Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Sviluppo di un sistema evoluto per la gestione di tornei di grandi dimensioni di giochi da tavolo o di carte.

Relatore:
Chiar.mo Prof.
Luciano Bononi

Presentata da:
Piero Nicolli

Sessione III
Anno Accademico 2013-2014

Indice

Elenco delle figure

Introduzione

L'idea nasce dall'esperienza personale nell'ambiente del gioco organizzato. Con l'espressione "gioco organizzato" si intende l'organizzazione di tornei di un determinato gioco, definiti da un regolamento specifico e seguiti da uno staff di una o più persone esperte del regolamento, il cui numero spesso dipende dal numero di giocatori. Questo staff si occupa solitamente della gestione logistica del torneo, oltre che di assicurarne lo svolgimento secondo le regole ufficiali. Vediamo come esempio il Grand Prix Milano 2014, torneo internazionale di un famoso gioco di carte collezionabili. Nella foto in Figura 1 più di 1000 giocatori sono pronti ad iniziare il torneo, mentre circa 60 persone dello staff lavorano alla gestione dell'evento. Tornei di questo tipo contano dai 500 ai 5000 giocatori, e gli staff variano dai 40 ai 200 membri.

Questi tornei si svolgono solitamente su più giorni, sottoponendo gli arbitri ad un notevole sforzo fisico. I grandi spazi rendono spesso necessario camminare a lungo per svolgere semplici compiti, come l'affissione di fogli con gli abbinamenti per le partite o con le classifiche parziali, oppure la consegna ad ogni coppia di giocatori dell'apposito foglio su cui dovranno indicare il risultato finale del proprio incontro e consegnarlo ad un membro particolare dello staff, lo scorekeeper, che ha la responsabilità di inserire tutti i risultati correttamente nell'apposito programma sul pc (che da qui in avanti chiameremo "Reporter"). In aggiunta, una giornata di torneo può durare dalle 10 alle 12 ore per un membro dello staff; queste ore vengono trascorse prevalentemente in piedi a camminare tra i tavoli da gioco, rendendo così di massima importanza la limitazione al minimo dello sforzo impiegato per svolgere i vari

Figura 1: Un esempio di torneo internazionale di alto livello. Immagine concessa da Alessandra Farina.

compiti logistici necessari per il torneo.

L'idea di sviluppare questo sistema nasce dal desiderio di ridurre al minimo indispensabile questo sforzo, poi si è evoluta in un sistema più complesso a cui si possono aggiungere strumenti utili in base all'ambiente specifico per cui viene implementato. Il grosso lavoro di squadra necessario per portare a termine questi tornei richiede di comunicare spesso con i membri dello staff, ma solitamente le comunicazioni sono molto brevi. Fare ciò richiede di camminare mediamente 50m solo per raggiungerlo e tornare alla propria mansione precedente[1]. La frequenza di questa operazione può essere stimata in circa 70 volte al giorno, cioè mediamente ogni 10 minuti. Azzerare gli spostamenti necessari per queste operazioni significherebbe arrivare quasi a

[1] 50m in un torneo di circa 1000 persone come quello in foto. Il numero non è molto più alto però per tornei di dimensioni maggiori, dove solitamente lo staff si divide in più team e si massimizza la coordinazione tra questi.

dimezzare la distanza percorsa giornalmente da ciascuno. A volte purtroppo questi spostamenti si rendono comunque necessari, quando dobbiamo portare oggetti fisici come fogli di carta, però possiamo comunque ridurre di almeno 2km al giorno la distanza percorsa da ogni arbitro.

Un fattore molto importante in questi tornei è anche il tempo impiegato per svolgere i diversi compiti. Dal momento che solitamente una giornata inizia intorno alle 9 e termina intorno alle 23, si cerca di risparmiare ogni minuto possibile. Un guadagno di tempo c'è con il sistema che presenteremo ora, anche se non è particolarmente rilevante: si può considerare infatti che il tempo impiegato a percorrere 50m sia di poco superiore al tempo impiegato a compiere due azioni di tocco e scrivere un messaggio di 10 parole con il proprio smartphone. In ogni caso il risparmio può aumentare con l'aggiunta di strumenti specifici del gioco per cui si andrà ad implementare il sistema.

Passiamo ora a descrivere invece lo scorekeeper e il modo in cui questo sistema aiuta anche la sua figura. Come detto in precedenza, è un membro particolare dello staff. A differenza degli altri, è seduto davanti al pc quasi tutto il giorno. Il suo lavoro richiede grande concentrazione e sforzo mentale: deve inserire tutti i risultati delle partite nel Reporter, che calcola classifica e abbinamenti degli incontri successivi. È un lavoro di grande responsabilità, dal momento che un risultato sbagliato può portare a perdite di tempo negli incontri successivi.
Consideriamo un torneo con 2000 partecipanti, che si vede ormai di frequente; i numeri di questi eventi stanno aumentando drasticamente negli ultimi anni, i partecipanti in media sono raddoppiati negli ultimi 5 anni. Lo scorekeeper riceve 1000 risultati, da inserire nel pc nell'arco di un turno di incontri, che dura circa un'ora. Ciò significherebbe inserire circa 16 risultati al minuto. Purtroppo però questo non è ciò che realmente accade. Non meno del 50% dei risultati, infatti, arriva allo scorekeeper negli ultimi 20 minuti a disposizione, portando così il numero di incontri da inserire a 25 al minuto, nel caso migliore. È evidente che egli ha bisogno dei minuti iniziali di un turno di gioco, dove non ha questo lavoro da svolgere, per riposare la mente ed

essere pronto ad un nuovo burst di inserimenti. Solitamente, invece, deve passare questo tempo ad inserire altre informazioni di secondaria importanza, ma che vanno comunque registrate nel report dell'evento: le penalità. Ogni regolamento prevede logicamente diverse penalità per chi non lo rispetta, che dipendono dal tipo di infrazione commessa dal giocatore. Mentre i risultati sono una grande responsabilità e in alcuni ambienti di gioco organizzato solamente lo scorekeeper può avere questa responsabilità[2], sulle penalità possiamo intervenire per migliorare le condizioni di questa figura. Parlando con alcuni di essi, infatti, hanno confermato che sarebbe ottimo se il nostro sistema permettesse agli arbitri dello staff, quando assegnano le penalità, di inserirle direttamente nel Reporter.

Vediamo ora un altro strumento che, sebbene sia di minore importanza, è volto a semplificare il lavoro dello staff a questi eventi: un timer che indica il tempo rimanente nel turno di gioco in corso, che verrà avviato sugli smartphone dello staff con un semplice click da parte dello scorekeeper. Vi sarà inoltre la possibilità di tenere più timer personali con un'etichetta per distinguerli. Spesso capita di essere impegnati in qualche mansione nel momento in cui viene dato il via al tempo per il turno di gioco corrente; grazie a questo strumento non abbiamo bisogno di un cronometro extra e non dobbiamo stare sempre attenti a non perderci l'annuncio di inizio turno. In molti di questi giochi, inoltre, a volte viene concesso tempo extra ad alcuni incontri, se sono stati ritardati per motivi logistici o per indagini varie: ecco che possiamo tenere un timer aggiuntivo tutto nella stessa app sul nostro smartphone.

Per concludere, è importante sottolineare che si parla di tornei internazionali, ciò significa che lo staff è composto da persone provenienti da ogni parte del mondo che difficilmente avranno accesso ad internet dal proprio smartphone sul luogo del torneo. Questo esclude ogni possibilità di utilizzare un sistema di comunicazione tra quelli già esistenti, che si appoggiano tutti a

[2]In questi casi sono persone selezionate ad hoc, e seguono un addestramento appositamente studiato.

server esterni, e rende contemporaneamente accettabile l'idea di connettere gli smartphone ad una rete locale senza accesso ad internet.

Capitolo 1

Progettazione

Vedremo innanzitutto l'architettura del sistema, con una descrizione degli strumenti necessari e delle scelte effettuate per quanto riguarda la rete e il server. In seguito vedremo la struttura delle applicazioni sviluppate e il modo in cui faremo comunicare i dispositivi.

1.1 Architettura del sistema

Abbiamo visto che gli elementi da tenere in considerazione per la progettazione del sistema sono:

- I membri dello staff e i loro smartphone (non connessi ad internet!)
- Lo scorekeeper e il suo pc
- Reporter software già esistente e in molti casi non sostituibile, se fornito dall'azienda produttrice del gioco
- Ampi spazi non suddivisi in cui si svolgono i tornei

La prima cosa che notiamo dunque è che dobbiamo creare una rete locale a cui connettere tutti i dispositivi in questione (smartphones e pc). Un moderno access point, ben posizionato, è in grado di coprire una sala delle dimensioni di quella vista precedentemente in Figura 1; per sale molto grandi è comunque

possibile creare una rete unica con più di un access point. La banda richiesta dalle informazioni scambiate nel nostro sistema è minima e le connessioni sono brevi e sparse, quindi questa rete sarebbe in grado di sostenere più di 100 utenti senza problemi. Supponiamo dunque di avere gli smartphones ed il pc connessi alla stessa rete locale, ci servono ora un'applicazione mobile e una per il pc dello scorekeeper, in grado di creare contemporaneamente un canale di comunicazione rapido, in stile chat, e capace di scambiare informazioni sull'evento, come ad esempio le penalità, senza richiedere azioni aggiuntive da parte degli utenti. Individuiamo dunque per prima cosa il protocollo da utilizzare per la comunicazione.

