riutilizzare il codice presente per altre piattaforme mobili (Android o iOS).

- la funzione *alloca()* utilizzata per allocare dinamicamente risorse sullo stack. Questa operazione non è possibile in *Symbian OS* in quanto la dimensione dello stack è tra le risorse più limitate. Pertanto abbiamo sostituito nel codice le chiamate ad *alloca()* a chiamate a *malloc()* e corrispettive chiamate a *free()*.

5.1.2 Estensioni a RakNet

RakNet permette di scrivere facilmente dei plugin in grado di fornire funzionalità aggiuntive. Alcuni dei plugin inclusi possono essere ulteriormente estesi in modo pulito e sicuro.

Nello sviluppo della libreria abbiamo implementato funzionalità aggiuntive per la gestione di classifiche personalizzabili per ogni gioco, utilizzando il plugin *Lobby2* ed implementando nuovi tipi di messaggi. inoltre abbiamo esteso *RoomsPlugin* sul server implementando una funzionalità di tracciamento del numero di richieste effettuate.

5.1.2.1 Classifiche

Il sistema di classifiche implementato permette di memorizzare varie statistiche relative ai giocatori (es. punteggio, partite vinte o partite giocate) all'interno del database del server, e di poter lanciare query personalizzabili per costruire una classifica relativa a ogni gioco basata su una qualunque combinazione di tali parametri, per esempio è possibile avere una classifica ordinata in base alla percentuale di vittorie suddivisa in scaglioni in base alle partite giocate.

Tabelle aggiunte al database: per la gestione delle classifiche è stata aggiunta al database di *Lobby2* un'unica tabella *playerscores* contenente, per ogni gioco e per ogni utente, fino a otto valori numerici rappresentanti i punteggi ottenuti da quel giocatore in quel gioco. Ogni gioco può gestire tali valori in modo arbitrario.

Classi implementate

Classifica_SubmitScore: questa classe estende *Lobby2Message* e permette di inviare al server il punteggio ottenuto durante una partita. il messaggio contiene i nomi dei giocatori e, per ciascuno di essi, un certo numero di operazioni da effettuare sui vari campi. Ogni operazione può essereaggiungere al valore contenuto nel campo un certo valore o assegnare un valore specificato al campo di riferimento.

SubmitPlayerInfo: questa struttura incapsula uno username di un giocatore e la relativa lista di operazioni da effettuare sui suoi dati.

SubmitOperation: questa struttura descrive un'operazione che può essere eseguita sul server. Essa contiene l'ID dell'operazione, che può essere un incremento o un assegnazione, un descrittore del campo da modificare, ovvero un carattere da 'A' a 'H', e un intero contenente il valore da assegnare o aggiungere al campo dpecificato.

Classifica_GetClassifica: questa classe estende *Lobby2Message* e permette di richiedere al server una classifica dei giocatori.

I parametri in ingresso sono:

userName: il nome dell'utente che richiede la classifica.

gameTitle: il nome del gioco di cui si vuole ottenere la classifica.

season: un ulteriore filtro usato per poter avere un maggior numero di classifiche per gioco, tipicamente per suddividerle per periodo.

limit: il numero massimo di posizioni da restituire

fetchColumns: quali campi restituire. Ogni bit rappresenta un campo, se impostato a 1 il campo corrispondente viene restituito e può essere incluso nei filtri successivi.

sortBy: questa stringa rappresenta l'ordinamento desiderato per la classifica, può contenere una semplice espressione matematica. I campi sono referenziati con lettere maiuscole da 'A' a 'H'.

filter: questa stringa rappresenta eventuali filtri aggiuntivi che possono essere applicati alla classifica. È possibile utilizzare espressioni matematiche e referenziare i campi come per *sortBy*.

I parametri restituiti dal server sono i seguenti:

classifica: la classifica generata, contenente il nome utente e i punteggi selezionati.

notInRanking: se l'utente che ha richiesto la classifica non è nelle prime posizioni, viene generato un ulteriore record e questo campo viene impostato a true

yourPosition: questo valore indica la posizione del giocatore che ha richiesto la classifica.

yourData: questo campo contiene i dati del giocatore che ha richiesto la classifica, se non è presente nelle prime posizioni.

PlayerInfoClassifica: questa struttura contiene un nome utente e una lista di interi rappresentanti i valori dei campi richiesti per la classifica relativi a quel nome utente. Essa viene usata per restituire ogni posizione della classifica.

5.1.3 Interfaccia applicativa lato client

Tutte le funzionalità di cui abiamo parlato finora sono state incapsulate in un'unica classe che implementa e gestisce automaticamente tutte le varie fasi della connessione e dello scambio di dati, nascondendo la complessità interna di *RakNet* ed esponendo solo le funzioni relative alle funzionalità controllabili dall'utente, in modo da renderne facile l'utilizzo.

Architettura

MixelGames::Core: Questa classe è il nucleo della libreria. Essa contiene un riferimento all'istanza di RakNet in esecuzione, implementa il ciclo di update e le callback dei vari plugin e si occupa della gestione dello stato di connessione al server e agli altri client, lasciando l'utente con il solo compito di inviare e ricevere i dati relativi al gioco stesso.

