UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI MATEMATICA

CORSO DI LAUREA IN INFORMATICA

TELEPRESENZA E VIDEO-CHIAMATE USANDO WEBRTC, EXT JS, PLAY E SCALA

TESI DI LAUREA IN INFORMATICA

RELATORE

CLAUDIO E. PALAZZI

LAUREANDO

TOMMASO FRASSETTO

ANNO ACCADEMICO 2012/2013

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

1 INTRODUZIONE

1.1 INTRODUZIONE ALL'AZIENDA

Visionest opera nell'ambito dello sviluppo *software* ed ha realizzato proprie applicazioni basate su tecnologia prevalentemente *open-source*. Il *software* prodotto da *Visionest* è basato sul paradigma del *Business Process Management* (BPM), un *trend* di grande interesse nel mercato, che prevede di realizzare le applicazioni *software* in modo tale che queste eseguano fedelmente quanto definito dai processi di *business* dei clienti, dopo che questi sono stati formalmente modellati durante un'apposita fase di analisi strutturata. L'*asset* principale realizzato da *Visionest* in questi anni è la piattaforma di sviluppo *software a>PLUS*, che risponde fedelmente a tutti gli *standard* di mercato. Sulla "dorsale tecnologica" costituita da *a>PLUS* è stata costruita negli anni una serie di pacchetti applicativi specializzati che sono orientati a specifici settori o mercati. Alcuni di essi sono:

- *Finanza 3000*: è un sistema informativo gestionale appositamente realizzato per gestire l'erogazione di prodotti di Finanza Agevolata. È utilizzabile da operatori finanziari pubblici e privati e consente la completa gestione dei fondi di terzi (fondi pubblici, statali, regionali, CEE), il rilascio di garanzie (Confidi);

- *ProERP*: è un sistema che consente all'azienda cliente di integrare l'operatività del proprio sistema gestionale "classico" (ERP), con la flessibilità dei processi insita nel *Business Process Management*. Tipicamente processi decisionali che coinvolgono più uffici o dipartimenti aziendali ed attingono informazioni da altri sistemi aziendali trovano risposta in *ProERP*: gestione ordini di acquisto, verifiche di fattibilità di prodotti e servizi, analisi dei costi/tempi della produzione. Si avvalgono di *ProERP* aziende manifatturiere di diversi settori o aziende di servizi di medie/grandi dimensioni oltre ad alcune *multiutility*;

- *d>PLUS*: è un sistema che rientra nella categoria dei *Digital Asset Management* (DAM) progettato per rendere efficien-

ti, flessibili ed economici i processi di gestione dei contenuti digitali (foto e video) nelle aziende del mercato *fashion/retail*. Centralizza i contenuti digitali ed i processi di gestione/pubblicazione, massimizzandone l'investimento trasformandoli in *asset* fruibili in modo veloce e controllabile. Questo comporta enorme efficienza e riduzione dei costi per quelle realtà del settore *retail/fashion* in cui il contenuto digitale è al centro del *business*. L'uso di *d>PLUS* apporta efficienza nei processi in ambito *Operation*, *Marketing* ed anche nella delicata fase di *Retail* (gestione campagne, *focal point*, ecc.).

1.2 INTRODUZIONE AL PROGETTO

Dato che le soluzioni applicative *Visionest* sono basate sul paradigma BPM, esse prevedono dei compiti che devono essere svolti dagli utenti che ricoprono i vari ruoli aziendali, i quali interagiscono col sistema tramite un'interfaccia *web* o *device* mobili.

La diffusione dei metodi cosiddetti *social* di scambio delle informazioni ha portato *Visionest* a considerare quali requisiti, nelle organizzazioni complesse che si dotano di *software* di *Business Process Management*, potessero essere soddisfatti da questi nuovi strumenti e quali casi d'uso, invece, potessero creativamente emergere dalla disponibilità di modi di comunicare nuovi, più semplici e più immediati (sia nel senso di più facili da usare, sia proprio nel senso di più veloci ed "in tempo reale"). L'interfaccia *web* preesistente dei *software* sviluppati da *Visionest* permette agli addetti di svolgere i loro compiti, ma è poco adatta all'attuazione di dinamiche *social*, non prevedendo alcun controllo per il caso in cui due diversi utenti svolgano lo stesso compito contemporaneamente, potenzialmente all'insaputa l'uno dell'altro e con un risultato finale incerto del compito.

Il problema può essere generalizzato introducendo il concetto di *risorsa* alla quale uno o più *utenti* si collegano; ad esempio, nel modulo di *d>PLUS* che prevede la definizione e modifica dei metadati relativi ad un'immagine di un prodotto destinato all'*e-commerce*, ogni specifico compito rappresenta una risorsa. Il progetto prevede di:

- realizzare una componente aggiuntiva all'interfaccia esistente che informi gli utenti quando si collegano contemporaneamente alla stessa risorsa;

- aggiungere alla componente sviluppata una funzionalità che permetta agli utenti collegati alla stessa risorsa di comunicare tra loro con messaggi testuali, chiamate e videochiamate, sfruttando anche la tecnologia *WebRTC*;
- integrare la nuova componente con l'interfaccia preesistente.

1.3 INTRODUZIONE A QUESTO DOCUMENTO

Il materiale di questo documento è così suddiviso:

NEL SECONDO CAPITOLO vengono introdotte le principali tecnologie utilizzate e l'architettura dell'applicazione preesistente;

NEL TERZO CAPITOLO vengono descritti i casi d'uso ed i requisiti individuati;

NEL QUARTO CAPITOLO vengono descritte l'architettura del prototipo e le sue componenti, fino alla singola classe;

NEL QUINTO CAPITOLO viene descritta l'integrazione del prototipo nell'applicazione preesistente;

NEL SESTO CAPITOLO vengono descritti i *test* che sono stati svolti sulle varie componenti e sul sistema nel complesso;

NEL SETTIMO CAPITOLO vengono esposte alcune considerazioni sul progetto e sullo *stage*.

1.3.1 *Notazioni*

In questo documento, le parole appartenenti ad una lingua straniera sono composte con il *corsivo*. Il nome di *software*, linguaggi di programmazione e simili è composto utilizzando il carattere *inclinato*. I frammenti di codice e simili sono composti con un carattere senza grazie. Le parole ed espressioni che richiedono una spiegazione non presente nel testo principale sono inserite in un glossario (a pagina 64) e sono composte come in questo esempio. Gli acronimi che richiedono una spiegazione sono riportati nell'omonima lista (a pagina 66) e sono composti in maiuscoletto, come in HTML.

2 STATO DELL'ARTE

2.1 INTRODUZIONE ALLE TECNOLOGIE UTILIZZATE

In questa sezione si introducono brevemente le principali tecnologie utilizzate nel progetto.

2.1.1 *WebRTC*

WebRTC è una API (*Application Programming Interface*) *JavaScript* che permette chiamate, video-chiamate e trasferimento di dati tra due *browser* in modalità P2P (*Peer to Peer*); il *server* interviene solo per coordinare i *browser*. *WebRTC* è al vaglio del W3C (*The World Wide Web Consortium*) (attualmente è una bozza) e per questo alcune sue implementazioni non sono ancora complete e la sua diffusione è limitata.

DIFFUSIONE DESKTOP *WebRTC* è disponibile nelle attuali versioni *release* (non *beta*) di *Chrome* e di *Firefox* sui principali sistemi operativi (*OS X*, *Linux* e *Windows*). Le quote di mercato di questi due *browser* vengono stimate[1] al 43% e 20% rispettivamente, pari al 63% del mercato.

DIFFUSIONE MOBILE La diffusione nell'ambito *mobile* è molto più ridotta; è disponibile dalla versione 29 di *Chrome* per *Android* come funzionalità stabile (che non richiede un'attivazione specifica), mentre non è implementato in alcun browser per *iOS*[2].

INTERCOMPATIBILITÀ *Chrome* e *Firefox* possono comunicare utilizzando canali video ed audio.

I canali dati sono implementati in entrambi i *browser* ma in maniera incompatibile; inoltre *Chrome* non fornisce al momento un canale dati con garanzia di integrità[3].

1 Dati di *StatCounter*, giugno 2013.
2 Esiste *Bowser*, che non è compatibile con le altre implementazioni e non viene aggiornato dal 2012.
3 Nella versione 29 esiste un'implementazione *beta* di un canale affidabile che utilizza SCTP e richiede l'attivazione esplicita dalle funzioni sperimentali.

La condivisione del video proveniente dallo schermo è disponibile solo in *Chrome* dopo essere stata abilitata esplicitamente nella pagina delle funzioni sperimentali.

2.1.2 *WebSocket*

WebSocket è una tecnologia *web* che permette ad un *browser* di mantenere una connessione bidirezionale aperta con un *server*. Le relative API *JavaScript* sono uno *standard* del W3C. Un *WebSocket* permette a *server* e *browser* di comunicare inviandosi messaggi testuali; spesso questi messaggi sono codificati in JSON (*JavaScript Object Notation*).

I *WebSocket* sono disponibili in tutte le versioni attuali dei *browser* più diffusi, mentre alcune versioni meno recenti (*in primis* le versioni precedenti alla 10 di *Internet Explorer*) non li supportano.

2.1.3 *Ext JS*

Ext JS è un *framework* *JavaScript* che permette di sviluppare applicazioni *web* interattive utilizzando componenti *standard* e riutilizzabili (*widget*). *Ext JS* introduce inoltre il concetto di classe ed ereditarietà in *JavaScript*, sollevando lo sviluppatore dalla necessità di interagire con il meccanismo nativo per l'ereditarietà (i prototipi). *Ext JS* è inoltre in grado di scaricare dal *server* e caricare in memoria esclusivamente le classi necessarie al funzionamento, tralasciando le altre.

Il sistema di classi predefinite rispetta il *pattern* MVC (*Model View Controller*), aiutando lo sviluppatore ad organizzare il codice in una maniera ragionevole e prevedibile. Le sue classi *view* sono in grado di mostrare autonomamente il contenuto proveniente dalle classi del modello; inoltre, le classi della categoria *store* sono in grado di recuperare i dati da visualizzare dal *server*, utilizzando formati *standard* come XML (*eXtensible Markup Language*) e JSON.

Ext JS utilizza solo componenti *standard* (non richiede *plugin* proprietari) ed è compatibile con tutti i *browser* moderni.

2.1.4 *Scala*

Scala è un linguaggio di programmazione orientato agli oggetti che permette anche la programmazione funzionale, consenten-

do quindi di applicare l'approccio che meglio si adatta ad ogni problema.

Scala è pensato per essere compilato in *bytecode Java*, che può essere poi eseguito in una JVM (*Java Virtual Machine*). Questa vicinanza a *Java* permette a classi scritte in *Scala* di chiamare metodi di una classe *Java* e viceversa, senza la necessità di uno strato intermedio che esegua la traduzione.

Scala ha il vantaggio di essere molto sintetico rispetto a *Java*, permettendo inoltre una notevole flessibilità sintattica. Ad esempio, in molti casi i tipi possono essere inferiti dal compilatore ed omessi nel codice; nonostante questo, *Scala* è un linguaggio fortemente tipato e conserva i vantaggi di questa categoria. La sintassi sintetica risulta poco comprensibile per un principiante ma permette ad un programmatore esperto di lavorare ad un livello più alto di astrazione, tralasciando i livelli più bassi.

Scala è stato pensato per creare applicazioni scalabili, con un buon supporto al parallelismo (derivante anche dall'aspetto funzionale). Lo stesso nome del linguaggio deriva da un'abbreviazione di *scalable language*.

2.1.5 *Play Framework*

Play è un *framework* per lo sviluppo di applicazioni *web* che aspira ad ottimizzare lo sviluppo con funzionalità come la possibilità di ricaricare le classi modificate senza dover riavviare tutto il *server* e la visualizzazione degli errori direttamente nella finestra del *browser*.

Play è progettato per funzionare in maniera asincrona, senza utilizzare chiamate bloccanti e non richiedendo quindi ai *thread* del *server* di rimanere bloccati in attesa di un evento di rete. In questa maniera può elaborare un dato volume di richieste utilizzando meno *thread* e quindi meno risorse di sistema.

La versione 1 era scritta in *Java* e supportava solo applicazioni *Java*; la versione 2, invece, è stata scritta in *Scala* e permette applicazioni *Scala*, *Java* e miste.

2.2 ARCHITETTURA DELL'APPLICAZIONE PREESISTENTE

La versione preesistente di *d>PLUS* si basa su un *server* scritto in *Java* in *Play* 1.2.5 che interagisce con una base di dati sottostante.

La parte *client* è realizzata in *Ext JS* 4.1.0. La *view* in cui si desidera inserire la componente aggiuntiva è quella che permette

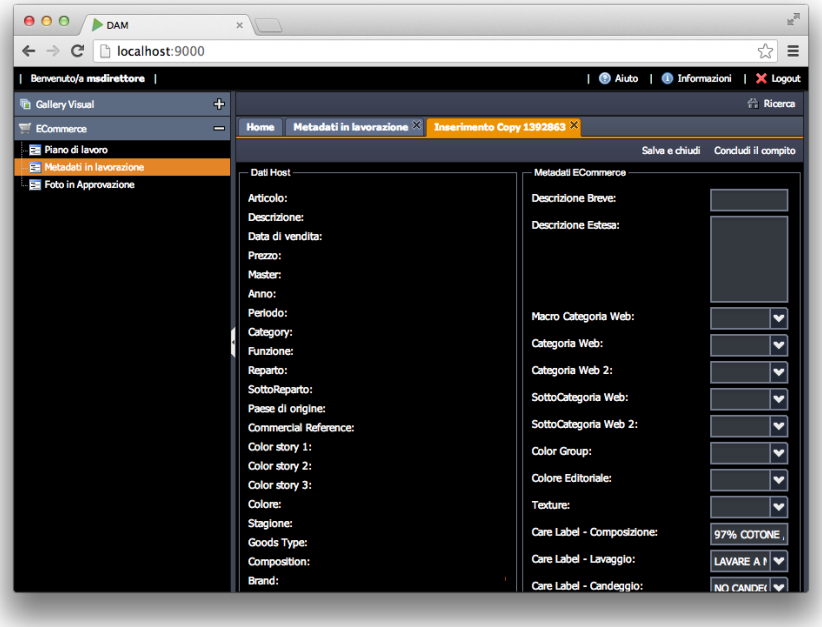

Figura 1: Interfaccia preesistente della sezione "Metadati in lavorazione" di *d>PLUS*

di modificare i metadati di un'immagine relativa ad un articolo dell'*e-commerce*; essa è chiamata "Metadati in lavorazione" ed è rappresentata dalla classe EditorArticoloECommerce. La *view* ha una *toolbar* superiore (in cui si vuole inserire la nuova componente grafica) e un corpo, costituito dall'insieme dei metadati che si possono modificare; si veda la figura 1.