1.2 XMPP

XMPP (Extensible Messaging and Presence Protocol)[3] è un protocollo di comunicazione in tempo reale che supporta la messaggistica istantanea tra due utenti o gruppi di persone, chiamate vocali e video e in generale lo scambio di dati XML. Lo scambio di informazioni avviene attraverso elementi detti "stanzas", che non sono altro che elementi XML che possiedono diversi attributi a seconda della funzione che ricoprono, fatta eccezione per 5 attributi base che si trovano in ogni stanza: `from`, `to`, `id`, `type` e `xml:lang`. Le stanzas possono essere di tre tipi diversi: `<message/>`, `<presence/>` e `<iq/>`. Il primo tipo rappresenta un meccanismo di messaggistica in stile push, in cui un'entità invia informazioni ad un'altra entità, come ad esempio le email o i servizi di messaggistica istantanea. Il secondo tipo rappresenta notifiche di presenza o disponibilità e può funzionare in stile broadcast normale, oppure in stile subscribe, in cui ognuno ha una lista di entità a cui è "iscritto"; solo chi è iscritto ad una certa entità riceverà le notifiche di presenza da parte di essa. Le iscrizioni avvengono con un sistema che richiede uno scambio di autorizzazioni da parte di entrambe le entità; i client però possono essere solitamente configurati per accettare automaticamente le richieste di iscrizione. Il terzo tipo di stanza, infine, rappresenta un sistema di

comunicazione in stile richiesta/risposta, con richieste `get` o `set` e risposte `result` o `error`. Un esempio di stanza `<message/>` è il seguente:

```
<message from="romeo@montague.net/orchard"
    to="juliet@capulet.com/balcony"
    type="chat"
    xml:lang="en">
    <body>Neither, fair saint, if either thee dislike.</body>
</message>
```

Gli utenti sono identificati dal loro "jid" (Jabber ID), che è composto da tre valori: username, domain e resource. Nell'esempio, il jid di Romeo è `romeo@montague.net/orchard`, quindi lo username è "romeo", il domain è "montague.net" e la resource è "orchard". Username e domain sono di ovvia comprensione, nel nostro caso il domain sarà quello scelto come *Server Name* durante la configurazione del server, che vedremo più avanti in questo capitolo. La resource serve a identificare il client da cui è connesso l'utente. Ciò significa che un utente può essere connesso da più client contemporaneamente, se per ogni client usa una resource differente. Solitamente i client permettono di impostare una resource personalizzata, ma non è importante metterla: in caso non sia impostata, ne verrà assegnata una casuale dal server in fase di login. Se un utente si connette con la stessa resource da due client, il primo client verrà disconnesso e riceverà un messaggio di errore. Da notare come `<body/>` sia un elemento figlio di `<message/>` e non un attributo. Questo è importante, perché un messaggio potrebbe avere più elementi `<body/>`, a patto che ciascuno di questi elementi abbia un attributo `xml:lang` differente. Un altro elemento figlio possibile, che ci interessa ai fini del progetto, è `<subject/>`. Possiamo impostare il valore di questo figlio per indicare il soggetto del messaggio inviato. Anche `<subject/>` può essere presente più volte in un messaggio, con la stessa restrizione sulla lingua dell'elemento `<body/>`.

Sulla base di queste tre tipologie di comunicazione, lo sviluppatore può costruire le stanzas a suo piacimento, organizzando i dati all'interno nella maniera che risulta più utile e senza compiere grande sforzo, dal momento

che sono elementi XML di facile comprensione. Per questo progetto, abbiamo bisogno di due tipi di messaggio: i messaggi di comunicazione tra gli utenti e i messaggi che invece useranno i client per comunicare informazioni di servizio tra di essi. Entrambi i tipo di messaggi potranno essere unicast o broadcast. Per i messaggi di chat unicast tra gli utenti imposteremo l'attributo `type` dei messaggi al valore *chat*, come nell'esempio precedente. Per i messaggi di servizio unicast useremo messaggi di tipo *normal* e ne imposteremo il subject ad un valore appropriato, per esempio *penalty* nel caso di un arbitro che invia una penalità al Reporter. Per rendere il messaggio interpretabile utilizziamo una formattazione particolare dell'elemento `<body/>` definita appositamente. Vediamo un esempio:

```
<message from="piero@eventdomain"
    to="scorekeeper@eventdomain"
    type="normal"
    xml:lang="en">
    <subject>penalty</subject>
    <body>NEW:::1:::45:::infraction:::penalty:::descr</body>
</message>
```

La stringa contenuta nel body può essere facilmente spezzata in corrispondenza della stringa ":::" e interpretata dal client dello scorekeeper. Essa conterrà i dati necessari all'inserimento della penalità: turno in cui è avvenuta l'infrazione, numero del giocatore che l'ha commessa, nome dell'infrazione, penalità assegnata e descrizione dell'accaduto.

Per i messaggi broadcast, invece, useremo il tipo *groupchat* e ci appoggeremo alle stanze di chat multiutente definite nell'estensione XEP-0045 del protocollo XMPP. In questo caso, creeremo una stanza di comunicazione sul server per ogni tipo di messaggio che vogliamo inviare. Nel caso della comunicazione fra utenti, avremo una stanza dal nome *broadcast* in cui sarà possibile per lo scorekeeper inviare un messaggio a tutto lo staff. Un'estensione futura delle applicazioni mobili potrebbe dare ad un selezionato gruppo di utenti l'opzione di mandare annunci in broadcast. Nell'esempio successivo vediamo un possibile annuncio allo staff.

```
<message to="broadcast@conference.eventdomain"
    type="groupchat"
    xml:lang="en"
    from="scorekeeper@eventdomain">
    <body>Ritrovo sotto il banner rosso alle 15:00.</body>
</message>
```

Nel caso della comunicazione tra i client indirizzeremo il messaggio alla stanza *broadcast-control* e useremo una stringa simile a quella usata nel caso unicast. Ad esempio, per avviare i timer dei client mobile, invieremo questo messaggio:

```
<message to="broadcast-control@conference.eventdomain"
    type="groupchat"
    xml:lang="en"
    from="scorekeeper@eventdomain">
    <body>TIMERSTART:::5:::50:::1425049860</body>
</message>
```

Le informazioni inviate sono, nell'ordine: turno di gioco in corso, durata del timer in minuti, Unix timestamp dell'invio del messaggio[1]. Il turno serve come semplice informazione per aggiornare l'interfaccia delle applicazioni sugli smartphone, il timestamp viene inviato per un calcolo più preciso della durata del timer, secondo la formula

$$timer = d - (t_{now} - t_{rec})$$

dove d è la durata del timer indicata nel messaggio, t_{now} è lo Unix timestamp corrente, mentre t_{rec} è il timestamp ricevuto. Così facendo calcoliamo con buona approssimazione i secondi trascorsi da quando il Reporter ha avviato il timer, e impostiamo il timer dell'arbitro di conseguenza.

Per quanto riguarda la nostra architettura, costruiremo dunque due client XMPP, uno per smartphone e uno per pc. Come server useremo Openfire[4], la cui installazione è facilissima e la configurazione è molto veloce e comprensibile da chiunque. Possiamo installarlo sullo stesso pc dello scorekeeper, ma possiamo anche decidere di installarlo su un altro pc connesso alla stessa rete, se disponibile, per ridurre il carico di lavoro sul pc con il Reporter.

[1] Secondi trascorsi dalla mezzanotte del 1/1/1970 (data nota anche come *epoch*).

Configurazione di Openfire

La configurazione necessaria a far funzionare il sistema è molto rapida e richiede pochi passaggi: installazione, configurazione di base, creazione degli utenti e delle stanze per la chat di gruppo.

Il target del sistema è una piattaforma Windows, solitamente usata per i Reporter già esistenti. L'installer Windows di Openfire offre una procedura guidata molto semplice ed è sufficiente seguirla fino in fondo per completare correttamente l'installazione. Il sito ufficiale fornisce ulteriori istruzioni per avviarlo come servizio Windows, in caso di necessità[2]. Un comodo launcher,

Figura 1.1: Launcher di Openfire per Windows.

che viene installato insieme al server, permette di avviare o fermare il server con un click (Figura 1.1) e include un bottone che porta direttamente alla pagina web di configurazione.

[2]http://www.igniterealtime.org/builds/openfire/docs/latest/documentation/install-guide.html#windows

Le operazioni che è necessario compiere in questa pagina sono molto semplici, anche se in questa prima versione del sistema richiedono tempo. Infatti dovremo inserire manualmente tutti gli utenti (cioè i membri dello staff). Openfire permette la creazione automatica di account da parte degli utenti, ed è sicuramente una delle prime feature da sviluppare in futuro.

Per sicurezza disabilitiamo l'accesso da parte di utenti non registrati: Nella tab *Server*, nel sottomenù *Server Settings*, troviamo l'opzione *Anonymous Login*. Impostiamolo su *Disabled* come in Figura 1.2.

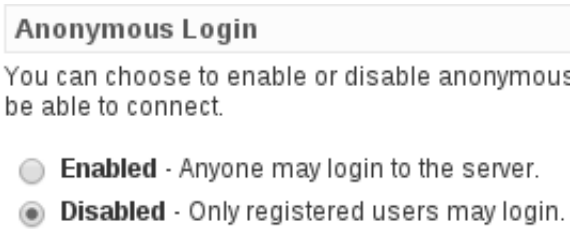

Figura 1.2: Proibiamo l'accesso anonimo; vogliamo solo membri dello staff sul server.

La creazione degli utenti è disponibile nella tab *Users/Groups*, selezionando dal menù laterale la voce *Create New User*. Per creare invece le stanze per le chat di gruppo, selezionare la tab *Group Chat* e dal menù laterale *Create New Room*. In questa schermata, oltre ad inserire il nome della stanza, è importante impostare il numero massimo di utenti ammessi contemporaneamente all'interno di essa, dato che il valore di default 30 non è sempre sufficiente nel nostro caso. In questo momento le uniche stanze che ci servono sono quelle per la comunicazione tra le applicazioni e quella degli annunci.

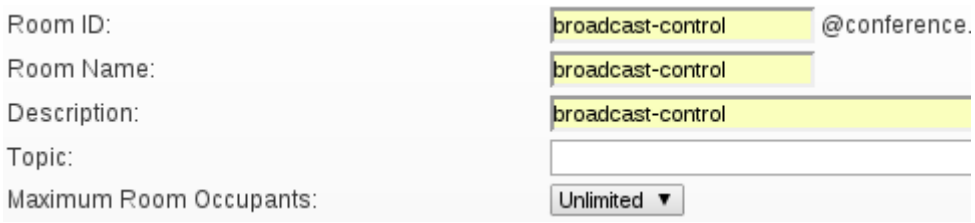

Figura 1.3: Esempio di creazione di una stanza.

Nella schermata di creazione della stanza, nei campi *RoomID*, *Room Name* e *Description*, scriviamo *broadcast-control* per creare la prima stanza, *broadcast* per creare la seconda. Vediamo un esempio in Figura 1.3. In entrambi i casi, togliamo il limite di utenti nella stanza, dal momento che vogliamo che tutto lo staff vi entri.

Il server è ora configurato a dovere ed è pronto per l'uso. È importante segnarsi il *Server Name* impostato (lo possiamo leggere nella home page dell'interfaccia web) e comunicarlo allo staff, affinché si possa connettere.

1.3 Descrizione del sistema

Per supportare lo sviluppo futuro di un sistema specifico e, contemporaneamente, rendere possibile un'estensione di questo sistema per qualsiasi gioco da tavolo di cui si possano organizzare tornei ufficiali, vediamo la definizione dei casi d'uso del sistema, il diagramma degli stati e quello delle classi. Consideriamo per questo compito che anche il Reporter faccia parte del sistema, anche se, come detto in precedenza, per alcuni giochi è necessario utilizzarne uno già esistente.

Il diagramma dei casi d'uso è molto semplice, dato che la maggior parte delle operazioni sono svolte dallo scorekeeper sul Reporter. Come vediamo in Figura 1.4, abbiamo suddiviso lo staff in due attori principali: scorekeeper e arbitro, date le diverse mansioni svolte dalle due figure. Le uniche operazioni svolte dagli arbitri direttamente sul sistema sono quelle sul proprio smartphone, quindi l'inserimento di penalità e la comunicazione con altri membri dello staff. A fine capitolo è riportata la specifica dei casi d'uso.