MixelGames::Callbacks: Questa classe definisce le funzioni di callback chiamate da *MixelGames::Core* al verificarsi di certi eventi, ovvero il completamento della connessione al server o ai client, che sia andata a buon fine o meno, e la ricezione di messaggi e dasti relativi al gioco.

MixelGames::DebugCallback: questa classe definisce delle funzioni virtuali che vengono chiamate internamente alla libreria per stampare messaggi di debug o di errore. Se si vuole ricevere l'output di tali messaggi, è sufficiente estandere *MixelGames::DebugCallback* e implementare almeno la funzione *log()* tramite le API di debug relative alla piattaforma.

debug_core.h: in questo header sono definite delle macro e strutture che abbiamo utilizzato internamente per facilitare le operazioni di debug e tracciamento. In particolare, abbiamo definito diversi livelli di debug, a seconda dei quali vengono stampati messagi di errore, log o anche l'entrata e uscita da ogni funzione. Ciò è stato necessario per trovare e risolvere alcuni errori comparsi su Symbian, ma non su altre piattaforme.

SubmitInfo, FetchInfo, EntryClassifica: queste strutture sono dei wrapper per le funzioni relative alla classifica (sottomissione e ritrovamento) prive di dipendenze interne a *RakNet*, inoltre possono essere estese per includere controlli e semplificazioni sulle espressioni di ordinamento e filtraggio da utilizzare.

mg_enums.h: in questo header sono definiti tutti i tipi enum che la libreria espone all'applicazione. Essi rappresentano lo stato di connesisone al server e agli altri client, errori di rete e i risultati delle operazioni di login e di registrazione di account. Inoltre sono presenti delle funzioni che convertono ogni valore di tali enum in una stringa contenente il nome dell'enum stesso, informazione utile per il debug.

QtMixelGames: questa classe è stata implementata utilizzando le librerie Qt e rappresenta l'interfaccia tra la libreria multiplayer che abbiamo sviluppato e il resto dell'applicazione. Contiene un puntatore a *MixelGames::Core* i cui metodi sono mappati 1:1 ai rispettivi metodi di questa classe, e le cui callback sono mappate verso segnali Qt. QUesto approccio inoltre

consente di poter includere la libreria nell'applicazione tramite static linking mantenendo al minimo le dipendenze esterne.

L'intera libreria viene utilizzata tramite i metodi e i segnali di questa classe. Riportiamo di seguito i più importanti.

getInstance(): restituisce un puntatore all'istanza in esecuzione, creandola se necessario.

connectToServer(): avvia la connessione verso il server.

disconnect(): chiude tutte le connessioni.

registerNewUser(): registra un nuovo utente tramite Lobby2. è necessario fornire nome utente e password, e un'indirizzo email opzionale.

login(): effettua il login al server tramite lobby2 utilizzando nome utente e password forniti.

logout(): effettua il logout dal server.

startAutomatch(): avvia la procedura di automatching, ovvero cerca altri giocatori connessi tramite *RoomsPlugin*, quando ne trova abbastanza elegge un host tra di essi, i giocatori si connettono all'host, effettuando il Nat Punchthrough se necessario e ricorrendo a UDP Proxy in caso di fallimento. A procedura completata, viene emesso il segnale *readyForPlay()*, quindi è possibile scambiarsi messaggi di gioco.

cancelAutomatch(): annulla un automatch precedentemente iniziato.

endMatch(): comunica al server che una partita è finita e, se host, invia i punteggi dei giocatori al server per l'aggiornamento delle classifiche.

sendGameData(): invia un messaggio arbitrario ai giocatori connessi. i destinatari riceveranno il messaggio tramite il segnale *dataReceived()*.

5.2 Server

Il master server implementato è un semplice programma che crea un'istanza di *RakNet*, inizializza i vari plugin utilizzati ed infine entra nel ciclo di update dove riceve i messaggi in entrata da *RakNet*, i quali vengono processati internamente dai plugin, e scrive su un file un log delle connessioni in entrata.

I plugin utilizzati dal master server sono:

NatPunghthroughServer questo plugin si occupa di coordinare il Nat Punchthrough tra i client che ne fanno richiesta.

UDPProxyCoordinator nel caso il Nat Punchthrough fallisca, questo plugin si occupa di allocare un proxy UDP che faccia da tramite ai client.

UDPProxyServer questo plugin inoltra il traffico proveniente dai client che non sono riusciti a connettersi direttamente. è possibile installare questo plugin anche su altre macchine per alleggerire il carico sul server nel caso dovesse diventare troppo pesante.

Lobby2Server questo plugin si occupa di gestire il database degli utenti e delle partite. Abbiamo esteso questo plugin per implementare le nostre classifiche personalizzate.

RoomsPlugin questo plugin si occupa di mettere in contatto i giocatori che vogliono giocare partite multiplayer. Abbiamo inoltre esteso il plugin per implementare funzionalità di tracciamento delle richieste di partite.

Il Server è stato installato su una macchina virtuale fornita dal servizio di *cloud computing RackSpace*[1].