3

ANALISI DEI REQUISITI

3.1 CASI D'USO

Questa sezione tratta i casi d'uso del sistema che sono stati analizzati. Si è scelto di definire il sistema come l'insieme del codice esistente di *d>PLUS* e della nuova componente; nei diagrammi tale sistema è indicato con il nome *d>PLUS integrato*.

Ai fini della componente che si vuole sviluppare non esistono differenze tra gli utenti; per questo è stato definito solo un attore, *Utente*. Visto che le funzionalità di *login* e *logout* a *d>PLUS* sono già esistenti, esse non sono prese in considerazione nell'analisi seguente; si assumerà che l'utente abbia già effettuato correttamente il *login* all'applicazione *web*.

I singoli casi d'uso hanno un identificativo come UC:*n*, dove *n* è una classificazione numerica gerarchica.

3.1.1 *UC:0 – Scenario principale*

DESCRIZIONE L'utente può connettersi ad una risorsa e successivamente disconnettersi; mentre è connesso può consultare la lista degli altri utenti collegati e iniziare una chiamata oppure inviare un messaggio testuale ad un altro utente collegato (si veda la figura 2).

PRECONDIZIONI L'utente ha effettuato il *login* a *d>PLUS*.

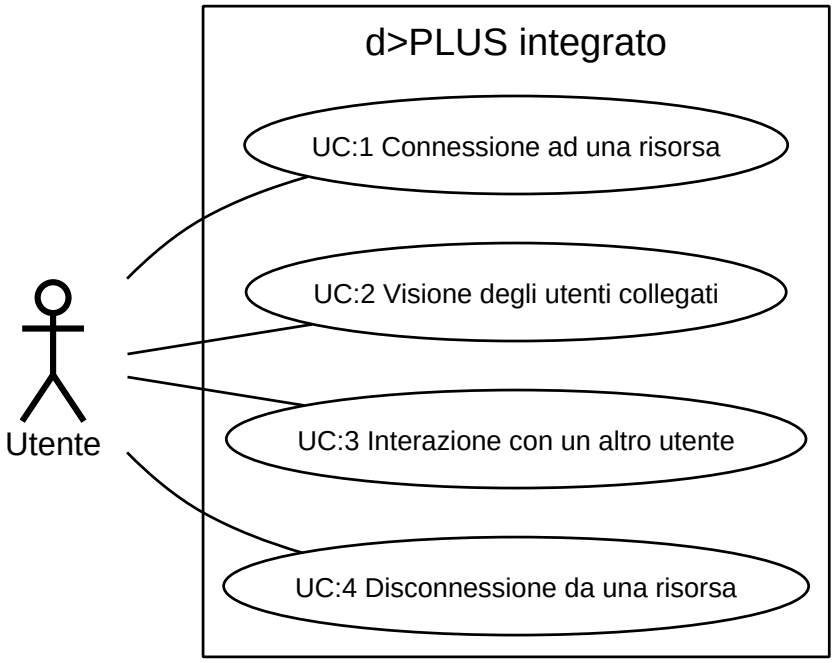

Figura 2: UC:0 – Scenario principale

3.1.2 *UC:1 – Connessione ad una risorsa*

DESCRIZIONE L'utente seleziona una risorsa da una lista e si connette alla risorsa.

PRECONDIZIONI L'utente ha effettuato il *login* a *d>PLUS*.

POSTCONDIZIONI L'utente è collegato alla risorsa selezionata; può consultare la lista degli altri utenti collegati ed è incluso nella lista consultabile dagli altri utenti collegati.

3.1.3 *UC:4 – Disconnessione da una risorsa*

DESCRIZIONE L'utente si disconnette da una risorsa a cui era precedentemente collegato.

PRECONDIZIONI L'utente è collegato alla risorsa da cui si vuole disconnettere.

POSTCONDIZIONI L'utente non è più collegato alla risorsa selezionata; non può consultare la lista degli altri utenti collegati e non è incluso nella lista visualizzata dagli altri utenti collegati.

3.1.4 *UC:2 – Visione degli utenti collegati*

DESCRIZIONE L'utente consulta la lista degli utenti collegati ad una risorsa a cui è connesso.

PRECONDIZIONI L'utente è collegato alla risorsa di cui vuole visualizzare la lista.

POSTCONDIZIONI L'utente ha visualizzato la lista degli utenti collegati alla risorsa.

3.1.5 *UC:3 – Interazione con un altro utente*

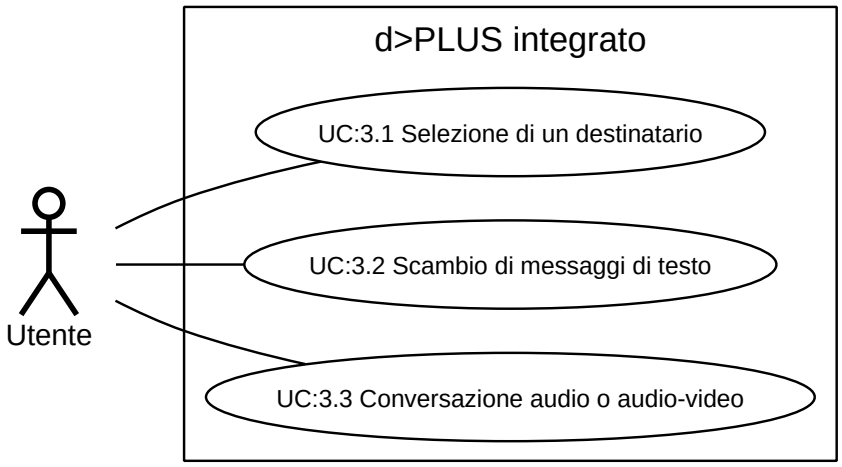

Figura 3: UC:3 – Interazione con un altro utente

DESCRIZIONE L'utente che è collegato ad una risorsa può selezionare un altro utente connesso alla stessa risorsa, scambiare con lui messaggi di testo ed avere con lui una conversazione audio o audio-video (si veda la figura 3).

PRECONDIZIONI Il mittente ed il destinatario sono connessi alla stessa risorsa.

DESCRIZIONE L'utente seleziona un utente collegato ad una risorsa a cui entrambi sono connessi.

PRECONDIZIONI Entrambi gli utenti sono collegati alla stessa risorsa.

POSTCONDIZIONI L'utente ha selezionato il destinatario desiderato.

3.1.6 *UC:3.2 – Scambio di messaggi di testo*

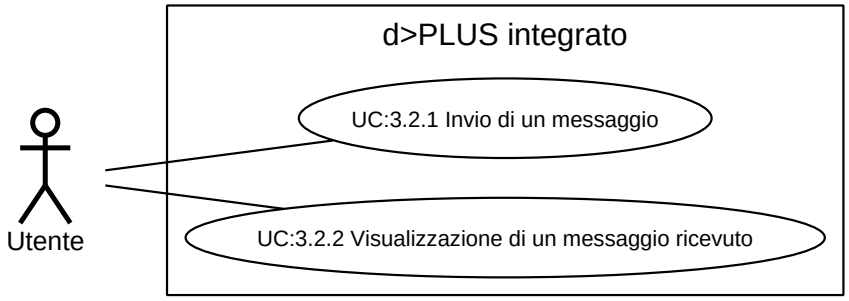

Figura 4: UC:3.2 – Scambio di messaggi di testo

DESCRIZIONE L'utente può inviare e ricevere messaggi di testo (si veda la figura 4).

PRECONDIZIONI L'utente è collegato ad una risorsa.

3.1.7 *UC:3.2.1 – Invio di un messaggio*

DESCRIZIONE L'utente scrive ed invia un messaggio ad un destinatario che ha selezionato.

PRECONDIZIONI Il mittente ha selezionato il destinatario.

SCENARIO PRINCIPALE

- L'utente scrive il testo del messaggio.
- L'utente invia il messaggio al destinatario.

POSTCONDIZIONI Il destinatario può leggere il messaggio inviato. Se il destinatario si è collegato da postazioni multiple, ognuna di esse ha ricevuto il messaggio.

3.1.8 *UC:3.2.2 – Visualizzazione di un messaggio ricevuto*

DESCRIZIONE L'utente visualizza un messaggio che un altro utente gli ha inviato.

PRECONDIZIONI Un altro utente gli ha inviato un messaggio.

POSTCONDIZIONI Il destinatario può leggere il messaggio inviato.

3.1.9 *UC:3.3 – Conversazione audio o audio-video*

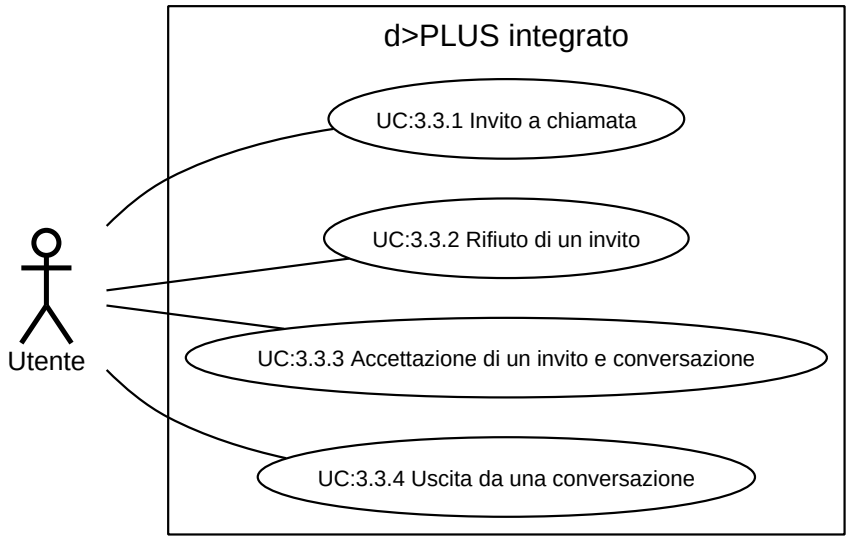

Figura 5: UC:3.3 – Conversazione audio o audio-video

DESCRIZIONE L'utente può invitare altri utenti in una conversazione, accettare e rifiutare inviti, partecipare alla conversazione ed uscire da essa (si veda la figura 5).

PRECONDIZIONI L'utente è collegato ad una risorsa.

3.1.10 *UC:3.3.1 – Invito a chiamata*

DESCRIZIONE L'utente invita un altro utente ad una chiamata o video-chiamata.

PRECONDIZIONI L'utente ha selezionato il destinatario. L'utente non ha alcuna conversazione in corso oppure ha una o più conversazioni con utenti diversi dal destinatario collegati alla stessa risorsa.

POSTCONDIZIONI Il destinatario visualizza l'invito. Se il destinatario è collegato da più di una postazione, ognuna di esse visualizza l'invito.

SCENARIO ALTERNATIVO Se il destinatario è occupato oppure utilizza *browser* che non supportano le necessarie tecnologie, il mittente visualizza un messaggio informativo.

3.1.11 *UC:3.3.2 – Rifiuto di un invito*

DESCRIZIONE L'utente rifiuta un invito che ha ricevuto.

PRECONDIZIONI L'utente ha ricevuto un invito.

POSTCONDIZIONI Il mittente dell'invito visualizza un messaggio.

3.1.12 *UC:3.3.3 – Accettazione di un invito e conversazione*

DESCRIZIONE L'utente accetta un invito che ha ricevuto.

PRECONDIZIONI L'utente ha ricevuto un invito.

POSTCONDIZIONI Viene attivata una connessione tra il mittente ed il destinatario; se il mittente era in connessione con altri utenti viene aperta una connessione tra ognuno di essi ed il destinatario. Quando le connessioni sono state stabilite avviene la conversazione.

3.1.13 *UC:3.3.4 – Uscita da una conversazione*

DESCRIZIONE L'utente abbandona una conversazione a cui stava partecipando.

PRECONDIZIONI L'utente ha una conversazione attiva.

POSTCONDIZIONI L'utente esce dalla conversazione ed è in grado di iniziarne altre. Se nella conversazione abbandonata ci sono ancora almeno due partecipanti la connessione tra di essi viene mantenuta ed è possibile continuare la conversazione.

3.2 REQUISITI

In questa sezione sono indicati i requisiti che sono stati ricavati dall'analisi dei casi d'uso sopra esposti e dagli altri vincoli presenti. Il codice identificativo di un requisito è formato dal prefisso R: e dalla concatenazione dei seguenti elementi:

1. un indicatore del tipo di requisito, tratto dai seguenti:
   - F per i requisiti funzionali: i servizi che il sistema deve fornire;

V per i requisiti di vincolo: i vincoli imposti da proponente e committente, riguardanti in particolare le tecnologie da utilizzare e l'ambiente in cui il *software* verrà utilizzato.

2. un indicatore della rilevanza, ispirato al metodo MoSCoW, tratto dai seguenti:
   - M per i requisiti obbligatori: i requisiti minimi che il sistema deve rispettare per essere considerato accettabile;
   - S per i requisiti desiderabili: requisiti importanti per il successo del prodotto che però possono essere sviluppati in un secondo momento oppure tralasciati;
   - C per i requisiti opzionali: requisiti meno importanti che però aumentano il valore del prodotto;
   - W per i requisiti di opportunità: requisiti meno rilevanti in assoluto che verranno trattati solo in caso di anticipo nello sviluppo;
3. uno o più numeri separati da punti. La parte numerica è univoca e gerarchica.