Nel diagramma degli stati in Figura 1.5 è mostrato il succedersi degli eventi nel corso del torneo. La maggior parte di esso si svolge nello stato di *Results entry*, che è lo stato in cui lo scorekeeper sarà in attesa mentre i giocatori disputano i propri incontri. La fase di creazione del torneo e registrazione dei giocatori non si prolunga per più di un'ora e mezza, solitamente. Si nota immediatamente come le operazioni da compiere sul sistema

si svolgano ciclicamente per l'intera durata dell'evento:

1. Creazione degli abbinamenti ("pairings") dei giocatori per il prossimo turno di incontri;

2. Inserimenti dei risultati quando gli incontri terminano. Una volta terminati tutti gli incontri, ripetere dal punto 1 finché ci sono altri turni da giocare.

Il diagramma delle classi in Figura 1.6 descrive la struttura delle informazioni all'interno del sistema. Si nota come sia definito un modello di torneo che contiene tutte le informazioni sui partecipanti e sullo staff. Questo modello viene poi utilizzato dal Reporter sul PC per gestire l'evento. Possiamo dunque estendere le funzionalità del programma, creando gli strumenti necessari a costruire un canale di interazione con il torneo da parte degli arbitri, che possono così inserire le penalità e ricevere il timer del turno in corso. Si può facilmente estendere questo sistema, per esempio per permettere agli arbitri di inserire i risultati degli incontri, qualora fosse permesso per il gioco che ci interessa.

1.4 Smartphone client

Si vuole dunque progettare un client per smartphone che contenga gli strumenti descritti sopra e che disponga al suo interno di una chat per comunicare con il resto dello staff. Gli obiettivi primari sono la velocità e la semplicità con cui gli arbitri possono utilizzare questi strumenti e passare comodamente da uno all'altro. Raccogliamo dunque le linee guida più importanti per massimizzare l'usabilità dell'applicazione:

- Il timer del turno di gioco dovrebbe essere facilmente visualizzabile in ogni momento, dato che molti task dell'arbitro durante il torneo dipendono da quanto tempo manca al termine degli incontri.

- L'applicazione deve permettere di passare dalla chat ai timer alle penalità senza perdere lo stato del lavoro precedente (e.g. se stavo aggiungendo un timer personalizzato e voglio leggere un messaggio).

- Passare da uno strumento ad un altro deve richiedere un gesto o al massimo due, è importante che l'utilizzo dell'applicazione non rubi tempo alle normali operazioni.

Dati questi punti, vediamo com'è stata progettata l'applicazione, guardando prima il modello di funzionamento, poi l'interfaccia.

Gli elementi necessari per il modello sono un database, in cui tenere tutti i dati sulle conversazioni e le informazioni inviate al Reporter, e un servizio in background, che si occupi dello scambio di dati con il server anche quando l'applicazione è chiusa. Per l'interfaccia, invece, ci serve un elemento che permetta di vedere il timer del turno in corso da qualsiasi schermata stiamo visualizzando, un elemento che permetta di accedere velocemente alle diverse funzionalità, senza distruggere le schermate precedenti, per non perdere eventuali altre operazioni non portate a termine. Nel capitolo successivo vedremo nel dettaglio l'implementazione delle singole parti.

1.5 PC client

Il client per lo scorekeeper ha necessità differenti, dovute ai diversi task che egli deve compiere. È importante fare una premessa: per questa dimostrazione del sistema è stato sviluppato anche un prototipo di Reporter, anche se semplificato, così da poter dimostrare lo svolgimento del torneo per intero. Tornando alle linee guida per lo sviluppo del client, ci possiamo trovare in due situazioni: potremmo dover sviluppare un client con Reporter integrato, come quello che vedremo in questo lavoro, oppure un client standalone, che deve comunicare con un Reporter già esistente e che si deve quindi adeguare alle possibilità di interazione che questo mette a disposizione. Alcuni Reporter potrebbero ammettere lo sviluppo di plugin, altri limitarsi a poter leggere

informazioni da file di testo appositamente formattati. Per esempio, quello del gioco di cui mi occupo personalmente permette di scrivere funzionalità aggiuntive che leggono dati da file in formato CSV.

In entrambi i casi le linee guida fondamentali individuate sono:

- La chat non può occupare molto spazio sullo schermo, non deve coprire il Reporter, dal momento che lo scorekeeper deve poter passare velocemente da leggere una breve comunicazione a inserire risultati o compiere altre azioni.

- Il client deve essere in grado di ricevere e gestire automaticamente le informazioni inviate dagli smartphones, come le penalità.

- Lo scorekeeper deve poter avviare il timer su tutti i dispositivi mobili contemporaneamente. Non è fondamentale che i dispositivi siano perfettamente sincronizzati, qualche secondo di differenza è tollerato.

In aggiunta, dal momento che con questo progetto vogliamo presentare un sistema completo, è necessario sviluppare un Reporter funzionante, che sia in grado quindi di svolgere tutte le operazioni descritte nel diagramma dei casi d'uso visto sopra, in Figura 1.4.

Anche in questo caso ci serviranno un database per i dati del torneo e per la chat e un servizio in background che si occupi di scambiare i dati con il server. Per quanto riguarda l'interfaccia, creeremo un'unica schermata che comprenda il Reporter e la chat, affiancati in modo da rispettare gli obiettivi prefissati. La chat dovrà permettere il normale scambio di messaggi con gli altri membri dello staff; in aggiunta daremo la possibilità allo scorekeeper di inviare messaggi particolari: gli annunci. Questi non sono altro che broadcast che arrivano a tutto lo staff, inviati in caso di comunicazioni di servizio importanti da parte dello scorekeeper stesso, oppure di quel membro dello staff che si occupa di coordinare tutte le operazioni, che in alcuni ambienti prende il nome di "Head Judge".

Use case specifications

Caso d'uso: Create tournament
Attori: Scorekeeper
Precondizioni: Lo scorekeeper conosce i dati del torneo e dello staff (judges e scorekeeper)
Flusso principale:

1. Lo scorekeeper avvia la creazione del torneo nel sistema.
2. Lo scorekeeper inserisce i dati del torneo (nome, eventuale ID, metodologia di abbinamento giocatori, dati secondari a seconda del tipo di torneo)
3. Lo scorekeeper inserisce i dati dello staff (nome, cognome, ID)
4. Lo scorekeeper conferma la creazione dell'evento. Il caso d'uso termina.

Conseguenze: Lo scorekeeper ha creato il torneo nel sistema.

Caso d'uso: Registration
Attori: Scorekeeper
Precondizioni: Lo scorekeeper ha creato il torneo nel sistema.
Flusso principale:

1. Il primo giocatore in coda riferisce allo scorekeeper i dati richiesti (solitamente nome, cognome e un qualche ID).
2. Lo scorekeeper inserisce i dati del giocatore.
3. Se il torneo è a squadre, si passa al flusso alternativo 1.
4. Se ci sono altri giocatori in coda, ripeti dal passo 1.
5. Lo scorekeeper dà inizio al torneo. Il caso d'uso termina.

Flusso alternativo 1:

1. Il compagno di squadra del giocatore riferisce allo scorekeeper i propri dati.

2. Lo scorekeeper inserisce i dati del giocatore e crea la squadra.

3. Se le squadre del torneo devono essere composte da più di due giocatori, ripeti dal passo 1.

4. Ritorna al flusso principale.

Conseguenze: Tutti i giocatori sono iscritti al torneo.

Caso d'uso: Start tournament
Attori: Scorekeeper
Precondizioni: Tutti i giocatori sono iscritti al torneo.
Flusso principale:

1. Lo scorekeeper conclude la fase di registrazione dei giocatori e dà inizio al torneo. Il caso d'uso termina.

Conseguenze: Il torneo è iniziato. La classifica generale è impostata ad un valore iniziale di default.

Caso d'uso: Pairings
Attori: Scorekeeper, Judge
Precondizioni: Tutti i giocatori sono iscritti al torneo. Il torneo è appena iniziato, oppure sono stati inseriti tutti i risultati delle partite precedenti.
Flusso principale:

1. Lo scorekeeper avvia la creazione degli abbinamenti dei giocatori per il successivo turno di incontri.

2. Il sistema crea gli abbinamenti automaticamente a partire dalla classifica generale e dal metodo di abbinamento impostato in fase di creazione del torneo.

3. Lo scorekeeper consegna gli abbinamenti agli arbitri.

4. Gli arbitri espongono gli abbinamenti ai giocatori.

Conseguenze: I giocatori si accomodano al proprio tavolo e iniziano a giocare il proprio incontro. Inizia un turno di gioco.

Caso d'uso: Results entry
Attori: Judge, Scorekeeper
Precondizioni: Il giocatore ha terminato il proprio incontro in un determinato turno di gioco. Il giocatore può riferire il proprio risultato da un arbitro, oppure direttamente allo scorekeeper.
Flusso principale:

1. Se il giocatore riferisce il proprio risultato direttamente allo scorekeeper, salta al punto 3.

2. L'arbitro riferisce il risultato del giocatore allo scorekeeper.

3. Lo scorekeeper inserisce il risultato del giocatore nel sistema.

4. Se il sistema non è in attesa di altri risultati, il turno di gioco si conclude e il sistema calcola la nuova classifica.

5. Il caso d'uso termina.

Conseguenze: Dopo aver inserito tutti i risultati, il turno di gioco termina.

Caso d'uso: Penalties entry
Attori: Judge, Scorekeeper
Precondizioni: Un giocatore ha commesso un'infrazione. L'arbitro, grazie all'applicazione sul suo smartphone, può inserirla direttamente nel sistema.
Flusso principale:

1. L'arbitro inserisce direttamente l'infrazione nel sistema. Il caso d'uso termina.

Conseguenze: La penalità è inserita nel sistema, per tenere traccia del comportamento dei giocatori.

Caso d'uso: Standings
Attori: Scorekeeper, Judge
Precondizioni: Non vi è alcun turno di gioco in corso.
Flusso principale:

1. Lo scorekeeper consegna la classifica agli arbitri.
2. Gli arbitri espongono la classifica ai giocatori.

Conseguenze: I giocatori prendono visione della classifica.

Caso d'uso: Team communication
Attori: Judge, Scorekeeper
Precondizioni: Un membro dello staff ha necessità di comunicare con uno o più membri dello staff.
Flusso principale:

1. Se è un arbitro a dover comunicare, invia un messaggio tramite l'app mobile.
2. Se è lo scorekeeper a dover comunicare, invia un messaggio tramite il pc.
3. Se il messaggio è diretto ad un arbitro, lo riceve e legge tramite l'app mobile.
4. Se il messaggio è diretta allo scorekeeper, lo riceve e legge sul pc.