[1] `http://www.rackspace.com/`

Capitolo 6

Valutazione sperimentale della piattaforma

Per verificare il corretto funzionamento della libreria multiplayer da noi sviluppata, abbiamo sviluppato un'applicazione che usa tale libreria per la componente di gioco multiplayer e per la gestione delle classifiche online. Tale applicazione è stata pensata inizialmente per Symbian e iOS, con possibili sviluppi su Android e Windows Phone.

6.1 Applicazione: Bar Briscola

Bar Briscola[1] è un'applicazione sviluppata per Symbian contestualmente alla libreria multiplayer e parallelamente per iOS integrando tale libreria successivamente. In seguito verrà portata su Android e Windows Phone. Si tratta di un'applicazione che implementa alcuni giochi di carte classici italiani, tra cui Briscola e Scopa. È stata progettata per poter facilmente integrare un numero arbitrario di giochi di carte.

L'applicazione è suddivisa nei componenti relativi a Interfaccia Utente, Regole, Intelligenza Artificiale e Gestione del multiplayer, utilizzando il motore ad eventi personalizzato *mercurio* per la comunicazione tra i componenti.

Di seguito verrà descritta l'architettura dell'applicazione sviluppata per Symbian.

6.1.1 Mercurio

Per consentire la corretta interazione tra i vari componenti dell'applicazione, è stato utilizzato il motore ad eventi *Mercurio*, che abbiamo sviluppato internamente all'azienda per tutte le piattaforme (Symbian, Android e iOS)

[1] Scaricabile dal Nokia Store all'indirizzo `http://store.ovi.com/content/299705`

utilizzando il linguaggio nativo di esse. Di seguito ne verrà descritta l'architettura comune a tutte le piattaforme. Eventuali esempi di utilizzo fanno riferimento alla versione implementata in Qt/C++ per Symbian.

Scopo: Lo scopo di Mercurio è quello di poter dividere le varie parti di un'applicazione e metterle in comunicazione senza che i due interlocutori conoscano l'esistenza e la struttura reciproca. Mercurio è inoltre pensato nel modo più generico possibile per permettere la comunicazione tra qualunque parte senza che Mercurio conosca la loro struttura e la struttura degli eventi scambiati.

Descrizione: La classe Mercurio si occupa dello smistamento degli eventi ricevuti, a tutti coloro che, previa registrazione, sono interessati a ricevere quel determinato evento.

Un evento è un oggetto identificato da un id univoco che può contenere qualsiasi tipo di informazione. Un evento per essere tale deve implementare l'interfaccia *Evento*.

La classe Mercurio dovrà essere implementata come Singleton con costruttore privato e un metodo statico getInstance() che permetterà l'accesso all'istanza di Mercurio.

La classe Mercurio contiene una hash table che manterrà la relazione <*proprietario, Ricevitore*> e che verrà aggiornata dai metodi *registra(idEvento, callback, proprietario)* e *deregistra(idEvento, callback, proprietario)*.

Mercurio implementa un Ricevitore per ogni singolo proprietario. La classe *Ricevitore* gestisce il recapito degli eventi indirizzati a un determinato proprietario accodandoli se necessario. E' definita internamente alla classe Mercurio e non visibile agli oggetti esterni a Mercurio.

La classe Ricevitore è composta da una hash table contenente la coppia <*idEvento, Callback*> aggiornata a ogni chiamata di *registra(idEvento, callback)* e *deregistra(idEvento, callback)*, e una coda eventi, contenente gli eventi indirizzati al proprietario associato al Ricevitore ma non ancora processati.

Per permettere al proprietario di ricevere e processare l'evento, ogni volta che viene chiamato il metodo *inviaEvento(Evento)* lo stato del ricevitore viene impostato a "busy". Sarà il proprietario a cambiare lo stato del Ricevitore chiamando il metodo *setReady(proprietario)* della classe Mercurio.

Allo stesso modo ogni nuovo Ricevitore viene istanziato in stato "busy" per permettergli di finire le sue possibili funzioni di inizializzazione. Quando il proprietario sarà in grado di ricevere eventi chiamerà il metodo *setReady(proprietario)* della classe Mercurio per cambiare lo stato del Ricevitore a lui associato da "busy" a "free".

La classe Mercurio contiene inoltre un contatore delle richieste di "lock" e "unlock" che mantiene lo stato di Mercurio, e una coda di eventi che mantiene gli eventi in entrata.

A ogni chiamata al metodo *lock()* il contatore lock viene incrementato partendo da 0 (stato di "unlock"). Al contrario a ogni chiamata al metodo *unlock()* il contatore viene decrementato fino al valore 0. In questo modo Mercurio sarà in stato di "lock" quando il contatore è maggiore di zero e "unlock" quando il contatore è uguale a zero. Questo perchè è possibile che Mercurio sia messo in "lock" da più utilizzatori contemporanenamente e bisogna quindi garantire che tutti abbiano chiamato il metodo *unlock()*.