3.2.1 *Requisiti funzionali*

Identificatore	Descrizione	Fonti
R:FM1	Il sistema deve permettere agli utenti di collegarsi ad una risorsa	UC:1
R:FC1.1	Il sistema dovrebbe permettere ad un utente di connettersi alla stessa risorsa da più postazioni contemporaneamente	UC:1
R:FM2	Il sistema deve permettere ad un utente di disconnettersi da una risorsa a cui è connesso	UC:4
R:FM3	Il sistema deve permettere agli utenti connessi ad una risorsa di consultare la lista degli altri utenti connessi alla stessa risorsa	UC:2

Continua nella prossima pagina

Identificatore	Descrizione	Fonti
R:FM3.1	Il sistema deve informare gli altri utenti connessi ad una risorsa quando un utente si connette	UC:1, UC:2
R:FM3.2	Il sistema deve informare gli altri utenti connessi ad una risorsa quando un utente si disconnette	UC:2, UC:4
R:FM4	Il sistema deve permettere ad un utente di inviare un messaggio testuale ad un altro utente connesso alla stessa risorsa	UC:3, UC:3.1, UC:3.2, UC:3.2.1, UC:3.2.2
R:FC4.1	Il sistema dovrebbe inoltrare il messaggio testuale destinato ad un utente a tutte le postazioni dalle quali quell'utente è connesso	UC:3.2.2
R:FS5	Il sistema dovrebbe permettere ad un utente di invitare ad una chiamata o video-chiamata un altro utente connesso alla stessa risorsa	UC:3, UC:3.1, UC:3.3.1
R:FS5.1	Il sistema dovrebbe permettere di accettare un invito	UC:3.3.3
R:FS5.2	Il sistema dovrebbe permettere di rifiutare un invito	UC:3.3.2
R:FC5.3	Il sistema dovrebbe inoltrare un invito a chiamata o video-chiamata a tutte le postazioni da cui il destinatario è connesso e che supportano le necessarie tecnologie	UC:3.3.1
R:FC5.3.1	Quando il destinatario di un invito accetta o rifiuta in una postazione, le altre postazioni da cui è connesso dovrebbero considerare l'invito come non più pendente	UC:3.3.2, UC:3.3.3

Continua nella prossima pagina

Identificatore	Descrizione	Fonti
R:FC5.3.2	Quando un utente si collega ad una risorsa da una nuova postazione, se ha inviti in sospeso nelle postazioni preesistenti anche la nuova postazione dovrebbe essere in grado di accettarlo o rifiutarlo	UC:1, UC:3.3.1, UC:3.3.2
R:FC5.3.3	Quando un utente si scollega da tutte le postazioni, se ha inviti in sospeso questi dovrebbero essere rifiutati d'ufficio	UC:3.3.1, UC:4
R:FS5.4	Il sistema dovrebbe annullare d'ufficio un invito se non riceve risposta per 30 secondi	UC:3.3.1
R:FS5.5	Il sistema dovrebbe rifiutare d'ufficio inviti rivolti ad utenti già coinvolti in una conversazione oppure che hanno già un altro invito pendente	UC:3.3.1
R:FS5.6	Il sistema dovrebbe rifiutare d'ufficio inviti rivolti ad utenti che sono connessi da postazioni che non supportano le tecnologie necessarie per le chiamate e le video-chiamate	UC:3.3.1
R:FS6	Il sistema dovrebbe permettere ad un utente di partecipare ad una chiamata o video-chiamata con un altro utente connesso alla stessa risorsa	UC:3.3, UC:3.3.3
R:FS6.1	Il sistema dovrebbe permettere ad un utente di partecipare ad una chiamata con un altro utente connesso alla stessa risorsa	UC:3.3.3
R:FS6.2	Il sistema dovrebbe permettere ad un utente di partecipare ad una video-chiamata con un altro utente connesso alla stessa risorsa	UC:3.3.3

Continua nella prossima pagina

Identificatore	Descrizione	Fonti
R:FS6.3	Il sistema dovrebbe permettere di chiudere una conversazione precedentemente attiva	UC:3.3.4
R:FC7	Il sistema dovrebbe permettere ad un utente che sta partecipando ad una chiamata o video-chiamata di invitare ulteriori utenti (già connessi alla risorsa e attualmente non coinvolti in altre conversazioni)	UC:3.3.1

Tabella 1: Elenco dei requisiti funzionali

3.2.2 *Requisiti di vincolo*

Identificatore	Descrizione
R:VM10	Il sistema non deve permettere ad utenti non autorizzati di collegarsi e interagire con gli utenti autorizzati
R:VM11	Il sistema deve degradare in maniera adeguata nei *browser* che non supportano tutte le tecnologie richieste

Tabella 2: Elenco dei requisiti di vincolo

3.2.3 *Tracciamento inverso*

In questa sezione è riportato il tracciamento tra i casi d'uso ed i requisiti che emergono da ognuno.

Codice	Descrizione	Requisiti
UC:1	Connessione ad una risorsa	R:FM1, R:FC1.1, R:FM3.1, R:FC5.3.2
UC:2	Visione degli utenti collegati	R:FM3, R:FM3.1, R:FM3.2
UC:3	Interazione con un altro utente	R:FM4, R:FS5

Continua nella prossima pagina

Codice	Descrizione	Requisiti
UC:3.1	Selezione di un destinatario	R:FM4, R:FS5
UC:3.2	Scambio di messaggi di testo	R:FM4
UC:3.2.1	Invio di un messaggio	R:FM4
UC:3.2.2	Visualizzazione di un messaggio ricevuto	R:FM4, R:FC4.1
UC:3.3	Conversazione audio o audio-video	R:FS6
UC:3.3.1	Invito a chiamata	R:FS5, R:FC5.3, R:FC5.3.2, R:FC5.3.3, R:FS5.4, R:FS5.5, R:FS5.6, R:FC7
UC:3.3.2	Rifiuto di un invito	R:FS5.2, R:FC5.3.1, R:FC5.3.2
UC:3.3.3	Accettazione di un invito e conversazione	R:FS5.1, R:FC5.3.1, R:FS6, R:FS6.1, R:FS6.2
UC:3.3.4	Uscita da una conversazione	R:FS6.3
UC:4	Disconnessione da una risorsa	R:FM2, R:FM3.2, R:FC5.3.3

Tabella 3: Tracciamento inverso casi d'uso – requisiti

4

PROGETTAZIONE ED IMPLEMENTAZIONE DEL PROTOTIPO

4.1 ARCHITETTURA

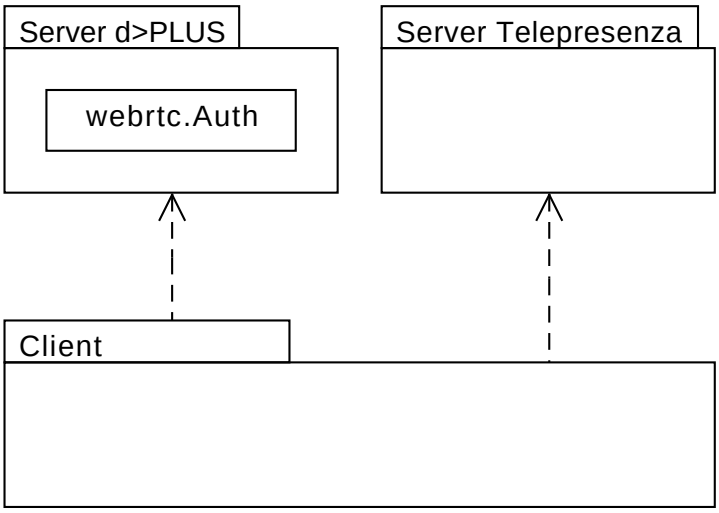

Figura 6: Architettura generale della componente aggiuntiva

Si è deciso di adottare l'architettura descritta nella figura 6.

La logica necessaria per la lista degli utenti connessi e le chiamate e video-chiamate è inclusa nella componente denominata *Server Telepresenza*, che sarà un processo separato. Questa componente non ha alcun accesso al *Server d>PLUS*, e quindi nemmeno alle credenziali utente.

Per applicare l'autenticazione in assenza di un accesso alle credenziali, il *Server Telepresenza* riceve un token di autenticazione dal *Client* che a sua volta lo ha richiesto al *Server d>PLUS*; le due componenti *server* possiedono a questo scopo un segreto condiviso. L'unica modifica al *Server d>PLUS* è quindi l'aggiunta di questa funzionalità, implementata dalla classe webrtc.Auth ed alcune altre.

Questa architettura è stata scelta per i seguenti motivi:

1. minimizza le modifiche al *server* preesistente, riducendo quindi le possibilità di introdurre errori in esso;
2. minimizza la quantità di codice che ha accesso ad informazioni sensibili;

3. rende il *Server Telepresenza* completamente indipendente dal *Server d>PLUS*, permettendo di svilupparlo utilizzando tecnologie differenti: *Play* 2.1.1 invece di 1.2.5, *Scala* invece di *Java*;

4. permette di gestire il *Server Telepresenza* indipendentemente, ad esempio installandolo in una macchina diversa.

4.1.1 *Il prototipo*

Per poter lavorare senza la necessità di un collegamento con il *Server d>PLUS*, si è scelto di sviluppare come prima cosa un prototipo, che consiste in:

- una componente *server*, costituita dal codice del *Server Telepresenza* unito ad Auth ed alle altre classi che dovranno essere inserite nel *Server d>PLUS*. Il codice del *Server Telepresenza* è stato sviluppato in *Scala*, mentre le classi destinate al *Server d>PLUS* sono state scritte in *Java* per facilitare il loro inserimento;

- una componente *client*, sviluppata a partire da un *mockup* minimale dell'applicazione esistente, a cui sono state aggiunte le nuove funzionalità; il *mockup* è stato progettato in modo da essere più simile possibile all'originale per facilitare l'integrazione del nuovo codice con il *Client* esistente.

La parte rimanente del capitolo espone la progettazione e l'implementazione del prototipo. Il prossimo capitolo espone invece l'integrazione con l'applicazione preesistente.

4.2 PROTOCOLLO

Questa sezione descrive il protocollo utilizzato nella comunicazione tra *client* e *server*.

4.2.1 *Connessione*

Quando il *browser* desidera annunciare la propria connessione ad una risorsa, apre una connessione *WebSocket* con un URL (*Uniform Resource Locator*) del tipo

```
ws://⟨server⟩/ws/⟨id-risorsa⟩?utente=⟨id-utente⟩
&timestamp=⟨timestamp⟩&hash=⟨hash⟩&rtc=⟨rtc⟩
```

Il campo ⟨server⟩ indica l'indirizzo e la porta del *Server Telepresenza*. Il campo ⟨rtc⟩ è true se il *browser* supporta *WebRTC*, false altrimenti. I campi ⟨timestamp⟩ e ⟨hash⟩ vengono utilizzati per l'autenticazione. Auth calcola uno *hash* a partire da id utente, id della risorsa, *timestamp* e un segreto condiviso; comunica poi al *client* solo gli id, il *timestamp* e lo *hash*. Il *Server Telepresenza* può quindi verificare che il *client* è lecito. Il *timestamp* permette di scartare gli *hash* più vecchi di un tempo prefissato, al fine di proteggersi da *replay attack*.

Quando smette di usare la risorsa, il *client* chiude la connessione *WebSocket* segnalando così al *server* la propria disconnessione.

4.2.2 *Tipi di messaggio*

Tutti i messaggi che passano sul *WebSocket* sono serializzati in formato JSON. I messaggi ping e pong sono trasmessi come semplici stringhe JSON; ogni altro messaggio è un oggetto JSON, la cui proprietà tipo (che è sempre definita) dichiara il tipo del messaggio, che specifica anche quali altre proprietà debbano essere definite.

I *client* possono chiedere al *server* di creare un circuito per comunicare con altri *client* connessi alla stessa risorsa; i *client* attualmente connessi ad un circuito possono invitare altri *client* non connessi. Ogni *client* connesso ad un circuito può richiedere di essere inserito in un suo sotto-circuito; infine, un *client* può inviare un messaggio su un sotto-circuito, che viene inoltrato agli altri *client* connessi al sotto-circuito (si veda la figura 7 a pagina 24).

4.2.3 *Messaggi (dal client al server)*

Sono qui elencati i tipi dei messaggi inviati dal *client* al *server*, insieme alle eventuali altre proprietà richieste (indicate tra parentesi, con il rispettivo tipo) ed al loro significato.

- msg (dest: String, msg: Object): indica che il mittente desidera inviare l'oggetto msg a tutti i *client* il cui nome utente sia dest.

- circ.invita (dest: String, dett: Object): indica che il mittente desidera aprire un circuito con uno dei *client* connessi con il nome utente dest, allegando all'invito l'oggetto dett.