Conseguenze: Lo scambio di messaggi è avvenuto correttamente.

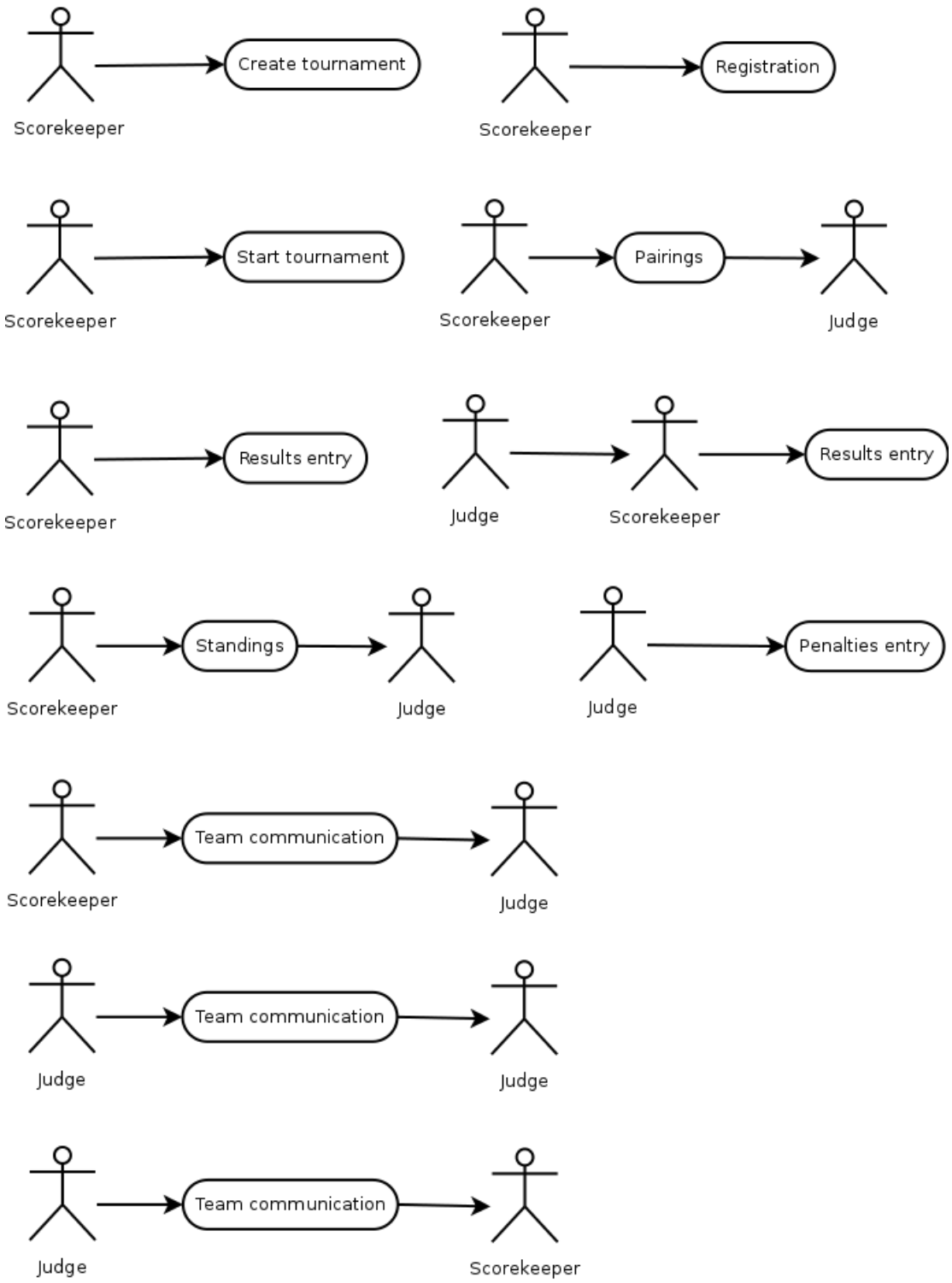

Figura 1.4: Diagramma dei casi d'uso.

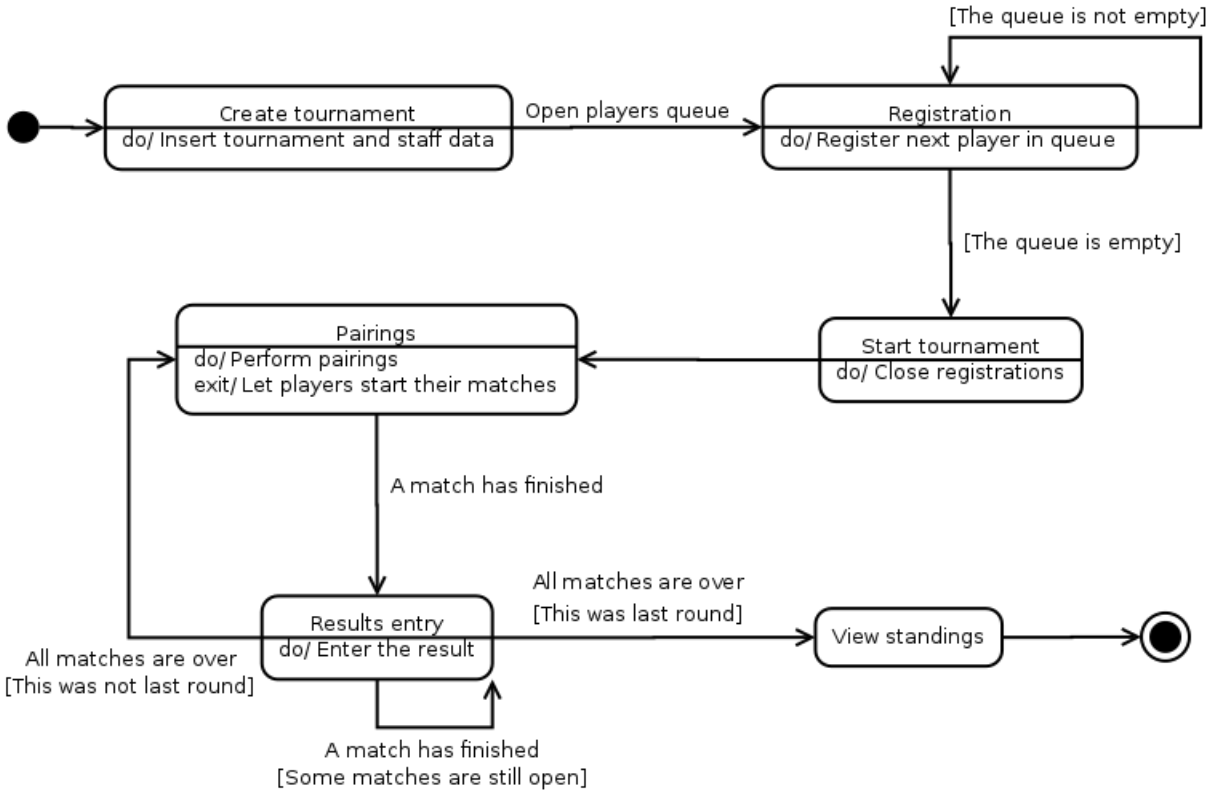

Figura 1.5: Diagramma degli stati.

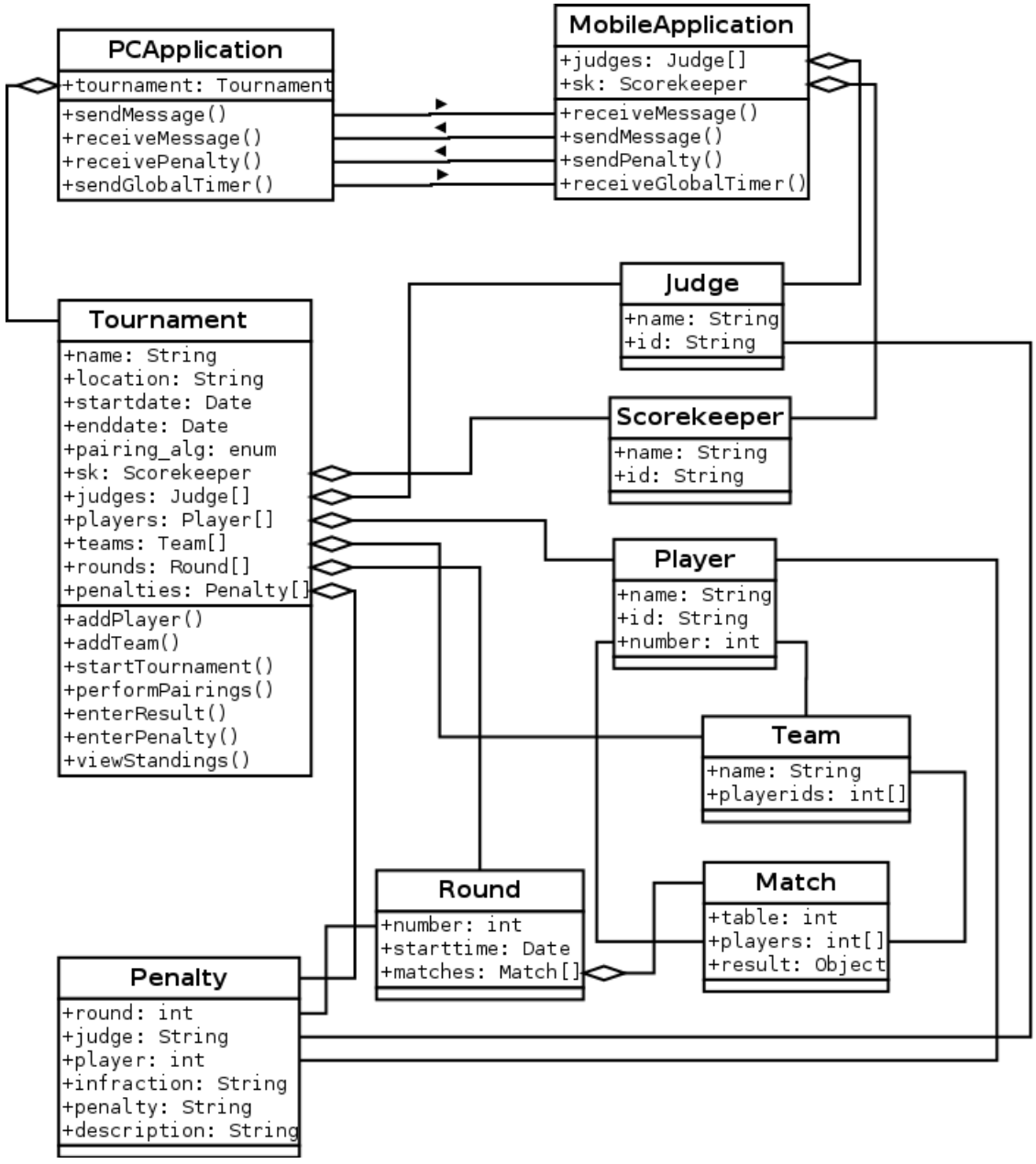

Figura 1.6: Diagramma delle classi.