Ogni evento ricevuto tramite *inviaEvento(Evento)* e *inviaEventoNonAMe(Evento, proprietario)* viene incapsulato nella classe ContenitoreEvento (classe che mantiene la relazione tra l'eventi ricevuto e l'eventuale proprietario/mittente da escludere durante l'invio del messaggio) e inserito nella coda degli eventi. Mercurio inizia quindi a processare il primo evento nella coda. L'evento viene recapitato a tutti i Ricevitori e solo dopo aver chiamato tutte le callback registrate, l'evento viene tolto dalla coda. È possibile che Mercurio si trovi già in fase di processamento, se una callback chiamata da mercurio genera a sua volta uno o più eventi. In questo caso il metodo ritornerà senza processare l'evento, che verrà processato al ritorno dalla callback, evitando così ricorsioni incontrollate che potrebbero causare stack overflow.

Il singolo evento viene processato richiamando il metodo *inviaEvento(Evento)* della classe *Ricevitore* su tutte le istanze contenute nella hash table(proprietario, Ricevitore). Se l'attributo *"escluso"* della classe *ContenitoreEvento* è diverso da NULL il proprietario contenuto nell'attributo dovrà essere escluso dall'invio dell'evento.

Quando Mercurio è in fase "lock", gli eventi inviati tramite inviaEvento(Evento) e inviaEventoNonAMe(Evento, proprietario) saranno inseriti nella coda eventi senza essere processati. Quando Mercurio torna nello stato "unlock" la coda verrà processata come sopra.

La classe *ContenitoreEvento* contiene due attributi: *evento* mantiene l'evento da processare, *escluso* mantiene il proprietario/mittente se l'evento è stato inviato tramite *inviaEventoNonAMe(Evento, proprietario)*, NULL se l'evento è stato inviato tramite *inviaEvento(Evento)*.

Metodi:

Classe *Mercurio*:

Mercurio getInstance() Ritorna l'instanza di Mercurio.
Parametri: nessuno
Ritorna l'istanza di Mercurio

registra(idEvento, callback, proprietario) Registra l'intenzione del proprietario a ricevere l'evento. Il metodo ricerca nella hash teble se al

proprietario è già associato un Ricevitore. Se il Ricevitore già esiste viene chiamato il metodo *registra(idEvento, callback)* sull'istanza Ricevitore trovata passando idEvento e callback ricevuti come input. Se invece il Ricevitore associato al proprietario non esiste viene creata una nuova istanza della classe Ricevitore e viene inserita nella hash table accoppiandola al proprietario ricevuto come input, successivamente viene chiamato il metodo *registra(idEvento, callback)* sull'istanza creata. Infine viene ritornato il valore di ritorno di *registra(idEvento, callback)*.
Parametri: *idEvento:* l'id dell'evento di cui si è interessato ricevere i messaggi; *callback:* la funzione richiamata per l'invio del messaggio; *proprietario:* il proprietario della registrazione.
Ritorna true se la registrazione è andata a buon fine, false altrimenti.

inviaEvento(Evento) Riceve l'evento passato come parametro, inserisce l'evento nella coda eventi. Se la coda è vuota viene processata come descritto sopra. Se la coda non è vuota il metodo ritorna senza processare l'evento.
Parametri: *Evento:* l'evento da inviare

inviaEventoNonAMe(Evento, proprietario) Riceve l'evento passato come parametro, inserisce l'evento nella coda eventi. Se la coda è vuota viene processata come descritto sopra. Se la coda non è vuota il metodo ritorna senza processare l'evento.
Parametri: *Evento:* l'evento da inviare; *proprietario:* l'oggetto che ha inviato l'evento, a cui non deve essere reinviato.

deregistra(idEvento, proprietario) Cancella la registrazione della callback per il dato idEvento cercando l'istanza Ricevitore associata al proprietario e chiamando il metodo *deregistra(idEvento)* sull'istanza Ricevitore trovata. *deregistra(idEvento)* ritornerà il numero di eventi ancora registrati. Se gli eventi registrati al Ricevitore saranno zero la coppia <*prorietario, Ricevitore*> sarà cancellata dalla hash table.
Parametri: *idEvento:* l'id dell'evento da deregistrare; *proprietario:* il proprietario della registrazione

setReady(proprietario) Ricerca l'istanza Ricevitore associata al proprietario. Sull'istanza trovata chiama il metodo *free()* per cambiare lo stato del Ricevitore da "busy" a "free".
Parametri: *proprietario:* il proprietario da sbloccare

lock() Blocca Mercurio inibendo lo smistamento degli eventi ricevuti. Gli eventi ricevuti dopo la chiamata di *lock()* saranno salvati e recapitati solo dopo la chiamata *unlock()*.
Parametri: nessuno.

unlock() Sblocca Mercurio.
Parametri: nessuno.

isLocked() Ritorna lo stato di Mercurio.
Parametri: nessuno.
Ritorno: true se Mercurio è bloccato, false altrimenti.