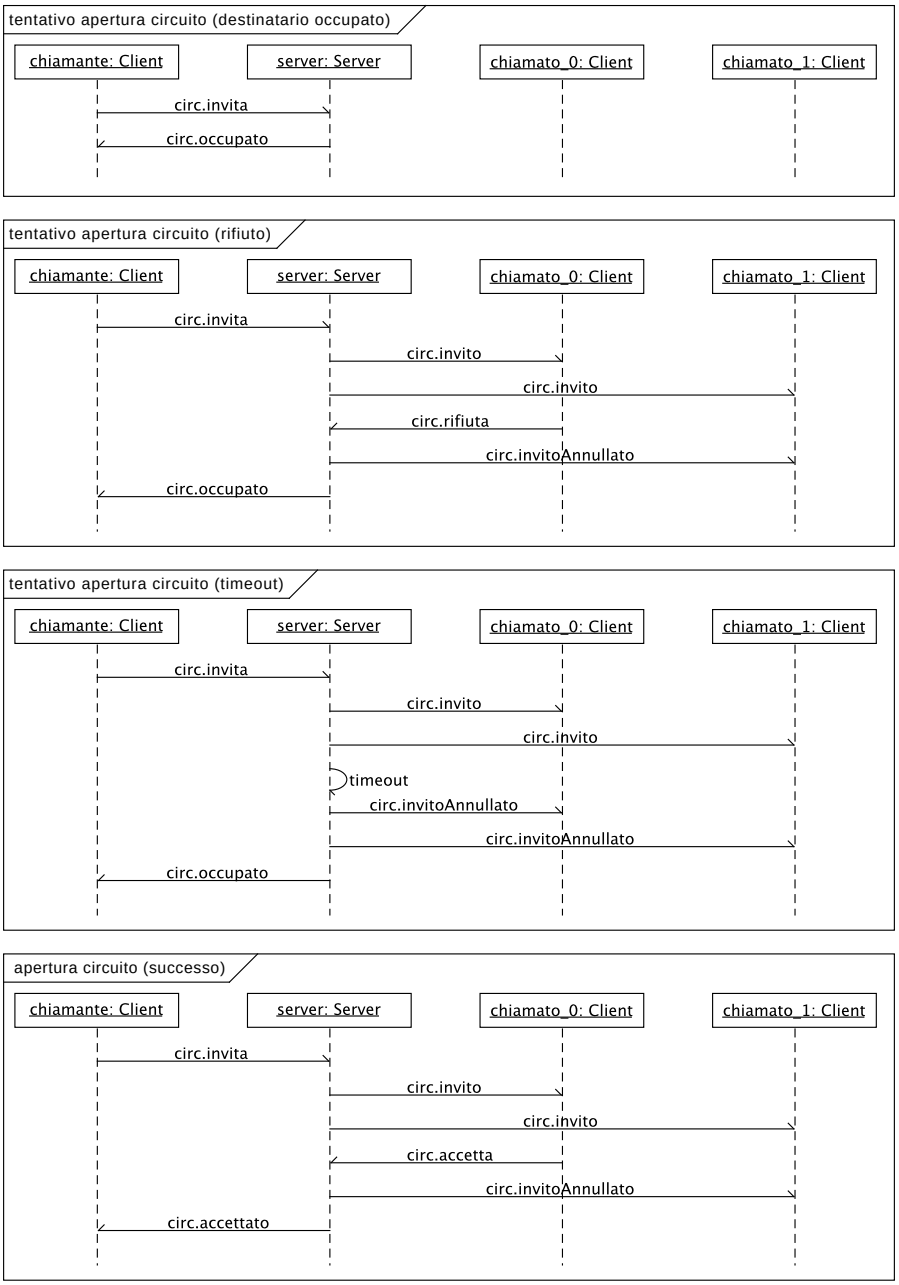

Figura 7: Diagrammi di sequenza dell'interazione tra più *client* ed il *Server Telepresenza* durante l'apertura di un circuito; chiamato_0 e chiamato_1 sono collegati con le stesse credenziali. Dall'alto, tentativo di apertura di un circuito quando il destinatario è occupato; tentativo di apertura di un circuito quando il destinatario rifiuta; tentativo di apertura di un circuito quando il destinatario non risponde; apertura di un circuito andata a buon fine. La proprietà JSON tipo dei messaggi scambiati tra *server* e *client* è indicata in figura.

- circ.accetta: indica che il mittente accetta l'invito che gli era stato inviato.
- circ.rifiuta: indica che il mittente rifiuta l'invito che gli era stato inviato.
- circ.abbandona: indica che il mittente intende uscire dal circuito a cui è attualmente connesso.
- circ.cmd.addSub (sub: String): indica che il mittente si iscrive al sotto-circuito sub.
- circ.cmd.msg (sub: String, dett: Object): indica che il mittente vuole inviare l'oggetto dett al sotto-circuito sub.

4.2.4 *Notifiche (dal server al client)*

Sono qui elencati i tipi delle notifiche inviate dal *server* al *client*, insieme alle eventuali altre proprietà richieste (indicate tra parentesi, con il rispettivo tipo) ed al loro significato.

- lista (lista: Object): informa i *client* delle altre connessioni alla stessa risorsa. Le proprietà dell'oggetto lista hanno come chiave il nome di un utente connesso alla risorsa e come valore un booleano che indica se l'utente utilizza un *browser* che supporta le chiamate e video-chiamate.
- msg (mitt: String, msg: Object): indica che l'utente mitt ha inviato l'oggetto msg.
- circ.invito (mitt: String, dett: Object): indica che l'utente mitt ha inviato un invito allegando l'oggetto dett.
- circ.invitoAnnullato: indica che l'invito precedentemente inviato non è più valido.
- circ.occupato (mitt: String, info: String): indica che l'invito inviato all'utente mitt è stato rifiutato; la stringa info può essere scaduto (se il destinatario non ha risposto), rifiutato, nonCollegato (se l'utente desiderato non risulta collegato), occupato e nortc (se il destinatario non utilizza un *browser* che supporta *WebRTC*).
- circ.accettato: indica che l'invito precedentemente inviato è stato accettato.

- circ.utAbb (mitt: String): indica che l'utente mitt ha abbandonato il circuito.

- circ.msg (mitt: String, sub: String, data: Object): indica che l'utente mitt ha inviato l'oggetto data sul sotto-canale sub.

4.3 SERVER (JAVA)

In questa sezione sono descritte le classi aggiuntive al *Server d>PLUS*; il diagramma in figura 8 ne presenta una visione d'insieme.

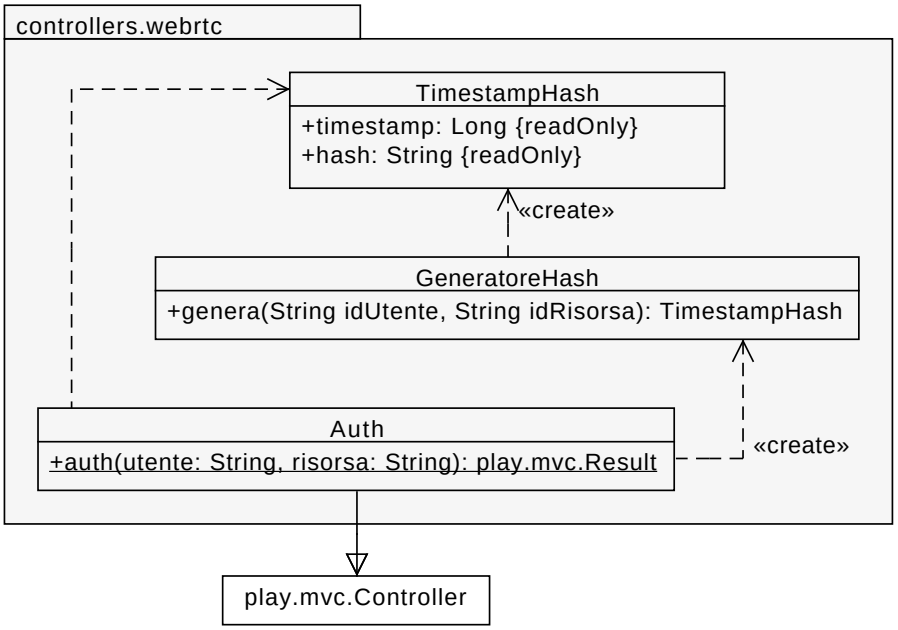

Figura 8: Diagramma delle classi relativo alle classi aggiuntive al *Server d>PLUS*, con i principali metodi

4.3.1 *TimestampHash*

Le istanze della classe GeneratoreHash rappresentano una coppia *(t, h)*, in cui *h* è uno *hash* e *t* è il relativo *timestamp*. Le istanze vengono inizializzate con i due parametri, che sono poi definitivi e non più modificabili. I due parametri sono accessibili tramite gli omonimi campi dati pubblici.

4.3.2 *GeneratoreHash*

Le istanze della classe GeneratoreHash vengono inizializzate con due stringhe che rappresentano l'algoritmo di *hashing* ed il segreto da includere nello *hash*; hanno un metodo:

- genera(idUtente: String, idRisorsa: String): TimestampHash: produce uno *hash* per i parametri dati; per fare questo concatena i due id, il *timestamp* corrente ed il segreto (separati da '$') e ne calcola lo *hash*. Restituisce *hash* e *timestamp*.

4.3.3 *Auth*

La classe ha un metodo statico:

- auth(utente: String, risorsa: String): Result: viene invocato dai *client* per ottenere uno *hash* con relativo *timestamp*. Crea un GeneratoreHash e lo usa per generare uno *hash*, che poi invia al *client*. La versione sviluppata in questo progetto non esegue alcun controllo sull'identità del *client*; sarà compito dell'azienda implementare la logica di controllo.

4.4 SERVER (SCALA)

La struttura del *Server Telepresenza* è la seguente. Il punto d'accesso è il metodo ws dell'oggetto Telepresenza, che gestisce l'apertura e la chiusura di un singolo *WebSocket* e la verifica del token di autenticazione (utilizzando Validatore). Le connessioni, disconnessioni e richieste sono inviate alla giusta Connessione; l'abbinamento tra un nuovo *WebSocket* e una Connessione (nuova o esistente) è svolto da un GestoreRisorse. Ogni Risorsa ha una lista di Utente, che rappresentano ogni utente connesso alla risorsa; ogni Utente ha a sua volta una lista di Connessione, che rappresentano ogni postazione da cui l'utente è connesso. Infine, una Connessione può avere un riferimento ad un Circuito, quando è collegata ad uno di essi. La struttura è riassunta nella figura 9.

Si è scelto di implementare Connessione come classe interna di Utente, che è a sua volta una classe interna di Risorsa.

Nelle prossime sezioni sono descritti nel dettaglio gli oggetti e le classi.

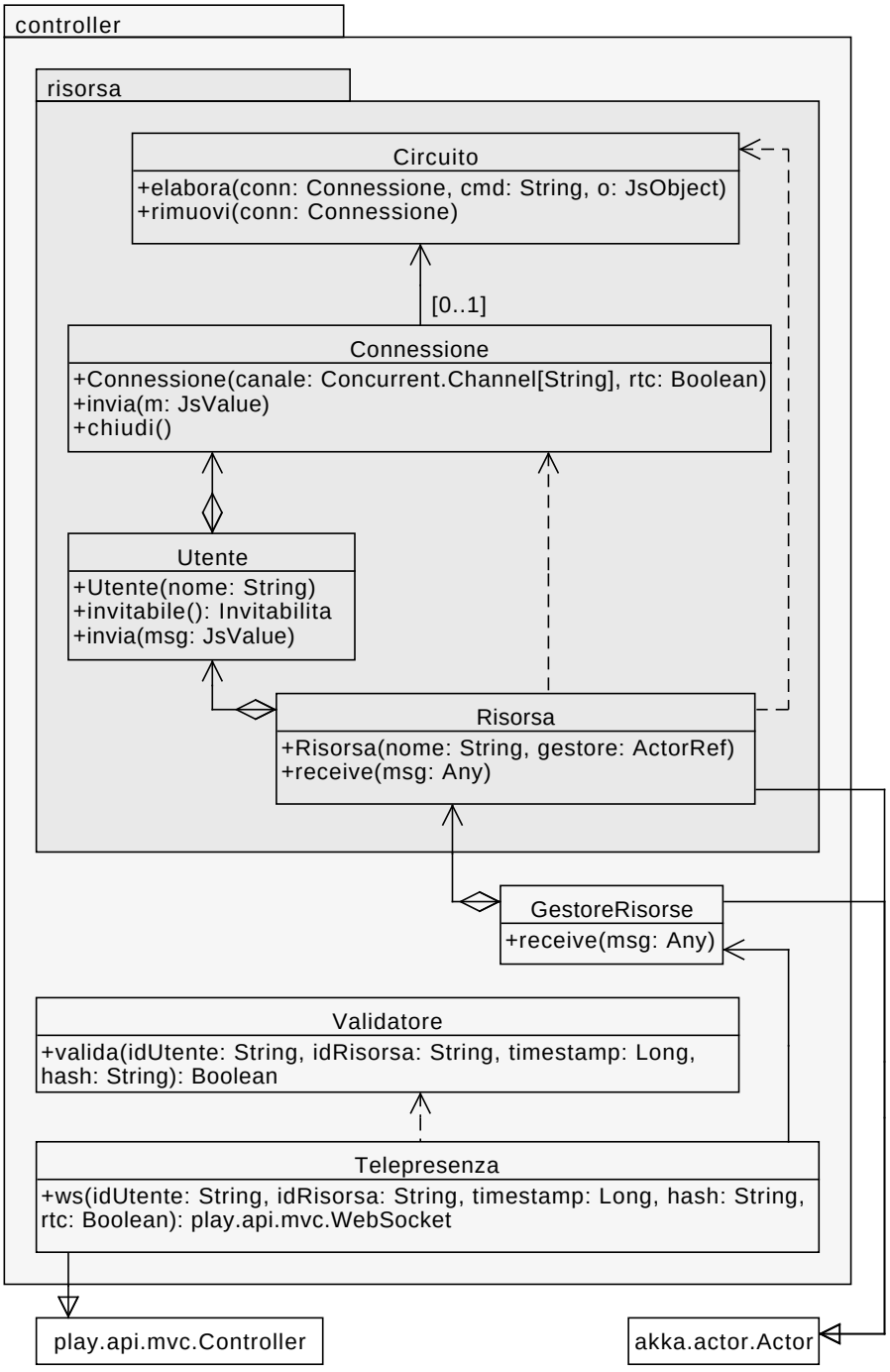

Figura 9: Diagramma delle classi relativo al *Server Telepresenza* con i principali metodi

4.4.1 *Validatore*

L'oggetto ha un metodo:

- valida(idUtente: String, idRisorsa: String, timestamp: Long, hash: String): Boolean: serve a verificare se le credenziali inviate dal *client* sono valide. Il segreto ed il nome dell'algoritmo di *hashing* vengono tratti dal *file* di configurazione `application.conf`. Il metodo restituisce true se e solo se lo *hash* ricevuto è uguale a quello calcolato e il *timestamp* è più recente di una certa soglia, ricavata a sua volta dal *file* di configurazione.

4.4.2 *Circuito*

La classe rappresenta un circuito a cui i *client* possono collegarsi, iscrivendosi ad uno o più sotto-circuiti e inviando e ricevendo su di essi messaggi. I sotto-circuiti sono mantenuti in memoria con una mutable.HashMap[String, Set[Connessione]]. I metodi sono:

- elabora(conn: Connessione, cmd: String, o: JsObject): riceve da una Risorsa un comando inviato da un *client*, cioè un messaggio con il tipo che inizia con circ.cmd (si veda la sezione 4.2.3 a pagina 23). Se il comando è corretto invoca uno dei metodi privati per gestirlo, altrimenti registra un messaggio di errore nel Logger;
- rimuovi(conn: Connessione): rimuove da tutti i sotto-circuiti conn, inviando ai *client* connessi un messaggio che indica la disconnessione;
- addSub(conn: Connessione, sub: String) (priv.): aggiunge la connessione conn al sotto-circuito sub;
- msg(conn: Connessione, sub: String, data: JsObject) (priv.): invia il messaggio data a tutti i *client* connessi al sotto-circuito sub.