Capitolo 2

Sviluppo del sistema

Dopo aver chiarito quali sono gli obiettivi del sistema e aver definito come si intende sviluppare ogni sua componente, vediamo l'implementazione finale dei due client.

2.1 Il client Android

Ai fini di questo lavoro, abbiamo sviluppato il client mobile per piattaforme Android. Per avere un interesse reale nel settore del gioco organizzato, si renderà sicuramente necessario svilupparne anche la versione per altre piattaforme, iOS in primis e possibilmente Windows Phone in seguito.

Il database SQLite

SQLite[5] è una libreria software che implementa un database engine SQL molto leggero, che non ha un processo server e si limita a salvare tutto il database in un unico file. Alcuni altri vantaggi di SQLite sono: supporto quasi totale di SQL92, indipendenza dalla piattaforma, non ha dipendenze esterne. Sono invece limiti di questa libreria: la gestione dei permessi di accesso e della concorrenza è lasciata al software che lo utilizza, non esiste una cache per le query, il comando `ALTER TABLE` permette solamente di rinominare la tabella o di aggiungere colonne in coda, diversi costrutti, sottoquery e trig-

ger non sono supportati. Molti di questi limiti, però, non sono assolutamente un problema per lo sviluppo di semplici applicazioni embedded come quelle per dispositivi mobili, ma anzi la leggerezza e velocità di SQLite lo rendono molto indicato sia per lo sviluppo per smartphone che per il salvataggio di dati semplici, come per esempio bookmark e cronologia di navigazione dei browser.

Android offre classi che rendono estremamente semplice la creazione e l'utilizzo di database Sqlite. Avremo bisogno di definire quattro tabelle: una per gli annunci, una per le penalità e due per i messaggi, di cui una per i messaggi veri e propri e l'altra di ausilio per calcolare i messaggi da inserire nella schermata che racchiude le conversazioni (la vedremo tra poco). Oltre a queste, aggiungeremo una classe per salvare i dati di accesso degli utenti, per offrire l'autocompletamento dei dati nella schermata di login (anche questa la vediamo tra poco).

Per mantenere una migliore organizzazione del codice, è stata creata una classe `DBHelper`, che eredita dalla classe Android `SQLiteOpenHelper`. `DBHelper` si occupa di gestire l'accesso al database, ma le operazioni sulle varie tabelle sono definite nelle classi ausiliarie come `MessagesTableHelper`, che contiene definizioni di metodi statici dedicati ad operare sulla tabella dei messaggi. Per ogni tabella del database esiste una classe ausiliaria che se ne occupa. Così facendo si contiene la lunghezza della classe `DBHelper` entro valori accettabili per una buona comprensione del codice. Essa infatti non fa altro che accettare richieste di query, delegarle ai metodi statici della classi ausiliarie, prenderne gli eventuali valori di ritorno e trasformarli per essere leggibili al meglio dall'applicazione. Questa struttura permette anche di avere un livello di astrazione in più tra applicazione e implementazione del database, che va a rendere più semplice la comunicazione e lo sviluppo degli stessi. Vediamo come esempio la classe `MessagesTableHelper`. In essa troviamo delle costanti che definiscono i nomi della tabella e dei suoi campi, più la stringa di creazione della stessa e un metodo statico che se ne occupa.

```
public class MessagesTableHelper {

    public static final String TABLE_NAME = "messages";
    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_EVENT = "event_id";
    public static final String COLUMN_USER = "username";
    public static final String COLUMN_SENDER = "sender";
    public static final String COLUMN_DEST = "dest";
    public static final String COLUMN_TEXT = "text";
    public static final String COLUMN_TIME = "time";

    private static final String TABLE_CREATE = "create table " +
            TABLE_NAME + "(" +
            COLUMN_ID + " integer primary key autoincrement, " +
            COLUMN_EVENT + " text not null, " +
            COLUMN_USER + " text not null, " +
            COLUMN_SENDER + " text not null, " +
            COLUMN_DEST + " text not null, " +
            COLUMN_TEXT + " text not null, " +
            COLUMN_TIME + " integer not null);";

    public static void onCreate(SQLiteDatabase db) {
        db.execSQL(TABLE_CREATE);
    }

    // Altri metodi della classe per le normali operazioni
    // di ricerca e inserimento
}
```

A questo punto la classe `DBHelper`, per inizializzare il database, non deve fare altro che chiamare il metodo `onCreate` di ogni classi ausiliaria:

```
public void onCreate(SQLiteDatabase db) {
    MessagesTableHelper.onCreate(db);
    MessagesPreviewTableHelper.onCreate(db);
    AnnTableHelper.onCreate(db);
    PenaltiesTableHelper.onCreate(db);
    UsersTableHelper.onCreate(db);
}
```

Come esempio di query osserviamo il codice del metodo di `DBHelper` per ottenere i messaggi relativi ad una conversazione con un singolo utente.

```
public ArrayList<JEHMessage> getSingleChat
        (String eventId, String name) {
    Cursor cursor = MessagesTableHelper.getSingleChat
            (getReadableDatabase(), eventId, username, name);
    ArrayList<JEHMessage> res = new ArrayList<JEHMessage>();
    while(cursor.moveToNext()) {
        res.add(new JEHMessage(JEHMessage.MSG,
                cursor.getString(0),
                cursor.getString(1),
                cursor.getString(2),
                cursor.getLong(3)));
    }
    cursor.close();
    return res;
}
```

`DBHelper` delega l'esecuzione della query al metodo statico `getSingleChat` della classe `MessagesTableHelper` e si occupa solo di trasformare il risultato in uno più comprensibile per l'applicazione, che comprende appunto l'utilizzo della classe `JEHMessage`, creata ad hoc per rappresentare i messaggi in questo progetto. Allo stesso modo vengono effettuate tutte le altre query sul database. Vediamo per completezza il metodo statico `getSingleChat` della classe ausiliaria.

```
public static Cursor getSingleChat(SQLiteDatabase dbReadable,
        String eventId, String username, String name) {
    return dbReadable.query(TABLE_NAME, new String[]
            {COLUMN_SENDER, COLUMN_DEST,
            COLUMN_TEXT, COLUMN_TIME},
            COLUMN_EVENT + "=? AND " + COLUMN_USER + "=? AND
            (" + COLUMN_SENDER + "=? OR " + COLUMN_DEST + "=?)",
            new String[] {eventId, username, name, name},
            null, null, COLUMN_TIME);
}
```

Questo metodo si occupa solamente di effettuare la query corretta, utilizzando delle costanti (i nomi in maiuscolo) per definire i nomi della tabella e delle colonne, e di restituire i dati richiesti.

Ogni classe ausiliaria contiene diverse costanti che definiscono i nomi delle tabelle e delle loro colonne, stringhe di creazione delle tabelle e di eventuali views. Avere tutte queste informazioni, per ogni tabella, nello stesso file, creerebbe molta confusione; così invece manteniamo un codice più ordinato e leggibile.

Il servizio di comunicazione XMPP

Anche in questo caso le molte utilities di Android ci vengono in aiuto. Vi è infatti la possibilità di creare un semplice servizio creando una sottoclasse di `Service`. Tramite un'interfaccia `IBinder` possiamo far comunicare la nostra applicazione con il servizio, e possiamo far eseguire il servizio su un thread separato per restare in attesa di pacchetti XMPP. Per fare ciò, usufruiamo della libreria Smack[1][6], una libreria open source puramente Java che ci permette di creare automaticamente una connessione XMPP verso il server e di restare in attesa di ricevere messaggi utilizzando appositi filtri messi a disposizione dalla libreria stessa, che possiamo sfruttare per eseguire comodamente una porzione di codice specifico all'arrivo di determinati pacchetti. Vediamo passo dopo passo il funzionamento di Smack. Inizializziamo innanzitutto la connessione all'interno della classe del servizio:

```
private AbstractXMPPConnection conn = null;
```

Nel momento in cui vogliamo effettuare il login, chiameremo il metodo `login` del servizio, che non fa altro che accettare i dati necessari per l'accesso e avviare il tentativo di login in un thread a parte, per non bloccare il thread della UI.

[1]Versione 4.1 beta, che è la prima versione a supportare nativamente Android.

```
private String host, user, pw, domain;

public void login(Bundle data) {
    user = data.getString("username");
    domain = data.getString("domain");
    host = data.getString("host");
    pw = data.getString("password");

    new LoginTask().execute();
}

private class LoginTask extends AsyncTask<Void, Void, Void> {

    @Override
    protected Void doInBackground(Void... params) {
        connect();
        login();
        return null;
    }

    private void connect() {
        XMPPTCPConnectionConfiguration connConfig =
                XMPPTCPConnectionConfiguration.builder()
                        .setUsernameAndPassword(user, pw)
                        .setServiceName(domain)
                        .setHost(host)
                        .setResource("JEH")
                        .build();

        AbstractXMPPConnection connection =
                new XMPPTCPConnection(connConfig);

        try {
            // effettua la connessione al server
            connection.connect();
            conn = connection;
        } catch {
            // ...codice di verifica delle cause degli errori
```

```
            // di connessione...
        }
    }

    private void login() {
        if(conn != null) {
            try {
                // tentativo di login con i dati inseriti nella
                // configurazione della connessione creata
                // nel metodo connect() sopra
                conn.login();

                // filtro che passa i messaggi di tipo Message
                // alla classe MyPacketListener
                PacketFilter filter =
                        new PacketTypeFilter(Message.class);
                conn.addPacketListener
                        (new MyPacketListener(), filter);

                /*
                 * Connessione alle stanze di broadcast
                 */
                // non vogliamo ricevere la history
                // dei messaggi precedenti mandati nella stanza
                DiscussionHistory history =
                        new DiscussionHistory();
                history.setMaxStanzas(0);

                MultiUserChatManager mucman =
                        MultiUserChatManager.getInstanceFor
                        (conn);
                control = mucman.getMultiUserChat
                        ("broadcast-control@conference." +
                        domain);
                control.join(user, pw, history,
                        conn.getPacketReplyTimeout());

                announcements = mucman.getMultiUserChat
```

```
                        ("broadcast@conference." + domain);
                announcements.join(user, pw, history,
                        conn.getPacketReplyTimeout());

            } catch {
                // ...codice di verifica delle cause
                // degli errori di autenticazione
            } finally {
                if (!isAuthenticated()) {
                    // chiude la connessione in caso il login
                    // non sia andato a buon fine
                    logout();
                }
            }
        }
    }
}
```

A questo punto, il nostro servizio è già connesso al server XMPP e ogni stanza di tipo `<message/>` verrà inviata al filtro definito da `MyPacketListener`. Possiamo osservare che, nel metodo `login` di `LoginTask`, viene effettuata una connessione a due `MultiUserChat`. Queste sono le due chat di gruppo, descritte nel capitolo precedente, usate per lo scambio di informazioni come il timer del turno di gioco e gli annunci in broadcast.