Classe Ricevitore:

registra(idEvento, callback) Registra l'evento identificato con *idEvento* inserendo la coppia <*idEvento*, *callback*> nella hash table solo se non presente. Non potranno essere registrati due eventi uguali nello stesso Ricevitore. Se la registrazione avviene ritorna true, false altrimenti.
Parametri: *idEvento:* l'id dell'evento di cui si è interessato ricevere i messaggi; *callback:* la funzione richiamata per l'invio del messaggio.
Ritorno: true se la registrazione è avvenuta, false altrimenti.

inviaEvento(Evento) Riceve l'evento passato come parametro e ricerca la callback associata all'idEvento dell'evento ricevuto nella hash table. Setta lo stato del Ricevitore a "busy" e richiama la callback se presente.
Parametri: *Evento:* l'evento da inviare.

deregistra(idEvento) Cancella la registrazione della callback per il dato *idEvento*. Ritorna il numero di coppie <*idEvento, callback*> ancora presenti nella hash teble dopo la cancellazione.
Parametri: *idEvento:* l'id dell'evento da deregistrato.
Ritorno: numero di coppie idEvento, callback ancora presenti nel Ricevitore.

free() Setta lo stato del Ricevitore a "free".
Parametri: nessuno

6.1.2 Il motore delle regole

L'applicazione Bar Briscola è stata concepita inizialmente contenente i giochi di carte Briscola, Scopa e Tressette. In seguito ad un cambiamento degli obiettivi aziendali, L'applicazione conterrà i giochi Briscola, Briscola Chiamata e Scopa, di cui Briscola è già implementata, mente gli altri giochi verranno sviluppati in seguito e rilasciati come aggiornamento dell'applicazione. Inoltre, essendo l'applicazione rilasciata per Symbian e iOS, si è scelto di implementare le regole di gioco utilizzando esclusivamente C++ standard, compatibile con entrambe le piattaforme.

La struttura di gioco si basa su *comandi* che i giocatori inviano alle regole tramite *Mercurio*, i quali vengono processati e convertiti in una lista di *azioni* effettuate sullo stato di gioco, che vengono inviati ai giocatori sempre tramite

Mercurio. A questo punto i giocatori effettuano le operazioni corrispondenti sul loro stato interno.

Classi

L'architettura delle regole è strutturata nelle seguenti classi:

RulesInterface La classe base da cui far derivare le regole di ogni gioco. dispone dei metodi virtuali *initPlayer()* e *gioca()*, che vengono chiamati rispettivamente all'inizializzazione dei giocatori e quando un giocatore effettua una mossa.

RegoleBriscola Questa classe implementa le regole del gioco Briscola.

GameLogic Questa classe interfaccia il motore delle regole (scritto in C++ standard) con Mercurio (scritto usando le librerie Qt). si occupa di inizializzare la classe corrispondente alle regole del gioco richiesto, chiamare i metodi *initPlayer()* e *gioca()* all'arrivo degli eventi corrispondenti ai comandi *Ready for Play* e *Gioca* e inviare tramite mercurio gli eventi corrispondenti ad ogni *azione*.

Comandi

I *comandi* accettati sono i seguenti:

Ready for Play Viene inviato da un giocatore all'inizio della partita per segnalare che ha inizializzato il proprio stato, ed è pronto per ricevere eventi relativi alla partita. contiene inoltre un id temporaneo[2] da utilizzare come identificativo fintanto che non riceve il proprio *playerID* ufficiale per la partita. quando tutti i giocatori hanno inviato il comando *Ready for Play*, la partita ha inizio.

Gioca Viene inviato dal giocatore di turno quando ha deciso la propria mossa. i parametri presenti sono: il *playerID* del giocatore che invia il comando, e l'id della carta che ha intenzione di giocare. Alla ricezione del comando, le regole controllano che la mossa sia valida, quindi ne calcolano gli effetti e generano la lista delle azioni prodotte, che viene inviata ai giocatori.

Azioni

Le *azioni* inviate sono le seguenti:

[2] un intero casuale a 32 bit

Assegna ID Giocatore Viene generata in risposta al comando *Ready for Play*. Contiene l'id temporaneo e il *playerID* del giocatore che ha inviato il comando.

Inizio Partita Viene generata in risposta al comando *Ready for Play*, quando tutti i giocatori hanno inviato il comando. non contiene parametri aggiuntivi.

Estratta Briscola Indica che la briscola è stata estratta. Questa azione viene generata solo nel gioco Briscola

Carta Pescata Indica che una carta è stata pescata. Contiene l'ID del giocatore e della carta.

Carta Giocata Indica che una carta è stata giocata. Contiene l'ID del giocatore e della carta.

Carta Presa Indica che una carta è stata presa. Contiene l'ID del giocatore e della carta, e un flag che indica se è l'ultima azione di questo tipo in questa lista (necessario per le animazioni).

Aggiorna Punteggio viene inviato quando il punteggio di un giocatore cambia. Contiene l'ID del giocatore e il suo nuovo punteggio.

Fine Partita Indica che la partita è terminata. Contiene i punteggi finali di tutti i giocatori.

Tocca a Te Questa azione viene inviata sempre come l'ultima azione della lista generata dalle regole, e indica che le regole sono in attesa della mossa successiva. Contiene l'ID del giocatore a cui tocca la prossima mossa, ed una lista di tutte le mosse valide che il giocatore può effettuare in quel momento.