4.4.3 *Risorsa*

La classe gestisce le informazioni relative ad una singola risorsa in uso da uno o più *client*; è implementata come una sottoclasse di akka.actor.Actor per garantire la correttezza dell'accesso

concorrente. In questa maniera le richieste in arrivo vengono elaborate una ad una (iniziando dal metodo receive), senza la possibilità di un accesso contemporaneo da parte di più *thread*.

4.4.3.1 *Risorsa#Utente#Connessione*

La classe Connessione viene inizializzata con i parametri canale, di tipo Concurrent.Channel, e rtc, un booleano che indica se il *browser* supporta *WebRTC*. Ha inoltre i campi dati ultimoPong, *var* che rappresenta il *timestamp* dell'ultimo messaggio pong ricevuto dalla connessione, inAttesa, *var* che è l'insieme di Utente a cui il *browser* ha mandato inviti che sono ancora pendenti, e circuito, *var* di tipo Option[Circuito], che contiene, se esiste, il Circuito a cui la Connessione è collegata.

I metodi sono:

- invia(m: JsValue): serializza il messaggio m e lo invia sul canale;
- chiudi(): chiude il canale;
- erroreStato(op: String): invia al Logger un messaggio di errore relativo ad una operazione op che allo stato attuale è illegale;
- inviaErroreChiamata(chiamato: String, info: String): invia al *client* un messaggio di occupato relativo all'invito a chiamato, includendo come motivo la stringa info;
- inviaAccettazioneChiamata(altro: String): invia al *client* un messaggio per informarlo che il suo invito è stato accettato.

4.4.3.2 *Risorsa#Utente*

La classe Utente viene inizializzata con un nome. Ha inoltre i campi dati connessioni, che è un insieme mutevole di Connessione appartenenti all'Utente, e invitato, *var* di tipo Option[(Utente#Connessione, Cancellable, JsObject)]; quando l'utente ha un invito pendente, questo campo contiene l'oggetto Connessione del chiamante, l'oggetto Cancellable relativo al *timeout* per l'annullamento dell'invito, e l'oggetto JSON contenente l'invito stesso.

I metodi sono:

- invitabile(): Invitabilita: restituisce un oggetto di tipo Occupato se l'Utente è in attesa di un altro utente oppure se una delle Connessione è già in chiamata oppure ha invitato un utente; altrimenti, se non c'è nessuna Connessione che supporta *WebRTC*, un oggetto di tipo NoRTC; altrimenti, un oggetto di tipo Invitabile;

- invia(msg: JsValue): invia il messaggio msg a tutti i *client* collegati di quell'Utente;

- inviaSoloRtc(msg: JsValue): invia il messaggio msg ai soli *client* di quell'Utente che supportano WebRTC;

- annullaInviti(tranne: Option[Connessione]): invia un messaggio di annullamento dell'invito (precedentemente inviato) a tutti i *client* dell'Utente; se tranne non è None, il messaggio non viene inviato al *client* indicato.

4.4.3.3 *Risorsa*

La classe viene inizializzata con due parametri, nome (l'id della risorsa) e gestore (un attore a cui comunicare che la risorsa non ha più *client* collegati).

Le istanze della classe ricevono i seguenti messaggi dall'esterno (attraverso il metodo receive):

- NuovaConnessione(nome: String, Canale: Concurrent.Channel[String], rtc: Boolean): segnala la connessione di un nuovo *client* alla Risorsa. Provoca l'invio di un messaggio di risposta che contiene la nuova Connessione;

- Rimuovi(conn: Risorsa#Utente#Connessione): segnala la disconnessione di un *client* da una Risorsa;

- Richiedi(conn: Risorsa#Utente#Connessione, s: String): richiede di elaborare la richiesta s proveniente dalla Connessione conn;

- Spegni(): provoca l'interruzione del funzionamento dell'attore.

La classe possiede i seguenti metodi (tutti privati):

- messaggioConnessi(): JsObject: restituisce un oggetto JSON di tipo lista (si veda la sezione 4.2.4 a pagina 25);

- aggiungi(nome: String, can: Concurrent.Channel[String], rtc: Boolean): aggiunge una nuova Connessione all'Utente con nome nome, creandolo se necessario. Invia il messaggio di risposta contenente la nuova Connessione. Invia il nuovo messaggio di tipo lista (si veda la sezione 4.2.4 a pagina 25). Infine, se l'utente della nuova Connessione aveva un invito pendente, lo invia anche al nuovo *client*;

- rimuovi(conn: Utente#Connessione): rimuove una Connessione. Annulla gli eventuali inviti pendenti che la Connessione abbia inviato e la rimuove da un eventuale Circuito a cui fosse connessa. Se il *client* che si sta disconnettendo è l'ultimo che supporta *WebRTC*, rifiuta un eventuale invito pendente per l'Utente della Connessione disconnessa. Invia il nuovo messaggio di tipo lista (si veda la sezione 4.2.4 a pagina 25). Infine, se non c'è più nessuna Connessione, invia un messaggio PossoEssereSpenta al suo gestore;

- gestisciRichiesta(conn: Utente#Connessione, s: String): deserializza il messaggio in JSON e controlla se è ben formato; se lo è invoca uno dei metodi seguenti, altrimenti invia al Logger un messaggio di errore;

- messaggio(conn: Utente#Connessione, dest: String, m: JsValue): invia una notifica msg (si veda la sezione 4.2.4 a pagina 25) a tutti i *client* dell'utente dest;

- circInvita(conn: Utente#Connessione, ndest: String, dett: JsValue): se l'utente ndest non è collegato, oppure è occupato, oppure non utilizza alcun *browser* che supporta *WebRTC*, invia al chiamante un messaggio opportuno (si veda la sezione 4.2.4 a pagina 25); altrimenti, invia al destinatario un invito, preparando al contempo un *timeout* per annullare l'invito in caso di mancata risposta e aggiornando opportunamente i campi invitato ed inAttesa;

- circAccetta(chiamato: Utente#Connessione): se il *client* non era stato invitato produce un messaggio di errore; altrimenti, annulla il *timeout*, annulla gli inviti agli altri eventuali *client* connessi con l'Utente destinatario, connette chiamante e chiamato ad un circuito (quello del chiamante se già esistente, altrimenti uno nuovo), informa

il chiamante ed infine aggiorna opportunamente i campi invitato ed inAttesa;

- circRifiuta(chiamato: Utente#Connessione): se il *client* non era stato invitato produce un messaggio di errore; altrimenti, annulla il *timeout*, annulla gli inviti agli altri eventuali *client* connessi con l'Utente destinatario, informa il chiamante ed infine aggiorna opportunamente i campi invitato ed inAttesa;

- circAbbandona(conn: Utente#Connessione): se il *client* non è connesso ad un Circuito produce un messaggio di errore, altrimenti lo rimuove da quel Circuito;

- circCmd(conn: Utente#Connessione, cmd: String, msg: JsObject): se il *client* non è connesso ad un Circuito produce un messaggio di errore, altrimenti invia il comando al Circuito;

- annullaInvito(chiamato: Utente, info: String): se chiamato è stato invitato, annulla quell'invito, informando chiamante e chiamato e aggiornando opportunamente i campi invitato ed inAttesa;

- broadcast(m: JsValue): invia il messaggio m a tutte le Connessione di tutti gli Utente connessi;

- spegni(): se la Risorsa ha ancora *client* connessi invia al Logger un errore; in ogni caso disattiva l'attore collegato alla Risorsa;

- controllaPong(): invia a tutti i *client* il messaggio ping, poi disconnette d'ufficio tutti i *client* che non inviano un pong da un tempo maggiore di quello indicato nel *file* di configurazione.

4.4.4 *Telepresenza*

L'oggetto Telepresenza ha il compito di ricevere le richieste di creazione di un WebSocket e di inoltrare le richieste provenienti da questo e la sua disconnessione alla corretta Risorsa.

Per gestire l'accesso concorrente alla lista delle Risorsa, possiede una classe interna GestoreRisorse sottoclasse di akka.actor.Actor.

4.4.4.1 *Telepresenza#GestoreRisorse*

GestoreRisorse ha come campo dati mappa, una HashMap mutabile che collega l'id di ogni Risorsa all'istanza corrispondente.

La classe riceve i seguenti messaggi (attraverso il metodo receive):

- getRisorsa(idRisorsa: String, idUtente: String, canale: Concurrent.Channel[String], rtc: Boolean): viene inviato dall'oggetto Telepresenza quando viene richiesto un nuovo *WebSocket*. La classe recupera la Risorsa se già esistente oppure ne crea una nuova e le invia un messaggio NuovaConnessione. Se la Risorsa ha un *timeout* di spegnimento in corso lo interrompe. Quando la Risorsa risponde con la nuova istanza di Connessione, la classe risponde al messaggio inviando la Risorsa e la Connessione;

- PossoEssereSpenta(nome: String): viene inviato da una Risorsa il cui ultimo *Client* si è disconnesso. La classe avvia un timeout di spegnimento (di durata configurabile) al termine del quale invia a se stessa un messaggio di tipo SpegniRisorsa;

- SpegniRisorsa(nome: String): provoca la rimozione dalla mappa della Risorsa e l'invio del messaggio Spegni a quest'ultima.

4.4.4.2 *Telepresenza*

L'oggetto ha un campo di tipo ActorSystem e uno che contiene l'istanza di GestoreRisorse.

Il suo metodo ws (idUtente: String, idRisorsa: String, timestamp: Long, hash: String, rtc: Boolean) viene invocato per creare un nuovo *WebSocket*.

Come prima cosa, ws invoca Validatore.valida; se le credenziali non sono accettate restituisce (null, Enumerator.eof) per chiudere la connessione.

Il metodo ha un valore promise, di tipo Promise, che verrà riempito quando saranno disponibili la Risorsa e la Connesione relative al WebSocket.

L'*output* viene gestito attraverso un Concurrent.Channel, alla cui apertura viene inviato un messaggio getRisorsa al GestoreRisorse. Quando il gestore risponde con una coppia (risorsa, connessione), essa viene inserita in promise.

L'*input* viene gestito attraverso un Iteratee. In esso sono definite due funzioni per la gestione dei messaggi. ck è quella che risponde inizialmente; se promise non è ancora stato riempito, si limita ad ignorare l'input; altrimenti, passa il controllo alla funzione creata attraverso la crealtera. crealtera prende come parametri la Risorsa e la Connessione e restituisce una funzione itera che tratta le richieste e le disconnessioni inviando gli opportuni messaggi alla Risorsa.

4.5 CLIENT

Il prototipo del *client* è stato sviluppato in *JavaScript* utilizzando la libreria già in uso, *Ext JS* 4.1. Le funzionalità minime dell'applicazione esistente (lista di risorse, componenti grafiche di base) sono state implementate tramite dei *mockup*, che non sono qui descritti.

Sono stati progettati ed implementati varie classi della *view* (AltriUtenti, MessaggioIn e MessaggioOut) ed i rispettivi *controller* (AltriUtenti, Messaggi). Ogni risorsa è rappresentata da una istanza della classe Risorsa. Ogni Risorsa ha un riferimento ad un Socket, che gestisce un *WebSocket* e controlla che sia funzionante correttamente; ogni risorsa può inoltre avere un riferimento ad una WebrtcConnection, che gestisce tutti gli aspetti relativi ad una connessione *WebRTC*. La struttura di queste ultime classi è riassunta nella figura 10.

4.5.1 *Risorsa*

Le istanze della classe rappresentano una singola risorsa a cui l'utente si collega; ogni Risorsa ha un nome e un Socket per comunicare col *server*. Inoltre, una Risorsa può avere un riferimento ad una WebrtcConnection.

I metodi di una Risorsa sono:

- *costruttore*(): alla creazione, viene impostato il campo nome con il nome e il campo ws con un nuovo Socket;
- chiudi(): chiude il Socket e la WebrtcConnection;
- creaRtc(): se non esiste già una WebrtcConnection ne crea una e la assegna alla proprietà rtc;
- getPulsanti(dest: String, destRtc: Boolean): String[]: restituisce una lista di operazioni disponibili con l'utente dest. Se *WebRTC* non è disponibile nel *browser*, restituisce

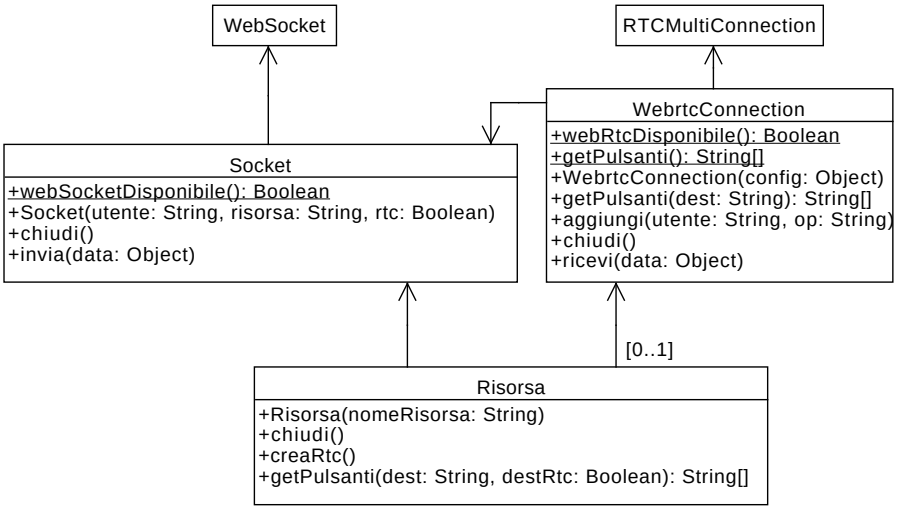

Figura 10: Diagramma delle classi relativo a Risorsa, Socket e WebrtcConnection

['nortc']; se invece non è disponibile per il chiamato, restituisce ['destNortc']; altrimenti, se esiste una WebrtcConnection chiama su di essa il metodo getPulsanti, altrimenti chiama sulla classe WebrtcConnection il metodo statico getPulsanti.

4.5.2 *Socket*

La classe si occupa di gestire un *WebSocket*, rispondendo inoltre ai messaggi ping e ricreando un *WebSocket* quando non riceve alcun messaggio ping per un intervallo superiore ad una certa soglia.