Vediamo infine il funzionamento del filtro dei messaggi. Il codice di questa parte è abbastanza lungo, principalmente a causa dell'elaborazione delle stringhe di testo ricevute. Per focalizzarci sulla sua logica utilizziamo dello pseudocodice.

```
private class MyPacketListener implements PacketListener {

    @Override
    public void processPacket(Packet packet)
            throws NotConnectedException {
        Message m = (Message) packet;

        // messaggio di chat normale
```

```
        if (m.getType() == Type.chat) {
            if(application_is_open) {
                receive_message(m);
            }
            else  {
                save_message_in_message_queue(m);
                // inviamo una notifica di sistema all'utente
                send_notification();
            }
        }
        // messaggio da una stanza broadcast, dobbiamo
        // scoprire da quale, analizzando il mittente
        else if (m.getType() == Type.groupchat) {
            if (fromChannel.equals("broadcast-control")) {
                operation = read_instruction(m);
                if (operation.equals("TIMERSTART")) {
                    timer = read_timer_value(m);

                    if (application_is_open)
                        receive_timer(timer);
                    else {
                        save_message_in_message_queue(m);
                        send_notification();
                    }
                }
            }
            else if (fromChannel.equals("broadcast")) {
                if (application_is_open) {
                    receive_announcement(m);
                }
                else {
                    save_message_in_message_queue(m);
                    send_notification();
                }
            }
        }
    }
}
```

I metodi `receive_message`, `receive_timer` e `receive_announcement` rappresentano il codice necessario a comunicare l'arrivo dei rispettivi messaggi all'applicazione, tramite l'interfaccia definita in fase di creazione del servizio. Se l'applicazione non fosse aperta in questo momento, però, non potremmo effettuare questa operazione, poiché ci verrebbe restituito un errore quando si cerca di modificare l'interfaccia che non è visualizzata. Dobbiamo dunque salvare i messaggi ricevuti in una coda e inviare una notifica all'utente tramite il `NotificationManager` di sistema. All'apertura dell'applicazione, l'utente riceverà tutti i messaggi che sono stati inseriti in coda, non appena l'applicazione verrà connessa nuovamente al servizio. Questa operazione viene svolta, come vediamo di seguito, seguendo la stessa logica del filtro `MyPacketListener`.

```
private void checkMsgQueue() {
    while(!msgq.isEmpty()) {
        Message m = msgq.poll();
        if (m.getType() == Type.chat)
            receive_message(m);
        else if (m.getType() == Type.groupchat) {
            if (fromChannel.equals("broadcast-control")) {
                operation = read_instruction(m);
                if (operation.equals("TIMERSTART")) {
                    timer = read_timer_value(m);
                    receive_timer(timer);
                }
            }
            else if (fromChannel.equals("broadcast")) {
                receive_announcement(m);
            }
        }
    }
}
```

Per concludere, vediamo i metodi che si occupano dell'invio di messaggi e penalità.

```
public void sendChatMessage(Bundle data) {
    Message msg = new Message();
    msg.setType(Type.chat);
    msg.setBody(data.getString("text"));
    msg.setTo(data.getString("dest") + "@" + domain);

    conn.sendPacket(msg);
}

public void sendPenalty(Bundle data) {
    long id = data.getLong("id");
    String round = data.getString("round");
    String player = data.getString("player");
    String infr = data.getString("infr");
    String penalty = data.getString("penalty");
    String desc = data.getString("desc");

    Message msg = new Message();
    msg.setType(Type.normal);
    msg.setTo("scorekeeper@" + domain);
    msg.setSubject("penalty");
    msg.setBody("NEW:::" + id + ":::" + round +
            ":::" + player + ":::" + infr +
            ":::" + penalty + ":::" + desc);

    conn.sendPacket(msg);
}
```

Approfittiamo di questi due metodi per mostrare le ultime nozioni sull'utilizzo di Smack. Per prima cosa, vediamo quanto è semplice inviare messaggi XMPP. Basta impostare destinatario e corpo del messaggio, e chiamare l'apposito metodo di invio `sendPacket`. Vediamo inoltre dagli esempi che viene rispettato il metodo di distinzione delle diverse tipologie di messaggi spiegato nel capitolo precedente, cioè tramite l'attributo `type` e l'elemento `<subject/>`, oltre ovviamente all'attributo `from`, cioè il mittente. Come ultimo dettaglio, notiamo come lo scorekeeper abbia il nickname prefissato a

"scorekeeper"; questa convenzione semplifica il lavoro di sviluppo del sistema, senza limitarne in alcun modo le funzionalità.

L'interfaccia

L'applicazione è sviluppata pensando alla compatibilità con il maggior numero possibile di dispositivi, poiché è importante che sia utilizzabile da tutti i membri dello staff. La versione minima di Android supportata è la numero 8, rendendo così possibile raggiungere quasi tutti i dispositivi in circolazione. Per fare ciò, è necessario usare le librerie di supporto di Android[7], che contengono una versione retrocompatibile delle API del framework.

L'interfaccia è studiata per essere usata solo in modalità portrait, poiché è la più comoda per usufruire al meglio degli strumenti del sistema quando probabilmente stiamo camminando e potremmo avere una mano impegnata da fogli di carta o altro. È composta solamente da due activities. La prima è una semplice `Activity` di login, in cui l'utente inserisce i dati di accesso e verrà effettuato un tentativo di login. Ad ogni tentativo di login riuscito,

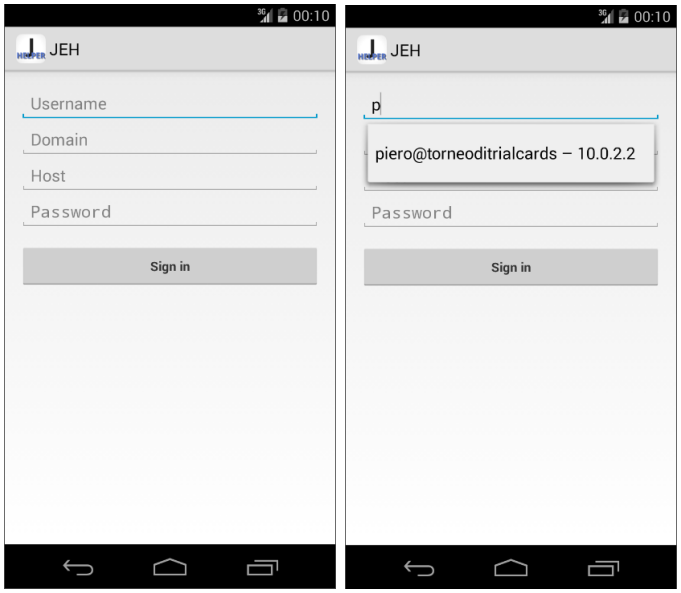

Figura 2.1: Login e autocompletamento dei dati.

verranno salvati i dati di accesso nel database, per essere utilizzati in futuro per offrire una funzione di autocompletamento dei dati di accesso. Vediamo un esempio in Figura 2.1.

L'altra `Activity` si occupa di organizzare tutti i restanti strumenti. Essa è composta da due soli oggetti, una `TextView`, che useremo per rappresentare il numero e il timer del turno corrente, e un `ViewPager`, che raccoglie diversi `Fragment`. È possibile navigare tra questi `Fragment` semplicemente con

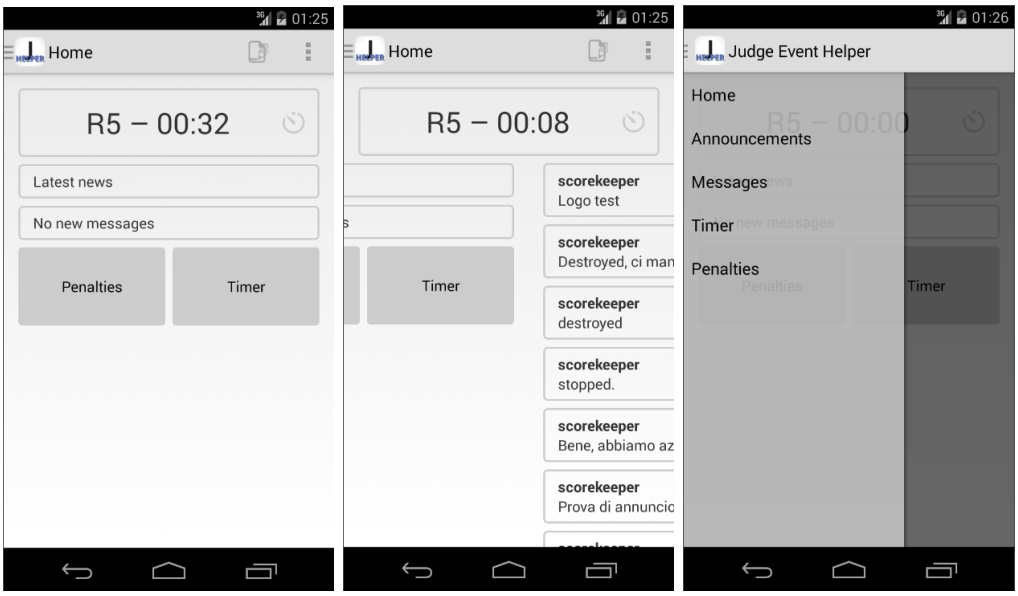

Figura 2.2: A sinistra: la home page. Al centro: animazione di cambio pagina. A destra: il menù laterale.

un gesto di trascinamento laterale del `Fragment` corrente, come se fosse uno slideshow. È inoltre possibile impostare il `Fragment` attualmente visualizzato tramite il metodo `setCurrentPage(int)` che ci permette di sfruttare al meglio il `ViewPager` con un altro importantissimo elemento dell'interfaccia: il menù di navigazione laterale (`NavigationDrawerFragment`). Vediamo il layout dell'activity (le impostazioni di margini e stili sono omesse).

```
<android.support.v4.widget.DrawerLayout
    android:id="@+id/drawer_layout" >
```

```
    <LinearLayout
        android:orientation="vertical" >

        <TextView
            android:id="@+id/round_timer" />

        <android.support.v4.view.ViewPager
            android:id="@+id/container" />
    </LinearLayout>

    <fragment
        android:id="@+id/navigation_drawer" />

</android.support.v4.widget.DrawerLayout>
```

A questo punto, è sufficiente creare un `Fragment` per ogni funzionalità che ci interessa e aggiungerli al `ViewPager`, creando opportunamente le voci nel menù laterale. Otterremo così il risultato prefissato di avere il timer in testa ad ogni schermata, sempre in vista.