6.1.3 L'interfaccia utente

L'interfaccia utente è scritta in QML per la versione Symbian, e in C++ utilizzando il motore grafico Cocos2DX per la versione iOS. QML ha permesso l'implementazione rapida di un'interfaccia fluida e semplice. All'avvio dell'applicazione l'utente visualizza il menù principale dell'applicazione, tramite il quale può selezionare il sottomenù relativo al gioco interessato (Briscola, Scopa e Tressette, di cui il primo disponibile da subito, gli ultimi due saranno disponibili in seguito ad eventuali aggiornamenti). Nel menu di secondo livello, l'utente può scegliere tra giocatore singolo (partita in locale

contro l'IA) o multigiocatore, quest'ultimo a sua volta suddiviso in partita 1 contro 1, partita 2 contro 2 o briscola chiamata (disponibile tramite aggiornamento). Inoltre sono presenti diverse opzioni relative all'audio e alla personalizzazione dell'aspetto grafico.

La schermata di gioco presenta un'interfaccia minimale e mirata a rendere l'esperienza utente il più simile possibile a una vera partita di briscola. Nella parte inferiore dello schermo vengono visualizzate le carte in mano al giocatore, che può riposizionarle come preferisce con un leggero effetto grafico. gli altri lati sono dedicati agli altri giocatori. Nell'angolo superiore destro è presente il punteggio attuale (se attivato nelle opzioni) e il pulsante di pausa, mentre a sinistra viene visualizzato il mazzo di carte ancora da pescare e la briscola. Il mazzo si assottiglia mano a mano che la partita procede.

Durante una partita multigiocatore, i giocatori possono inviare dei segni al compagno di squadra, oppure inviare delle frasi preimpostate a tutti i giocatori.

A fine partita compare una schermata di riepilogo con i punteggi complessivi dei giocatori, e un effetto sonoro in caso di vittoria. Da questa schermata è possibile tornare al menù principale oppure chiedere di iniziare un'altra partita contro gli stessi avversari. Se tutti i giocatori accettano, viene iniziata una nuova partita con gli stessi giocatori e le stesse squadre.

6.1.4 Intelligenza Artificiale[3]

Così come per le regole, anche la parte di intelligenza artificiale è stata scritta in C++ standard, utilizzando un'interfaccia comune a tutti i giochi.

L'intelligenza artificiale quindi è strutturata in una classe base astratta che definisce l'interfaccia tramite cui chiamarne i metodi, e una (o più) implementazioni per ogni gioco presente nell'applicazione. Una ulteriore classe fa a ponte tra l'intelligenza artificiale scritta in C++ standard e Mercurio.

Classi

AIBase L'interfaccia tramite cui chiamare l'intelligenza artificiale. definisce i seguenti metodi:

virtual void pesca(int carta) questo metodo viene chiamato quando si riceve l'evento azione *Carta Pescata* e l'ID del giocatore rappresentato dall'AI corrisponde al *playerID* presente nell'evento. Riceve come parametro l'ID della carta pescata.

[3] In questa sezione viene utilizzato anche il termine AI, abbreviazione dall'inglese Artificial Intelligence, per indicare l'intelligenza artificiale

virtual void avversarioPesca() questo metodo viene chiamato quando si riceve l'evento azione *Carta Pescata* e l'ID del giocatore rappresentato dall'AI non corrisponde al *playerID* presente nell'evento.

virtual void avversarioGioca(int carta) questo metodo viene chiamato quando si riceve l'evento azione *Carta Giocata* e l'ID del giocatore rappresentato dall'AI non corrisponde al *playerID* presente nell'evento. Riceve come parametro l'ID della carta giocata dall'avversario.

virtual int gioca() questo metodo viene chiamato quando si riceve l'evento azione *Tocca a Te* e l'ID del giocatore rappresentato dall'AI corrisponde al *playerID* presente nell'evento. a questo punto l'AI effettua i propri calcoli e decide la miglior mossa da effettuare, quindi ritorna l'ID della carta giocata. Da notare che l'AI deve conoscere le regole e lo stato del gioco (altrimenti non sarebbe in grado di elaborare una strategia valida), quindi non è necessario passare come parametro la lista delle mosse valide. tuttavia non è escluso che si possa modificare questo aspetto in futuro per semplificare l'implementazione di altre AI.

virtual void estrattaBriscola(int briscola) questo metodo è implementato solo nelle AI per il gioco Briscola, e viene chiamato quando viene "estratta la briscola", azione speciale presente in tale gioco.

AIInterface Questa classe riceve gli eventi di mercurio diretti ai giocatori, e si occupa di chiamare i metodi della AI in corrispondenza dei relativi eventi, nonché di generare gli eventi di mercurio corrispondenti ai comandi *Ready for Play* e *Gioca* rispettivamente a inizializzazione effettuata e al ritorno dalla funzione *gioca()*. Inoltre si occupa di istanziare la giusta classe AI a seconda dei parametri passati.

DumbAIBriscola Questa classe implementa l'AI per Briscola utilizzata durante il testing iniziale del componente delle regole, non effettua calcoli e si limita a giocare la prima carta che ha in mano.

MarioAIBriscola questa classe implementa l'AI per briscola rilasciata con l'applicazione. La strategia implementata è basata su una IA precedente scritta in C da Mario Fumagalli, che abbiamo migliorato e reimplementato in C++. Implementa quattro strategie diverse, che vengono scelte in funzione dello stato del gioco.