La classe ha un metodo statico:

- webSocketDisponibile(): Boolean: restituisce true se e solo se i *WebSocket* sono disponibili sul *browser* in uso.

Le istanze della classe hanno i seguenti metodi:

- *costruttore*(utente: String, risorsa: String, rtc: Boolean): imposta le proprietà nomeUtente, nomeRisorsa e rtc, poi chiama _creaSocket;

- _creaSocket() (priv.): richiede delle credenziali di autenticazione aggiornate (*hash* e *timestamp*) al *Server Telepresenza*; quando riceve la risposta, crea un nuovo *WebSocket*. Imposta un *timeout* per il controllo del ping in maniera

da rilevare eventuali disconnessioni; se si rileva una disconnessione, dopo alcuni secondi viene chiamato nuovamente _creaSocket e viene generato un evento problemiRete. Quando il *WebSocket* riceve un messaggio, genera un evento, di tipo lista se il messaggio ha tipo lista o di tipo messaggio altrimenti, passando come parametro il messaggio deserializzato;

- _clearTimeoutRicreazione() (priv.): interrompe il *timeout* per la ricreazione del *WebSocket*;
- _clearTimeoutPing() (priv.): interrompe il *timeout* per il controllo del ping;
- _chiudiSocket() (priv.): se esiste un *WebSocket* attivo lo chiude; inoltre interrompe il controllo del ping;
- chiudi(): chiude il Socket, invocando _chiudiSocket e _clearTimeoutRicreazione;
- invia(data: Object): serializza in JSON data e lo invia sul *WebSocket*.

4.5.3 *WebrtcConnection*

La classe gestisce le connessioni *WebRTC*. Si è scelto di utilizzare la libreria RTCMultiConnection che fornisce un'interfaccia indipendente dal *browser* e gestisce autonomamente le chiamate con più di due partecipanti e le aggiunte e rimozioni di partecipanti; oltre a ciò, non richiede di utilizzare un particolare servizio per inviare i segnali richiesti, permettendo un facile adattamento all'architettura fin qui descritta.

La classe ha i seguenti metodi statici:

- webrtcDisponibile(): Boolean: restituisce true se e solo se il *browser* supporta *WebRTC*;
- getPulsanti(): String[]: restituisce una lista di operazioni che sarebbero disponibili in una nuova WebrtcConnection allo stato attuale (le operazioni contemplate sono quelle descritte nel campo pulsantiAV della classe view.MessaggioOut, descritta nella sezione 4.5.4.2 a pagina 41).

Nello *scope* di definizione di WebrtcConnection è definita la variabile inComunicazione che se il *browser* non ha comunicazioni audio o video attive è null, altrimenti è il nome della Risorsa che ha la comunicazione attiva.

Le istanze hanno i seguenti metodi:

- *costruttore*(config: Object): imposta le proprietà di configurazione: il Socket socket e la stringa nomeRisorsa. Inoltre, registra sull'evento problemiRete del socket il proprio metodo chiudi;
- getPulsanti(dest: String): String[]: restituisce la lista delle operazioni ammesse con l'utente dest allo stato attuale (le operazioni contemplate sono quelle descritte nel campo pulsantiAV della classe view.MessaggioOut, descritta nella sezione 4.5.4.2 a pagina 41);
- aggiungi(utente: String, op: String): richiede di interagire con l'utente utente; se op è 'v' si richiede di iniziare una chiamata video con lui; se è 'a' una chiamata audio; se è '+' di aggiungerlo alla chiamata in corso. Prima di inviare l'invito il metodo controlla che non si sia già in comunicazione con lo stesso utente e che non esista già una connessione aperta relativa ad un'altra risorsa. Crea anche una finestra che informa l'utente che è stata inviata la richiesta;
- chiudi(): chiude la comunicazione, se esistente, e chiude eventuali finestre relative ad essa;
- ricevi(data: Object): metodo invocato quando è arrivato un messaggio sul Socket destinato alla WebrtcConnection; un messaggio è destinato ad essa se è il tipo non è msg. Il messaggio viene trattato a seconda del suo tipo:
  1. circ.invito: è un invito a chiamata; viene chiesto all'utente se desidera rispondere. La risposta all'invito è di tipo circ.rifiuta o circ.accetta;
  2. circ.invitoAnnullato: provoca la rimozione della finestra contenente un invito precedente;
  3. circ.occupato: è la notifica che il destinatario ha rifiutato; provoca la chiusura della finestra di attesa e la comparsa di un messaggio di errore esplicativo;
  4. circ.accettato: è la notifica che il destinatario ha accettato; provoca la chiusura della finestra di attesa, la

creazione di una nuova finestra che informa l'utente che la connessione sta per essere aperta, e l'effettiva inizializzazione della connessione;

5. circ.utAbb: è la notifica che un utente ha abbandonato il circuito; provoca l'invocazione del metodo _utenteHaAbbandonato;
6. circ.msg: è un messaggio destinato all'oggetto RTCMultiConnection, a cui viene inoltrato;

- _utenteHaAbbandonato(nome: String) (priv.): rimuove la finestra dell'utente nome e, se necessario, chiude la RTCMultiConnection;
- _newConn(initiator: String, tipo: String) (priv.): crea un nuovo oggetto RTCMultiConnection e lo assegna alla proprietà conn; il metodo deve essere invocato con il nome dell'utente che ha iniziato la chiamata e con il tipo della chiamata (audio o video).

  Dopo aver creato una RTCMultiConnection, il metodo la configura in modo da lavorare correttamente con il resto dell'applicazione. Imposta il campo onstream della RTCMultiConnection ad una funzione che crea una finestra *Ext JS* adatta a mostrare un nuovo *stream* audio o video; la finestra relativa all'utente locale ha una × di chiusura, che provoca la chiusura della connessione e la rimozione di tutte le finestre. Il metodo inoltre imposta i campi onleave e onstreamended ad una funzione che chiama _utenteHaAbbandonato. Infine, imposta la proprietà openSignalingChannel ad una funzione che apre un nuovo sotto-circuito e restituisce un oggetto la cui proprietà send è una funzione che adatta i messaggi provenienti dalla libreria in maniera che possano essere inviati su un Socket e li invia su di esso.
- _chiudiConn() (priv.): provoca la chiusura della RTCMultiConnection; inoltre, avvia un timeout per rimuovere le finestre che non venissero chiuse dalla libreria.

4.5.4 *view*

4.5.4.1 *view.AltriUtenti*

La classe è una semplice sottoclasse di Container che serve a mostrare, all'interno di ogni *tab*, la lista degli altri utenti collegati

Figura 11: La barra superiore come appare nella versione integrata del *Client*; il gruppo di controlli a sinistra è un'istanza della *view* AltriUtenti.

(si veda anche la figura 11).

Le istanze hanno i seguenti metodi:

- setLista(data: Object): modifica la lista visualizzata; quando viene invocato, provoca la cancellazione della lista esistente e l'aggiunta della lista dei nuovi utenti. Il parametro data è un oggetto che contiene i dati degli altri utenti connessi alla stessa risorsa; ha come chiavi gli id utente e come valori dei booleani che indicano se l'utente utilizza *browser* che supportano *WebRTC*. Per ogni utente viene generato un Button che ha come testo i primi 10 caratteri dell'id utente e come tooltip l'id completo; alla proprietà destinatarioMessaggio del Button viene assegnato l'id utente, mentre alla destinatarioRtc viene assegnato il suo booleano sopra descritto;

- noWs(): viene invocato quando i *WebSocket* non sono disponibili nel *browser* e provoca la visualizzazione di un messaggio informativo al posto della lista di utenti connessi;

- problemiRete(): viene invocato quando il *browser* non riesce a connettersi al *Server Telepresenza* a causa di problemi di rete e provoca la visualizzazione di un messaggio informativo al posto della lista di utenti connessi.

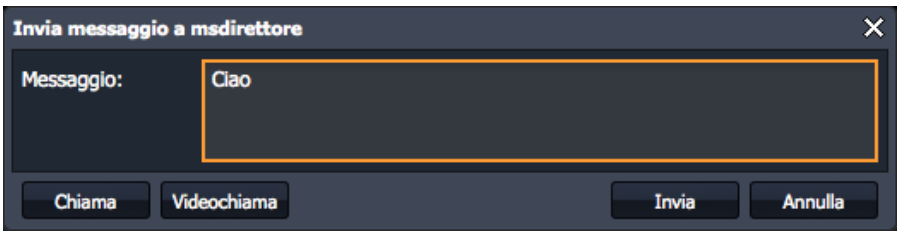

Figura 12: Un'istanza della *view* MessaggioOut come appare nella versione integrata del *Client*.

4.5.4.2 *view.MessaggioOut*

La classe è sottoclasse di Window ed include un campo in cui l'utente può inserire il messaggio per un altro utente (si veda anche la figura 12). Deve essere configurata passandole alcuni campi:

- nome: il nome del destinatario,
- risorsa: l'oggetto Risorsa relativo,
- pulsantiAV: un array di stringhe che rappresentano operazioni o messaggi. Le operazioni possibili sono:
  - 'v': chiamata video,
  - 'a': chiamata audio,
  - '+': aggiunta alla chiamata in corso.

  I messaggi disponibili sono:
  - 'nortc': informa l'utente che *WebRTC* non è disponibile nel *browser*,
  - 'destNortc': informa l'utente che *WebRTC* non è disponibile per il destinatario,
  - 'gia': informa l'utente che ha già una chiamata attiva relativa ad un'altra risorsa.

4.5.4.3 *view.MessaggioIn*

La classe è sottoclasse di Window ed include un pannello per mostrare il messaggio arrivato da un altro utente e un campo in cui è possibile scrivere una risposta. Deve essere configurata passandole alcuni campi:

- nome: il nome del mittente,
- messaggio: il testo da mostrare,
- risorsa: la Risorsa relativa.

4.5.5 *controller*

4.5.5.1 *controller.ListaRisorse*

La classe è un mockup che rileva il click su una risorsa e apre un *tab* dedicato nell'area centrale (MainTabs). Quando apre un *tab*, imposta su di esso la proprietà infoBaseCompito.compito al nome della risorsa che quel *tab* utilizza.

4.5.5.2 *controller.AltriUtenti*

La classe funge da *controller* per la *view* omonima. Viene informata quando una vista di tipo AltriUtenti viene creata o distrutta.

Quando una vista AltriUtenti viene creata (cioè quando un *tab* viene aperto), se i *WebSocket* non sono disponibili, chiama il metodo noWs sulla *view*. Altrimenti, recupera dal *tab* il nome della risorsa, crea un oggetto Risorsa e lo assegna alla proprietà risorsa della *view*. Si registra sul Socket della Risorsa appena creata per ricevere le liste di utenti collegati e per essere informato di problemi di rete; quando riceve una lista, rimuove dalla lista l'utente locale e passa gli elementi rimanenti al metodo setLista della *view*, mentre quando ci sono problemi di rete invoca su di essa il metodo problemiRete. Infine, genera su di sé un evento di tipo socketPronto.

Quando una vista AltriUtenti viene distrutta (cioè quando un *tab* viene chiuso), invoca il metodo chiudi sulla Risorsa salvata nella vista.

4.5.5.3 *controller.Messaggi*

La classe funge da *controller* per le *view* MessaggioIn e MessaggioOut.

Quando una *view* di tipo AltriUtenti emette un evento socketPronto, si registra sul Socket della Risorsa per ricevere i messaggi; quando riceve un messaggio di testo (con tipo msg) crea una finestra di tipo MessaggioIn per mostrare il messaggio; altrimenti, si assicura che la Risorsa abbia una WebrtcConnection e passa a quest'ultima il messaggio.

Quando l'utente preme il tasto *Invia* o *Rispondi* in una finestra, invia il messaggio inserito sul Socket della Risorsa della finestra.

Quando viene premuto il tasto di un utente nella *view* AltriUtenti, apre una *view* MessaggioOut dopo aver richiesto alla Risorsa la lista delle operazioni per quell'utente.

Infine, quando viene premuto uno dei tasti per avviare una chiamata, si assicura che la Risorsa abbia una WebrtcConnection e invoca su di essa il metodo aggiungi.

5 INTEGRAZIONE DEL PROTOTIPO

5.1 SERVER

La parte *Scala* del prototipo costituisce il *Server Telepresenza*; è stata progettata e sviluppata utilizzando la giusta versione di *Play*, quindi non ha richiesto operazioni aggiuntive.

La parte *Java* del prototipo è invece destinata al *Server d>PLUS*; è stata sviluppata in una versione di *Play* diversa da quella a cui è destinata, quindi ha richiesto alcuni adattamenti.

Le classi TimestampHash e GeneratoreHash non hanno dipendenze esterne, quindi sono rimaste immutate, ad eccezione del cambio di *package* da controllers a controllers.webrtc. Auth, invece, dialoga direttamente col *framework*; la differenza tra le due interfacce ha richiesto il cambiamento della firma del metodo da public static Result auth(String utente, String risorsa) a public static void auth(). Nella versione adattata i parametri vengono recuperati attraverso il campo params e l'*output* viene effettuato attraverso renderJSON.

5.2 CLIENT

Le classi del *Client* sono state sviluppate per poter essere inserite facilmente; per questo motivo hanno richiesto solo la modifica dei nomi per adeguarsi alla struttura gerarchica esistente. Le classi della *view* sono state inserite nel *package* APLUS.view.presence, mentre le altre classi sono state inserite in APLUS.controller.presence.

È stato incluso un riferimento alla libreria RTCMultiConnection nel *file* `index.html`; inoltre la libreria *Ext JS* è stata informata dei due nuovi *controller* (Messaggi ed AltriUtenti) inserendo i loro nomi nel *file* `app.js`.