In Figura 2.2 vediamo a sinistra la home page in condizioni normali con il timer che scorre. In questa schermata troviamo accesso rapido a tutte le funzionalità. I due bottoni portano rapidamente alla pagina corrispondente, mentre i due riquadri sopra i bottoni sono predisposti per contenere le anteprime dei nuovi annunci o messaggi in arrivo, che se toccate porteranno direttamente alla schermata degli annunci o messaggi, rispettivamente. Al centro è mostrata l'animazione che avviene quando trasciniamo la home page verso sinistra. Il timer rimane fermo, mentre la pagina del `ViewPager` viene cambiata e impostata alla seconda pagina, nel nostro caso gli annunci. Il funzionamento di `ViewPager` è completamente automatico, è sufficiente impostare il `FragmentPagerAdapter` che ne rappresenta il contenuto. Nel nostro caso, queste sono le poche semplici operazioni necessarie a configurarlo.

```
private static class MyAdapter extends FragmentPagerAdapter {

    public MyAdapter(FragmentManager fm) {
        super(fm);
```

```
    }

    @Override
    public Fragment getItem(int position) {
        Fragment fragment;
        switch (position) {
        case HOME:
            // mainFragment è un riferimento dell'activity
            // al fragment, che possiamo tenere per far
            // interagire i due oggetti
            mainFragment = MainFragment.newInstance();
            fragment = mainFragment;
            break;
                // creiamo un case per ogni fragment da inserire
        }
        return fragment;
    }

    @Override
    public int getCount() {
        return 5; // il numero di frammenti contenuti nel pager
    }
}
```

A differenza di quanto si possa credere leggendo il nome del metodo `getItem`, questo viene chiamato solo quando il fragment in posizione `position` deve essere creato, non ogni volta che viene visualizzato.

Impostando questo come `Adapter` per il nostro `ViewPager`, restano solo due cose da fare: far selezionare le pagine al menù laterale e gestire l'utilizzo del tasto Back di sistema, che di default ci farebbe uscire dall'applicazione in ogni caso, dal momento che operiamo sempre sulla stessa `Activity`. Per l'interazione col menù laterale, è sufficiente eseguire la riga di codice seguente ogni volte che viene selezionato un elemento:

```
public void onNavigationDrawerItemSelected(int position) {
    mPager.setCurrentItem(position);
}
```

Per la gestione del tasto Back, invece, ridefiniamo il metodo `onBackPressed` dell'`Activity` corrente. In questo caso, inizialmente la scelta era di tenere una pila in cui veniva salvata la cronologia di navigazione delle pagine, e con il tasto Back si andava a ritroso. Dai primi test si è visto subito che la scelta non era ottimale, si veniva a creare una pila molto grande rendendo praticamente impossibile uscire dall'app col tasto Back, o tornare rapidamente alla home. Proprio da quest'ultimo punto è nata l'idea che è stata sviluppata in seguito, che è quella implementata nella versione attuale: con il tasto Back si torna alla home, mentre se si è già nella home, si esce normalmente dall'applicazione. L'implementazione di `onBackPressed` che ne è risultata è:

```
@Override
public void onBackPressed() {
    if(mPager.getCurrentItem() == HOME)
        super.onBackPressed();
    else {
        mPager.setCurrentItem(HOME);
    }
}
```

Sempre in Figura 2.2, vediamo a destra il menù laterale aperto. Al primo avvio dell'applicazione dopo l'installazione, questo menù sarà aperto, per far sì che l'utente impari subito della sua esistenza. Il menù può essere aperto o chiuso trascinando con il dito da o verso il bordo sinistro dello schermo, oppure toccando l'icona in alto a sinistra in ogni schermata. Nell'immagine in alto a destra è mostrato il menù dell'applicazione, che si apre toccando l'icona coi tre quadrati in alto a destra nei sistemi nuovi, usando invece il tasto *MENU* apposito negli smartphone più datati. In questo menù troviamo solo la voce *Logout*, per disconnettersi dal server. Inoltre, sempre in alto a destra, a fianco dell'icona per aprire questo menù, c'è un'altra icona, che se toccata porta alla creazione di un nuovo messaggio, da qualsiasi pagina.

In Figura 2.3 a sinistra vediamo la pagina principale dei messaggi. L'intera interfaccia dei messaggi segue quelle che sono ormai convenzioni nel design di una chat per dispositivi mobili, cioè una pagina principale con la raccolta

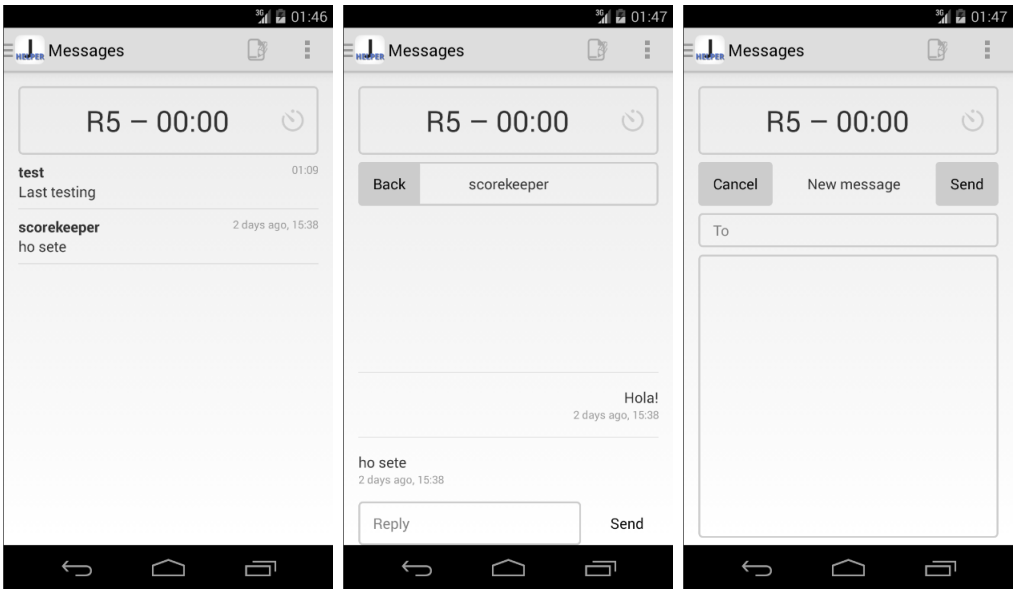

Figura 2.3: A sinistra: le conversazioni. Al centro: una chat con un utente. A destra: creazione di un nuovo messaggio.

delle conversazioni, in cui ogni riga mostra un'anteprima dell'ultimo messaggio scambiato con una determinata persona. Toccando un'anteprima si apre la conversazione con quella persona, come mostrato al centro. A destra, sempre in Figura 2.3, vediamo invece la creazione di un nuovo messaggio.

Tutta la chat è compresa in un unico `Fragment`, per fare ciò è stata usata una variabile che contiene lo stato della chat, che può essere uno tra *NEW*, *CHAT* e *LIST*. Ad ogni cambiamento di stato, generato dalle azioni dell'utente (e.g. toccando una conversazione si passa da *LIST* a *CHAT*, inviando un nuovo messaggio si passa da *NEW* a *CHAT*), viene caricato un layout nuovo nel `Fragment`, sostituendo quello precedente.

La pagina dei timer, mostrata in Figura 2.4, raccoglie i timer personalizzati dell'utente; a sinistra si vede la creazione di un nuovo timer, con la possibilità di inserire un'etichetta.

La pagina delle penalità, sempre in Figura 2.4, infine, è sviluppata seguendo lo stesso principio di quella dei timer, con la differenza che, una volta aggiunta una nuova penalità, essa verrà inviata automaticamente al

Reporter, sul pc dello scorekeeper.

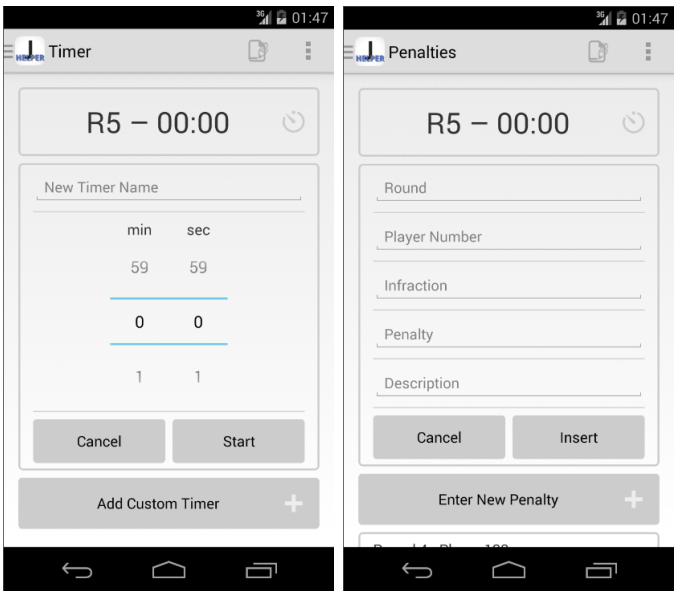

Figura 2.4: Creazione di timer e inserimento di penalità.

2.2 Il client per lo scorekeeper

Per lo sviluppo del client per pc è stato scelto il linguaggio Python con il framework Qt[8], tramite i binding offerti da PyQt4[9], che consente uno sviluppo veloce e molto semplice di un'interfaccia grafica funzionale.

Il database

Per il database, anche Python offre funzioni per facilitare l'utilizzo di Sqlite. La decisione più ovvia è dunque quella di ricreare un database con una struttura simile a quella vista per il client Android, con una classe ausiliaria per ogni tabella. Lo schema del database è leggermente diverso, dal momento che qui abbiamo anche i dati del torneo da memorizzare, oltre alla chat, ma la logica di funzionamento è la medesima. Avremo quindi le tabelle

per messaggi, conversazioni, penalità, giocatori, arbitri, incontri e classifica. Inoltre, bisogna risolvere il problema di come gestire più tornei; le due opzioni valutate sono: aggiungere di una chiave `eventID` ad ogni tabella del database, oppure tenere un database per ogni torneo. La scelta più logica sembra essere la seconda, che permette anche di spostare singoli tornei tra un pc e l'altro. Per fare ciò, creiamo un altro database di supporto con una tabella che contiene le informazioni sui nomi dei file dei database relativi a ciascun evento.