- La prima viene utilizzata durante la prima fase della partita, quando ci sono ancora più di due carte nel mazzo ancora da pescare, e l'IA è il primo giocatore a dover giocare in quella mano. le euristiche su cui si basa sono le seguenti:

- Se ha due briscole in mano, ne gioca una per evitare di rimanere con tre briscole e dover prendere senza fare punti la mano successiva.
  - Se di un seme sono usciti entrambi i carichi, gioca la figura più alta di quel seme se può. In questo modo ci sono dei punti sul tavolo e l'avversario di sicuro non ha un carico, quindi ottiene almeno 2 punti oppure obbliga l'avversario ad usare una briscola.
  - Altrimenti gioca la carta più bassa in ordine di preferenza semi che non sia un carico. L'ordine di preferenza semi si basa su un punteggio assegnato ald ogni seme in base alle carte giocate e ai carichi usciti. La strategia di base è quella di tentare di non prendere per poi strozzare le carte dell'avversario.
  - Se non trova carte da giocare secondo le regole sopra, vuol dire che ha tre carichi in mano. In questo caso cerca di fare la giocata più sicura che può. per prima cosa cerca di giocare un tre di un seme di cui sia già uscito l'asso, che non sia di briscola. se non ne ha, cerca di giocare un asso. altrimenti gioca un tre.
- La seconda viene utilizzata quando ci sono ancora carte da pescare e l'IA è il secondo giocatore di mano (si alterna alla prima strategia). La strategia un questo caso è molto semplice: per prima cosa calcola, per ogni carta che ha in mano, se prende con quella carta e quanti punti avrebbe la mano giocando. quindi si segna il punteggio massimo ottenibile, distinguendo i casi in cui giocherebbe una briscola o meno, e la perdita minima. quindi a seconda delle condizioni sceglie la carta da giocare:
  - Se non è possibile prendere, sceglie la carta che farebbe perdere meno punti possibile.
  - Se invece non è possibile non prendere, sceglie la carta che fa guadagnare più punti possibili e che non è una briscola. se ha solo briscole in mano, gioca la più bassa.
  - Nel caso generale, se può fare abbastanza punti senza giocare briscole prende, altrimenti, se può non prendere subendo una perdita non troppo elevata non prende. Se neanche questo è possibile, tenta di prendere senza utilizzare briscole. se fallisce ancora, prende con una briscola (se può) nel caso ne abbia due in mano o nel caso la perdita minima sia troppo elevata. altrimenti non prende. Tutte le soglie di punteggio per decidere se prendere o non prendere sono configurabili, e possono venire influenzate dal comportamento dell'avversario.
- La terza viene utilizzata quando rimangono esattamente due carte nel mazzo, e la briscola è un asso o un tre. In questo caso tenta di non

prendere cercando al comtenpo di perdere meno punti possibile. In questo modo nella mano successiva ha l'asso o il tre di briscola in mano.

- La quarta viene utilizzata durante le ultime tre mani, quando tutte le carte sono state pescate e quindi l'IA ha informazioni complete sullo stato del gioco. In questo caso l'IA calcola la miglior mossa possibile, ovvero la sequenza di mosse in grado di garantire maggior punteggio nelle ultime tre mani, nell'ipotesi che l'avversario faccia lo stesso.

6.1.5 Componente multiplayer

L'applicazione utilizza la libreria multiplayer da noi sviluppata per la gestione del flusso di dati che deve scambiarsi durante le partite multiplayer. Innanzitutto, ci si connette al master server, quindi chiede di iniziare una partita utilizzando la funzionalità di automatch. La libreria gestisce in automatico le operazioni necessarie, quindi invia un segnale quando i giocatori sono connessi. a questo punto inizia la partita. L'host inizializza la logica di gioco e ogni giocatore manda un messaggio all'host contenente il comando *Ready for Play* quando ha inizializzato l'interfaccia grafica.

A questo punto i client di gioco si scambiano i messaggi di gioco, che corrispondono agli eventi inviati a mercurio internamente all'applicazione, e che vengono serializzati e inoltrati in rete. I messaggi sono formati nel modo seguente: Il primo byte indica il tipo di messaggio, ovvero di che comando o azione si tratta, mentre i successivi contengono i parametri dell'evento. Ogni evento richiede al massimo una decina di byte per essere trasmesso, quindi la banda richiesta dall'applicazione è minima.

Al termine della partita, l'host invia i risultati al server, ovvero i nomi dei giocatori e chi ha vinto. Il server memorizza tali informazioni, che utilizzerà per costruire le classifiche.

Le classifiche sono suddivise tra chi ha giocato a più di 100 partite (*classifica pro*) e chi ne ha giocate di meno, e sono ordinate in base alla percentuale di partite vinte rispetto a quelle giocate.

Classi

La componente multiplayer comprende le seguenti classi:

RNMercurio Questa classe si occupa dell'inizializzazione della libreria multiplayer quando l'utente decide di iniziare una nuova partita, nonché della gestione della classifica.

RemoteLogic Questa classe viene istanziata sui client e prende il posto della logica. Riceve gli eventi di mercurio generati dall'interfaccia e li

invia all'host tramite la libreria multiplayer. Allo stesso tempo riceve i dati dalla rete e genera gli eventi di mercurio corrispondenti.