L'inclusione della *view* AltriUtenti nella *toolbar* di EditorArticoloECommerce si è rivelata l'operazione più complessa. La *toolbar* è definita nella superclasse di EditorArticoloECommerce, Compito, sotto forma di operazione di inserimento all'interno di un metodo più ampio. Quando si vorrà applicare l'approccio sviluppato nel corso di questo progetto a tutti i compiti previsti nell'applicazione, la classe Compito sarà il luogo natu-

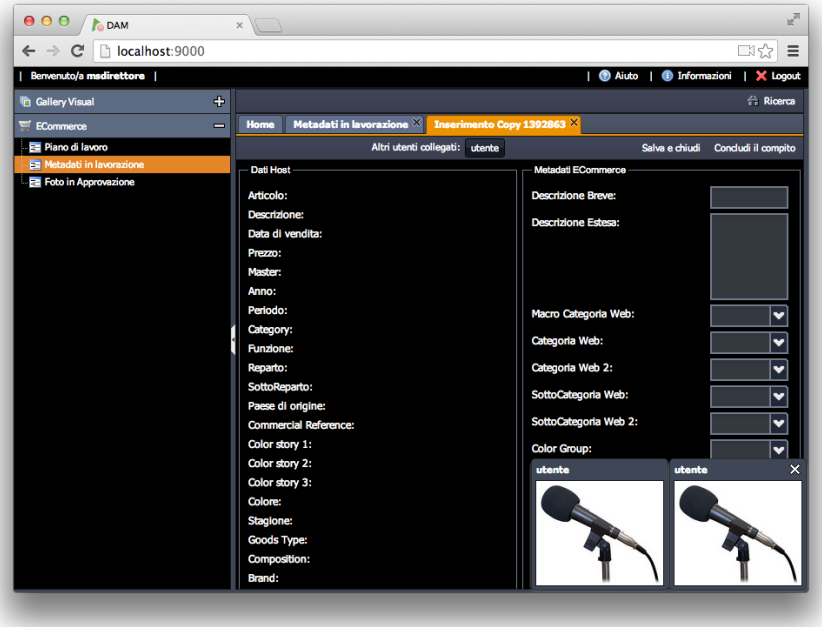

Figura 13: Interfaccia integrata, con una conversazione audio in atto

rale in cui inserire questa operazione; tuttavia, il progetto sperimentale prevede l'inserimento nella sola EditorArticoloECommerce. Risulta quindi molto difficile implementare l'aggiunta della componente grafica alla *toolbar* come ridefinizione di un metodo nella definizione della classe EditorArticoloECommerce, mentre si è preferito non modificare Compito per non alterare il comportamento delle altre sottoclassi, non coinvolte nel progetto.

L'inserimento della componente AltriUtenti è stato quindi delegato al *controller* AltriUtenti, che reagisce alla visualizzazione di una EditorArticoloECommerce inserendo nella sua *toolbar* la componente. Questa soluzione, pur teoricamente sconsigliabile, è stata scelta alla luce delle considerazioni sopra esposte.

L'interfaccia finale ottenuta è visibile nella figura 13.

6 TEST

Questo capitolo espone i *test* che sono stati eseguiti sulle due componenti *server* e sul sistema nel complesso.

Il codice identificativo di un *test* è formato dal prefisso R: e dalla concatenazione dei seguenti elementi:

1. un indicatore della componente sottoposta al *test*, tratto dai seguenti:
   - D per i *test* sulle classi aggiuntive del *Server d>PLUS*;
   - T per i *test* sul *Server Telepresenza*;
   - S per i *test* effettuati sull'intero sistema.
2. un numero univoco.

Un'eventuale *precondizione* indica qual è lo stato del sistema prima dell'inizio del *test*; la *descrizione* indica quali sono le operazioni da eseguire e qual è il comportamento atteso; l'*esito* indica se il *test* è stato superato; infine, per quei *test* che verificano uno o più requisiti, la lista di questi è indicata alla voce *verifica*.

6.1 DESCRIZIONE DEI TEST

6.1.1 *Test delle classi aggiuntive del Server d>PLUS*

Il *test* è stato effettuato prima sulla parte *Java* del prototipo e successivamente sul *Server d>PLUS* integrato.

6.1.1.1 *T:D1 – Test di Auth*

DESCRIZIONE

1. Usando un *browser*, si visita la pagina

   `http://⟨server⟩/auth?risorsa=nomerisorsa&utente=nomeutente`

2. Si deve visualizzare una stringa del tipo

   `{"timestamp":⟨timestamp⟩,"hash":"⟨hash⟩"}`

3. Si deve ottenere ⟨hash⟩ calcolando lo *hash* della stringa

```
nomeutente$nomerisorsa$⟨timestamp⟩
$⟨segreto⟩
```

ESITO superato

6.1.2 *Test del Server Telepresenza*

I *test* sono stati effettuati sulla parte *Scala* del prototipo, che corrisponde al *Server Telepresenza*.

Per creare le connessioni *WebSocket* è stata utilizzata la *utility* a linea di comando `wscat`, che permette di connettersi ad un *WebSocket* e di interagire con esso direttamente attraverso il terminale; è inoltre stato usato uno *script bash*, `ws.sh`, che prende come parametri i nomi dell'utente e della risorsa, genera *timestamp* ed *hash* e si collega al *WebSocket*; ws.sh dichiara sempre di supportare *WebRTC*.

Per agevolare i *test*, durante il loro svolgimento il *timeout* dovuto alla mancata risposta ad un messaggio ping è stato aumentato.

6.1.2.1 *T:T1 – Test di connessione riuscita*

DESCRIZIONE

1. Usando lo *script*, ci si collega con l'utente `u1` alla risorsa `r1`.
2. Si deve ricevere il messaggio

   ```
   {"tipo":"lista","lista":{"u1":true}}
   ```

3. Si deve ricevere una serie di messaggi ping fino a quando ci si disconnette.

ESITO superato

6.1.2.2 *T:T2 – Test del timeout*

DESCRIZIONE

1. Solo per questo *test*, si imposta il *timeout* dovuto alla mancata risposta al ping al valore ordinario.
2. Usando lo *script*, ci si collega con l'utente `u1` alla risorsa `r1`.
3. Si attende lo scadere del *timeout*.

4. Il *WebSocket* deve chiudersi.

ESITO superato

6.1.2.3 *T:T3 – Test di connessione con credenziali errate*

DESCRIZIONE Usando direttamente `wscat`, ci si collega con uno *hash* sbagliato. Il *WebSocket* deve chiudersi immediatamente.

VERIFICA R:VM10

ESITO superato

6.1.2.4 *T:T4 – Test di connessione con credenziali scadute*

DESCRIZIONE Usando direttamente `wscat`, ci si collega con uno *hash* corretto ma scaduto. Il *WebSocket* deve chiudersi immediatamente.

VERIFICA R:VM10

ESITO superato

6.1.2.5 *T:T5 – Test delle liste utenti*

DESCRIZIONE

1. Usando lo *script*, ci si collega con l'utente `u1` alla risorsa `r1`.
2. `u1` deve ricevere il messaggio

   ```
   {"tipo":"lista","lista":{"u1":true}}
   ```

3. Usando lo *script*, ci si collega con l'utente `u2` alla risorsa `r1`.
4. `u1` e `u2` devono ricevere il messaggio

   ```
   {"tipo":"lista","lista":{"u2":true,
   "u1":true}}
   ```

5. `u1` si disconnette.
6. `u2` deve ricevere il messaggio

   ```
   {"tipo":"lista","lista":{"u2":true}}
   ```

ESITO superato

6.1.2.6 *T:T6 – Test accettazione invito*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa r1.

DESCRIZIONE

1. u1 invia un invito:

   ```
   {"tipo":"circ.invita","dest":"u2",
   "dett":{"q":42}}
   ```

2. u2 deve ricevere

   ```
   {"tipo":"circ.invito","mitt":"u1",
   "dett":{"q":42}}
   ```

3. u2 risponde

   ```
   {"tipo":"circ.accetta"}
   ```

4. u1 deve ricevere

   ```
   {"tipo":"circ.accettato",
   "mitt":"u2"}
   ```

ESITO superato

6.1.2.7 *T:T7 – Test rifiuto invito*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa r1.

DESCRIZIONE

1. u1 invia un invito:

   ```
   {"tipo":"circ.invita","dest":"u2",
   "dett":{"q":42}}
   ```

2. u2 risponde

   ```
   {"tipo":"circ.rifiuta"}
   ```

3. u1 deve ricevere

   ```
   {"tipo":"circ.occupato",
   "mitt":"u2","info":"rifiutato"}
   ```

ESITO superato

6.1.2.8 *T:T8 – Test timeout invito*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa r1.

DESCRIZIONE

1. u1 invia un invito:

   ```
   {"tipo":"circ.invita","dest":"u2",
   "dett":{"q":42}}
   ```

2. Allo scadere del *timeout* dell'invito, u1 deve ricevere

   ```
   {"tipo":"circ.occupato","mitt":"u2",
   "info":"scaduto"}
   ```

ESITO superato

6.1.2.9 *T:T9 – Test del sotto-circuito*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa r1 e sono connessi allo stesso circuito (come alla fine del T:T6).

DESCRIZIONE

1. u1 si iscrive ad un sotto-circuito con

   ```
   {"tipo":"circ.cmd.addSub","sub":"sub"}
   ```

2. u1 invia

   ```
   {"tipo":"circ.cmd.msg","sub":"sub",
   "data":{"q":42}}
   ```

3. u2 non deve ricevere alcun messaggio (fatta eccezione per i ping).
4. u2 si iscrive ad un sotto-circuito con il messaggio usato al punto 1.
5. u1 invia nuovamente il messaggio inviato al punto 2.
6. u2 deve ricevere

   ```
   {"tipo":"circ.msg","mitt":"u1",
   "sub":"sub","data":{"q":42}}
   ```

7. u1 esce dal circuito con

   ```
   {"tipo":"circ.abbandona"}
   ```

8. u2 deve ricevere

   ```
   {"tipo":"circ.utAbb","mitt":"u1"}
   ```

9. u1 si iscrive ad un altro sotto-circuito con

```
{"tipo":"circ.cmd.addSub","sub":"sub2"}
```

10. u2 invia il messaggio inviato al punto 2.
11. u1 non deve ricevere alcun messaggio (fatta eccezione per i ping).

ESITO superato

6.1.3 *Test di sistema*

Tutti i *test* sono stati svolti sul prototipo; i *test* che è possibile eseguire con due soli utenti sono stati applicati anche al sistema integrato. I *test* sono stati eseguiti in *Chrome*, salvo indicazioni diverse.

Nel prototipo è possibile selezionare l'utente con cui ci si collega aggiungendo la stringa `#utente=⟨utente⟩` all'URL.

6.1.3.1 *T:S1 – Connessione e lista utenti*

DESCRIZIONE

1. u1 si connette alla risorsa risorsa1.
2. ad u1 non deve risultare alcun utente connesso.
3. u2 si connette alla risorsa risorsa1.
4. ad u1 deve risultare u2 connesso.

ESITO superato

VERIFICA R:FM1, R:FM3, R:FM3.1

6.1.3.2 *T:S2 – Connessione multipla*

DESCRIZIONE

1. u1 si connette alla risorsa risorsa1 da più di una postazione.
2. u2 si connette alla risorsa risorsa1 da più di una postazione.
3. Ogni *browser* di u1 deve mostrare una sola volta u2.
4. Ogni *browser* di u2 deve mostrare una sola volta u1.

ESITO superato

VERIFICA R:FM1, R:FC1.1, R:FM3

6.1.3.3 *T:S3 – Disconnessione ordinaria*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1.

DESCRIZIONE

1. u2 chiude il *tab* (inteso come l'elemento grafico all'interno dell'interfaccia) di risorsa1.
2. Ad u1 non deve più risultare u2.

ESITO superato

VERIFICA R:FM2, R:FM3, R:FM3.2

6.1.3.4 *T:S4 – Disconnessione per chiusura della pagina*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1.

DESCRIZIONE

1. u2 chiude il *tab* del *browser* che contiene la pagina.
2. Ad u1 non deve più risultare u2.

ESITO superato

VERIFICA R:FM2, R:FM3, R:FM3.2

6.1.3.5 *T:S5 – Invio messaggio di testo*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1, u2 da più di una postazione.

DESCRIZIONE

1. u1 invia un messaggio di testo non vuoto a u2.
2. Ogni postazione di u2 deve visualizzare il messaggio.

ESITO superato

VERIFICA R:FM4, R:FC4.1

6.1.3.6 *T:S6 – Invito a chiamata e conversazione audio*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1, u2 da più di una postazione.

DESCRIZIONE

1. u1 invia un invito a chiamata audio a u2.
2. Ogni postazione di u2 deve visualizzare l'invito.
3. Una postazione di u2 accetta l'invito.
4. Tutte le altre postazioni di u2 non devono più mostrare l'invito.
5. Si accettano eventuali messaggi di autorizzazione dei *browser*.
6. Deve iniziare la conversazione.

ESITO superato

VERIFICA R:FS5, R:FS5.1, R:FC5.3, R:FC5.3.1, R:FS6, R:FS6.1

6.1.3.7 *T:S7 – Rifiuto invito*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1.

DESCRIZIONE

1. u1 invia un invito a chiamata audio a u2.
2. Ogni postazione di u2 deve visualizzare l'invito.
3. Una postazione di u2 rifiuta l'invito.
4. Tutte le postazioni di u2 non devono più mostrare l'invito.
5. u1 deve visualizzare un messaggio informativo di rifiuto.

ESITO superato

VERIFICA R:FS5, R:FS5.2, R:FC5.3, R:FC5.3.1

6.1.3.8 *T:S8 – Timeout invito*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1.

DESCRIZIONE

1. u1 invia un invito a chiamata audio a u2.
2. Scaduto il *timeout* di risposta, tutte le postazioni di u2 non devono più mostrare l'invito.
3. u1 deve visualizzare un messaggio informativo.

ESITO superato

VERIFICA R:FS5, R:FC5.3.1, R:FS5.4

6.1.3.9 *T:S9 – Connessione con invito pendente*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1.

DESCRIZIONE

1. u1 invia un invito a chiamata audio a u2.
2. Una nuova postazione si connette a risorsa1.
3. La nuova postazione deve visualizzare l'invito.

ESITO superato

VERIFICA R:FS5, R:FC5.3.2

6.1.3.10 *T:S10 – Disconnessione con invito pendente*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1.

DESCRIZIONE

1. u1 invia un invito a chiamata audio a u2.
2. u2 si disconnette.
3. u1 deve visualizzare un messaggio informativo di rifiuto.

ESITO superato

VERIFICA R:FS5, R:FC5.3.3

6.1.3.11 *T:S11 – Invito di utente con invito pendente*

PRECONDIZIONE u1, u2 e u3 sono connessi alla risorsa risorsa1. u2 ha un invito pendente da u1.