Il client XMPP

Per lo sviluppo del client XMPP la scelta questa volta è caduta sulla libreria SleekXMPP[10], che offre funzionalità equivalenti a quelle di Smack per Java/Android: anche in questo caso possiamo inviare messaggi chiamando una semplice funzione e passando i valori dei campi che ci interessano della stanza `<message/>` che vogliamo inviare. Gli unici obbligatori sono il destinatario, il tipo del messaggio e il corpo del messaggio. Il filtraggio dei pacchetti avviene in maniera leggermente diversa da Smack, ma la logica di fondo è la stessa. In questo caso, ogni pacchetto ricevuto scatenerà uno o più eventi diversi in base al tipo; alcuni esempi di eventi sono:

- `got_online` quando un'altra persona si connette al server
- `message` quando viene ricevuto un messaggio
- `groupchat_message` quando viene ricevuto un messaggio da una chat di gruppo
- `session_start` quando il login è avvenuto con successo

Collegando un'opportuna funzione ad un evento, possiamo gestire i pacchetti in arrivo esattamente come abbiamo visto sopra per l'app Android:

```
self.add_event_handler("message", self.message_received)
```

Siccome non consideriamo la necessità che l'utente chiuda l'applicazione lasciando connesso il client XMPP al server, come accade sugli smartphone, la gestione dei messaggi si semplifica di molto; è sufficiente, come vediamo ora, identificare il tipo di messaggio ricevuto e inviarlo all'apposita funzione di una classe di ausilio che funge da controller per tutto ciò che riguarda la connessione.

```
def message_received(self, msg):
    if msg["type"] == "chat":
        self.listener.message_received(msg)
    elif msg["type"] == "normal":
        if msg["subject"] == "penalty":
            self.listener.penalty_received(msg)
```

L'interfaccia

Qt segue uno schema molto semplice per lo sviluppo delle interfacce. Ha due oggetti principali, i widget e i layout; da questi due ereditano tutti gli oggetti specifici più complessi. Ad un frame dev'essere impostato un widget principale tramite la funzione `setCentralWidget`. Ad un widget può essere impostato un layout. Un layout può contenere un numero a piacere di altri widget e/o layout. Vediamo ad esempio lo scheletro principale di questo programma.

```
class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)

        cWidget = QtGui.QWidget(self)
        stacked_layout = QtGui.QStackedLayout()

        # creazione degli altri widget

        stacked_layout.addWidget(altro_widget)
        cWidget.setLayout(self.stacked_layout)
        self.setCentralWidget(cWidget)
```

`QStackedLayout` è un particolare tipo di layout che contiene più di un oggetto, ma ne mostra soltanto uno alla volta. Utilizzando questo layout otteniamo un comportamento identico a quello costruito per la chat per Android, in cui avremo le conversazioni, le singole chat e la creazione di un messaggio nuovo all'interno dello stesso widget.

Un altro oggetto interessante usato in questa interfaccia è `QTabWidget`, che ha un funzionamento simile a quello appena visto: esso contiene, infatti, più widget e ne mostra solo uno alla volta, ma in aggiunta dispone di un metodo automatico di selezione del widget da mostrare, cioè la famosa fila di tab in cima. Ogni tab ha un'etichetta che indica il suo contenuto, e cliccandola si visualizza il widget corrispondente. La creazione di un `QTabWidget` è molto simile a quella di un `QStackedLayout`.

```
self.tabs_widget = QtGui.QTabWidget()

self.event_tab = EventTab(self, self.dbhelper)
self.judges_tab = JudgesTab(self.dbhelper)
self.players_tab = PlayersTab(self, self.dbhelper)
self.rounds_tab = RoundsTab(self, self.dbhelper)
self.standings_tab = StandingsTab(self.dbhelper)
self.penalties_tab = PenaltiesTab(self.dbhelper)

# le stringhe inserite saranno mostrate nell'etichetta del tab
self.tabs_widget.addTab(self.event_tab, "Event Information")
self.tabs_widget.addTab(self.judges_tab, "Judges")
self.tabs_widget.addTab(self.players_tab, "Players")
self.tabs_widget.addTab(self.rounds_tab, "Rounds and Results")
self.tabs_widget.addTab(self.standings_tab, "Standings")
self.tabs_widget.addTab(self.penalties_tab, "Penalties")
```

Come descritto in precedenza, ai fini di questo progetto è stato sviluppato un prototipo di Reporter, che ha le funzioni base di un Reporter reale, più le funzionalità aggiunte dal nostro sistema.

Nella stessa finestra sono affiancati gli strumenti di gestione dell'evento, a sinistra, e il client della chat, a destra (Figura 2.5). Questa scelta permette allo scorekeeper di interagire con la chat, potendo tornare a lavorare

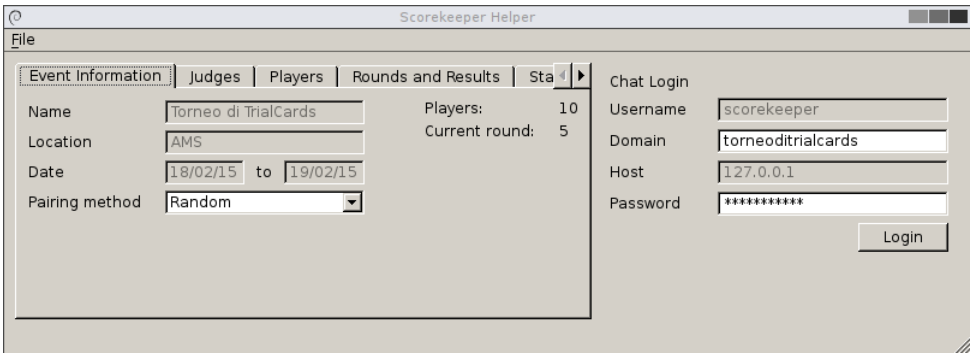

Figura 2.5: Informazioni sull'evento. Sulla destra vediamo la chat che richiede i dati di login.

sull'evento in un istante e senza preoccuparsi che un elemento del programma interferisca con l'altro in alcun modo.

Vediamo la chat, in Figura 2.6. La struttura è la stessa del client Android. Sebbene si possano creare client più efficaci su uno schermo grande come quello di un pc, in questo caso specifico è invece considerato più importante contenere lo spazio occupato, per lasciare più spazio possibile alle normali operazioni dello scorekeeper sul Reporter. Nella schermata di creazione di un nuovo messaggio notiamo un checkbox aggiuntivo rispetto al client Android. Spuntando questa casella, il messaggio inserito verrà inviato come annuncio a tutti i dispositivi mobili.

Figura 2.6: A sinistra: le conversazioni. Al centro: la chat con un utente. A destra: creazione di un nuovo messaggio.

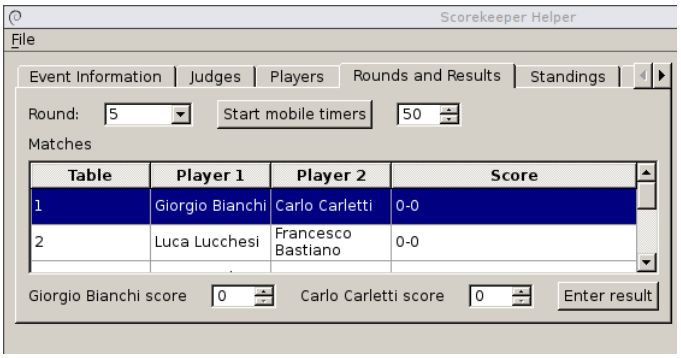

Figura 2.7: Tab per la gestione del turno di gioco in corso. In alto si vede il pulsante che avvia i timer sugli smartphone.

Come ultimo esempio vediamo la schermata di inserimento dei risultati, in Figura 2.7. Cliccando le righe della tabella verrà selezionato un incontro e sarà possibile inserirne il risultato nei campi in basso. Una volta terminati gli incontri, il programma offrirà la possibilità di creare i pairings per il turno successivo. Durante lo svolgimento del turno, invece, vediamo in alto un bottone che invia un timer ad ogni dispositivo mobile, della durata pari al valore inserito nella casella a fianco, in minuti.

Conclusioni e sviluppi futuri

Il sistema sviluppato offre uno strumento che permette di migliorare la qualità della vita e del lavoro dello staff di grandi eventi; il lavoro è incentrato su tornei di giochi da tavolo, ma un progetto simile potrebbe essere sviluppato per qualsiasi evento abbia luogo in ampi spazi, come ad esempio nel settore fieristico. Le funzionalità del sistema possono essere notevolmente ampliate a seconda delle necessità. In fase di sviluppo ci sono già:

- Possibilità di inserire una mappa del torneo, con la posizione dei tavoli e magari con la possibilità di prendere note per ogni tavolo.
- Invio di annunci da parte di un selezionato gruppo di utenti dal client mobile.
- Conversazioni di gruppo, utili poiché spesso lo staff è suddiviso in diversi team.
- Elenco dei contatti, per una maggiore usabilità dei due client di chat.

Molte altre funzioni possono essere pensate, relative per esempio al gioco a cui si va ad applicare il sistema. Ad esempio, nei giochi di carte collezionabili spesso i giocatori devono registrare il proprio mazzo su una lista e lo staff deve verificare la regolarità di ogni lista. L'app mobile potrebbe verificarne la regolarità con una foto. Oppure l'app potrebbe contenere i regolamenti completi in forma ipertestuale, in caso sia necessario sfogliarlo.

Le applicazione di un sistema come quello presentato possono essere molte, ed avere un unico dispositivo in grado di aiutare in questi eventi può portare un miglioramento notevole anche alla qualità degli eventi stessi.

Bibliografia

[1] Carli M., *Android 4. Guida per lo sviluppatore*, Apogeo, 11 Settembre 2013

[2] Summerfield M., *Rapid GUI Programming with Python and Qt. The Definitive Guide to PyQt Programming*, Prentice Hall, 28 Ottobre 2007

[3] "XMPP", http://www.xmpp.org (reperito il 04/03/2015)

[4] "Openfire Server", http://www.igniterealtime.org/projects/openfire (reperito il 04/03/2015)

[5] "SQLite", https://sqlite.org (reperito il 04/03/2015)

[6] "Smack", http://www.igniterealtime.org/projects/smack (reperito il 04/03/2015)

[7] "Android Support Library", http://developer.android.com/tools/support-library (reperito il 04/03/2015)

[8] "Qt Project", http://qt-project.org (reperito il 04/03/2015)

[9] "PyQt4", http://pyqt.sourceforge.net/Docs/PyQt4 (reperito il 04/03/2015)

[10] "SleekXMPP", https://github.com/fritzy/SleekXMPP (reperito il 04/03/2015)

Ringraziamenti

Grazie al Professor Bononi per il supporto nello svolgimento di questo lavoro in tempi veramente stretti.

Grazie alla mia famiglia, a Francesco e a tutti gli amici che mi hanno dato aiutato e fatto compagnia in questi anni di studio.

Grazie a Laura che, silenziosa, mi è sempre stata vicina.