RemotePlayer Questa classe viene istanziata sull'host e agisce in modo speculare a *RemoteLogic*. Riceve gli eventi di mercurio generati dalla logica e li inoltra ai client, e trasforma i messaggi in arrivo dai client in eventi di mercurio.

Capitolo 7

Valutazione e Conclusioni

Una valutazione oggettiva dei risultati raggiunti è teoricamente possibile confrontando per esempio il carico sul server o il tempo di risposta dell'applicazione in funzione del numero di giocatori connessi, tuttavia purtroppo non è stato possibile in quanto i giocatori connessi contemporaneamente finora non hanno mai superato le 8 unità.

Forniamo quindi una valutazione empirica basata sull'utilizzo da parte nostra dell'applicazione Bar Briscola. I terminali utilizzati sono: un Nokia N8 con sistema operativo Nokia Belle, un Nokia C7-00 con sistema operativo Symbian Anna, un Nokia C7-00 con sistema operativo Nokia Belle e un Nokia X7 con sistema operativo Nokia Belle.

I terminali sono connessi a Internet tramite wifi o tramite 3G.

A meno di problemi relativi alla connessione alla rete (non sempre stabile) la libreria si è dimostrata sempre veloce e affidabile. La connessione al master server è avvenuta quasi sempre entro un paio di secondi. Lo scambio di pacchetti di gioco è quasi istantaneo se il *Nat Punchthrough* ha successo, mentre si può notare un ritardo di circa mezzo secondo se il *Nat Punchthrough* fallisce. Ciò è dovuto al fatto che i pacchetti devono passare attraverso il proxy, che attualmente è installato su una macchina localizzata in Inghilterra.

Il collo di bottiglia è la connessione tra giocatori, in cui le operazioni di *Nat Punchthrough* hanno durate fino all'ordine delle decine di secondi.

7.1 Conclusioni

L'obiettivo del nostro lavoro è stato raggiunto, attualmente è possibile giocare partite a briscola multigiocatore tra dispositivi *Symbian* e il nostro framework può essere integrato in applicazioni iOS e Android e permettere la comunicazione tra dispositivi diversi.

Il nostro framework al momento fornisce le funzionalità base per permettere partite multigiocatore, ma in futuro potrà essere esteso per supportare

funzionalità più avanzate, tra cui social gaming comprendente la gestione di liste di amici e la possibilità di invitarli a giocare, la registrazione di coppie o squadre di giocatori permanenti e l'introduzione di classifiche a squadre, la gestione e condivisione di *achievements* o l'integrazione con social network (Facebook, Google+, Twitter) per facilitare l'incontro tra giocatori, in quanto adesso è difficile trovare altre persone online nel momento in cui si vuole giocare, data la scarsa utenza disponibile.

Bibliography

[1] Pagina web del *Game Center* di Apple: http://www.apple.com/game-center/

[2] Pagina Web di GREE Platform: http://comeplay.gree.net/

[3] Pagina Web di GameSpy: http://www.poweredbygamespy.com/

[4] Network Address Translation su Wikipedia http://it.wikipedia.org/wiki/Network_address_translation

[5] Achille Pattavina, *Reti di Telecomunicazioni*, ISBN 88-386-6065-4

[6] James F. Kurose, Keith W. Ross, *Reti di Calcolatori e Internet*, ISBN 88-7192-225-5

[7] Sito Web di Unity: http://unity3d.com/

[8] Home page di RakNet: http://www.jenkinssoftware.com/

[9] Manuale utente di RakNet: http://www.jenkinssoftware.com/raknet/manual/index.html

[10] Piattaforma Symbian su Wikipedia: en.wikipedia.org/wiki/Symbian

[11] P.I.P.S. su Wikipedia: https://en.wikipedia.org/wiki/PIPS

[12] P.I.P.S. su Nokia Developer: http://www.developer.nokia.com/Community/Wiki/P.I.P.S

[13] Manuale Utente di P.I.P.S.: http://www.developer.nokia.com/Community/Wiki/A_Guide_To_P.I.P.S.

[14] Manuale Utente di Open C: http://www.developer.nokia.com/Community/Wiki/Open_C_library

[15] Home page di Qt: http://qt-project.org/

[16] Qt su Wikipedia: https://en.wikipedia.org/wiki/Qt_(framework)

[17] Internet Engineering Task Force RFC 1631, *The IP Network Address Translator (NAT)*, maggio 1994

[18] Internet Engineering Task Force RFC 1918, *Address Allocation for Private Internets*, febbraio 1996

[19] Internet Engineering Task Force RFC 3022, *Traditional IP Network Address Translator (Traditional NAT)*, gennaio 2001

[20] Internet Engineering Task Force RFC 4008, *Definitions of Managed Objects for Network Address Translators (NAT)*, marzo 2005

[21] Internet Engineering Task Force RFC 4966, *State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs)*, marzo 2008

[22] Internet Engineering Task Force RFC 5128, *Reasons to Move the Network Address Translator - Protocol Translator (NAT-PT) to Historic Status*, luglio 2007