DESCRIZIONE

1. u3 invia un invito a chiamata audio a u2.
2. u3 deve visualizzare un messaggio informativo di occupato.
3. u2 non deve visualizzare alcun avviso proveniente da u3.

ESITO superato

VERIFICA R:FS5, R:FS5.5

6.1.3.12 *T:S12 – Invito di utente occupato*

PRECONDIZIONE u1, u2 e u3 sono connessi alla risorsa risorsa1. u2 ha una conversazione in atto con u3.

DESCRIZIONE

1. u3 invia un invito a chiamata audio a u2.
2. u3 deve visualizzare un messaggio informativo di occupato.
3. u2 non deve visualizzare alcun avviso.

ESITO superato

VERIFICA R:FS5, R:FS5.5

6.1.3.13 *T:S13 – Invito di utente senza WebRTC*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1. u2 utilizza un *browser* che non supporta *WebRTC*.

DESCRIZIONE

1. u1 apre la finestra per chiamare u2.
2. u1 deve visualizzare un messaggio informativo al posto dei pulsanti di chiamata.

ESITO superato

VERIFICA R:FS5, R:FS5.6

6.1.3.14 *T:S14 – Tentativo di chiamata senza WebRTC*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1. u1 utilizza un *browser* che non supporta *WebRTC*.

DESCRIZIONE

1. u1 apre la finestra per chiamare u2.
2. u1 deve visualizzare un messaggio informativo al posto dei pulsanti di chiamata.

ESITO superato

VERIFICA R:FS5, R:VM11

6.1.3.15 *T:S15 – Video-chiamata*

PRECONDIZIONE u1 e u2 sono connessi alla risorsa risorsa1.

DESCRIZIONE

1. u1 invia un invito a video-chiamata a u2.
2. u2 accetta.
3. Si accettano eventuali messaggi di autorizzazione dei *browser*.
4. La conversazione audio-video deve essere possibile.
5. u2 chiude la chiamata.
6. Le finestre di chiamata devono essere non più visibili in entrambi i *browser*.

ESITO superato

VERIFICA R:FS5, R:FS5.1, R:FS6, R:FS6.2, R:FS6.3

6.1.3.16 *T:S16 – Chiamata multipla*

PRECONDIZIONE u1, u2 e u3 sono connessi alla risorsa risorsa1. u1 ha una conversazione in atto con u2.

DESCRIZIONE

1. u1 invia un invito a chiamata a u3.
2. u3 accetta.
3. Si accettano eventuali messaggi di autorizzazione dei *browser*.

4. La conversazione a tre deve essere possibile.
5. u1 chiude la chiamata.
6. La conversazione tra u2 e u3 deve essere possibile.
7. u2 chiude la chiamata.
8. Le finestre di chiamata devono essere non più visibili tutti i *browser*.

ESITO superato

VERIFICA R:FS5, R:FS5.1, R:FS6, R:FS6.3, R:FC7

6.1.3.17 *T:S17 – Mancanza di WebSocket*

DESCRIZIONE

1. u1 cerca di collegarsi ad una risorsa utilizzando un *browser* privo di *WebSocket*.
2. u1 deve visualizzare un messaggio informativo al posto della lista di utenti.

ESITO superato

VERIFICA R:VM11

6.2 TRACCIAMENTO INVERSO

In questa sezione è riportato il tracciamento tra i requisiti ed i test che verificano ognuno di essi.

Requisito	*Test*
R:FM1	T:S1, T:S2
R:FC1.1	T:S2
R:FM2	T:S3, T:S4
R:FM3	T:S1, T:S2, T:S3, T:S4
R:FM3.1	T:S1
R:FM3.2	T:S3, T:S4
R:FM4	T:S5
R:FC4.1	T:S5

Continua nella prossima pagina

Requisito	*Test*
R:FS5	T:S6, T:S7, T:S8, T:S9, T:S10, T:S11, T:S12, T:S13, T:S14, T:S15, T:S16
R:FS5.1	T:S6, T:S15, T:S16
R:FS5.2	T:S7
R:FC5.3	T:S6, T:S7
R:FC5.3.1	T:S6, T:S7, T:S8
R:FC5.3.2	T:S9
R:FC5.3.3	T:S10
R:FS5.4	T:S8
R:FS5.5	T:S11, T:S12
R:FS5.6	T:S13
R:FS6	T:S6, T:S15, T:S16
R:FS6.1	T:S6
R:FS6.2	T:S15
R:FS6.3	T:S15, T:S16
R:FC7	T:S16
R:VM10	T:T3, T:T4
R:VM11	T:S14, T:S17

Tabella 4: Tracciamento inverso requisiti – *test*

7 CONCLUSIONI

7.1 PRODOTTO FINALE

Alla fine del progetto di *stage*, il prodotto integrato ha soddisfatto tutti i requisiti individuati durante l'analisi; questo risultato è dovuto sia all'assenza di problemi insormontabili durante la progettazione e l'implementazione sia all'esperienza precedente con le tecnologie coinvolte che ha permesso di evitare di porre in essere requisiti non realistici.

Il prodotto finale è quasi pronto per un eventuale utilizzo in produzione; l'unica parte da terminare è l'inclusione del codice che verifica l'effettiva identità dell'utente nella classe Auth. A questo si potrebbe aggiungere uno studio relativo alla migliore interfaccia grafica da utilizzare per la lista degli utenti connessi.

Il progetto ha dimostrato all'azienda che questo livello di funzionalità è raggiungibile utilizzando solo librerie *open source* e codice sviluppato in qualche settimana, senza la necessità di coinvolgere *software* ed infrastrutture a pagamento.

7.2 ANALISI DELLE CRITICITÀ

Uno dei punti più critici del lavoro è stata la progettazione delle classi Utente e Circuito (la soluzione adottata è esposta nella sezione 4.4 a pagina 27). In particolare, il problema più complesso è la gestione dello stato degli utenti e delle connessioni; dati come l'aver inviato un invito oppure la connessione ad un circuito appartengono logicamente ad una singola connessione, mentre dati come aver ricevuto un invito appartengono ad un utente.

All'inizio avevo cercato di unificare le classi Utente e Connessione per semplificare alcuni punti del codice, ma questo portava ad annidamento eccessivo in altri punti, oltre ad essere una modellazione non accurata.

In seguito avevo tentato una soluzione ispirata al *design pattern State*, che si è rivelata però eccessivamente complessa a causa della logica di *business* coinvolta e perché era necessario modificare solo una parte dello stato, senza sostituirlo interamente. L'aspetto peggiore era però che il cambio dello stato di una enti-

tà spesso richiede il cambio dello stato anche di altre entità; ad esempio, quando una Connessione risponde ad un invito, passa da disconnessa a connessa; l'Utente relativo passa da invitato ad occupato, mentre la Connessione che ha emesso l'invito passa da disconnessa a connessa, perdendo nel contempo uno degli inviti emessi ancora pendenti. Utilizzando il *pattern* era difficile garantire la consistenza.

La soluzione scelta è vicina allo stile procedurale. Connessione e Utente hanno dei campi per i vari dati del rispettivo stato. Alcuni metodi di Risorsa eseguono le varie operazioni, come la risposta ad un invito da parte di una connessione, consultando e modificando opportunamente i campi delle istanze di Connessione e Utente. Per evitare la duplicazione del codice sono stati aggiunti metodi di utilità a Connessione e Utente, ad esempio per inviare un messaggio di conferma sul *WebSocket*. Questo approccio, anche se non esattamente *object-oriented*, ha portato ad una soluzione chiara, che mantiene la consistenza facilmente ed è relativamente sintetica (il più lungo dei metodi nominati raggiunge una ventina di righe).

7.3 STRUMENTI E CONOSCENZE ACQUISITE

Molte tecnologie utilizzate nel corso del progetto, come *WebSocket*, *WebRTC* e *Java*, mi erano precedentemente note grazie al progetto di *Ingegneria del Software*; quel progetto mi ha inoltre fornito la *forma mentis* necessaria ad acquisire efficacemente familiarità con nuovi strumenti, come, in questo caso, *Scala*, *Play* e *Ext JS*.

Comprendere il modo di ragionare di *Scala* è stato complicato dal suo approccio funzionale, che viene trattato in maniera molto ridotta durante il corso di laurea. Ne è valsa però la pena: *Scala* è forse il linguaggio più elegante tra quelli che mi sono noti al momento; il suo alto livello permette di evitare di considerare aspetti di cui dovrebbe occuparsi il compilatore.

Play è un *framework event-driven* che è progettato in maniera moderna utilizzando tecnologie attuali (tra cui anche *Scala*); uno *stage* che voglia essere una introduzione al mondo del lavoro attuale e futuro dovrebbe utilizzare tecnologie innovative come questa.

Ext JS è uno dei *framework* più utilizzati per lo sviluppo di interfacce *web*; è ben documentato ed adatto a numerosi compiti. L'esperienza maturata con questa tecnologia sarà sicuramente utile in futuro.

7.4 CONSIDERAZIONI SULLO STAGE

L'esperienza generale dello *stage* è stata certamente molto positiva. Si è trattato infatti di una prima introduzione al mondo del lavoro che mi ha permesso sia di sperimentare il funzionamento interno di una azienda, al di fuori del contesto universitario, sia di imparare nuovi utili strumenti.

Ritengo che l'esperienza sarebbe stata ancora più utile e stimolante qualora lo *stage* fosse stato svolto in *team*; la collaborazione è infatti indispensabile nello sviluppo di progetti complessi in tempi accettabili.

Concludendo, lo *stage* è un'ottima ed indispensabile conclusione del corso di laurea, sia per mettere in pratica le conoscenze apprese durante il corso al di fuori dell'ambiente didattico, sia per toccare da vicino la mentalità del mondo del lavoro.

GLOSSARIO

BETA	caratterizza la versione di un *software* che è ancora sperimentale e non è ancora pronta per l'adozione di massa. 5
BYTECODE JAVA	codice binario che contiene le istruzioni per una JVM. È il risultato della compilazione di un sorgente *Java*. 7, 65
FRAMEWORK	struttura logica di supporto su cui un *software* può essere progettato e realizzato, spesso facilitandone lo sviluppo. Un *framework* spesso include una serie di librerie di codice utilizzabili con uno o più linguaggi di programmazione ed una serie di strumenti di supporto allo sviluppo del *software*. 6, 7
JAVASCRIPT	linguaggio di *scripting* interpretato orientato agli oggetti comunemente usato nei siti *web*. 65
METODO MoSCoW	metodo per indicare la rilevanza di un requisito nell'ambito dell'analisi dei requisiti. Prevede quattro livelli, **M** per *must have*, **S** per *should have*, **C** per *could have* e **W** per *would have*, in ordine decrescente di importanza. 15
POSTAZIONE	nel contesto del progetto, uno dei *browser* da cui l'utente si collega al sistema. 12, 13, 15–17, 27, 50–53

RELEASE — caratterizza la versione di un *software* che è stata ampiamente testata ed è pronta per l'adozione di massa. 5

SEGRETO CONDIVISO — una certa quantità di informazioni che è nota esclusivamente alle entità coinvolte in una comunicazione sicura. 21, 23, 64

THREAD — una porzione di un processo; ogni *thread* viene eseguito concorrentemente agli altri thread, sfruttando processori muniti di più di un *core*. 30

TOKEN DI AUTENTICAZIONE — una certa quantità di informazioni che un *client* possiede ed utilizza in un qualche modo per provare ad un *server* la propria identità. Nella maggior parte dei casi utilizza un segreto condiviso oppure sfrutta la crittografia a chiave pubblica. 21, 27

WIDGET — nel contesto dello sviluppo di interfacce utente, una componente riutilizzabile che è possibile inserire in una interfaccia, come ad esempio un campo di inserimento testo, un pulsante o una tabella. 6

ACRONIMI

API *Application Programming Interface*: la descrizione di come alcune componenti *software* dovrebbero interagire tra loro. Normalmente questa descrizione prende la forma di un insieme di chiamate che una componente può eseguire su un'altra. 5, 6

JSON *JavaScript Object Notation*: formato per lo scambio di dati in applicazioni *client-server*. Utilizza la stessa notazione di *JavaScript*. 6, 23, 24, 30–32, 37

JVM *Java Virtual Machine*: è il componente della piattaforma *Java* che esegue i programmi tradotti in *bytecode Java* dopo una prima compilazione. 7, 63

MVC *Model View Controller*: è un pattern architetturale molto diffuso nello sviluppo di sistemi *software object-oriented* in grado di separare la logica di presentazione dei dati dalla logica di *business*. 6

P2P *Peer to Peer*: tipo di *network* decentralizzato in cui ogni nodo si connette a nodi suoi pari; si differenzia dal modello *client-server* in cui tutti i nodi *client* si connettono ad un nodo *server*. 5

URL *Uniform Resource Locator*: è una sequenza di caratteri che identifica univocamente l'indirizzo di una risorsa in *Internet* rendendola accessibile ad un *client*. 22, 50

W3C *The World Wide Web Consortium*: organizzazione non governativa internazionale che ha come scopo lo sviluppo di tutte le potenzialità del *World Wide Web*. 5, 6

XML *eXtensible Markup Language*: linguaggio di *markup* generico e con un buon supporto per le lingue diverse dall'inglese, pensato per essere sia *human-readable* sia *machine-readable*, cioè leggibile facilmente sia dall'uomo sia da un *software*. Molti altri formati (come XHTML e RSS) sono basati su XML. 6

BIBLIOGRAFIA

RISORSE SU SCALA

- Martin Odersky, Lex Spoon, Bill Venners, *Programming in Scala*, Aritma, seconda edizione, 2010. Disponibile anche all'indirizzo http://www.artima.com/pins1ed
- *Scala API*, http://www.scala-lang.org/api/current/index.html
- Martin Odersky, *Scala By Example*, http://www.scala-lang.org/docu/files/ScalaByExample.pdf
- Michel Schinz, Philipp Haller, *A Scala Tutorial for Java Programmers*, http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html
- *Akka Actors API*, http://doc.akka.io/api/akka/2.2.1/
- *Actors – Akka Documentation*, http://doc.akka.io/docs/akka/snapshot/scala/actors.html
- Alvin Alexander, *An Akka actors 'ask' example – ask, future, await, timeout, duration, and all that*, http://alvinalexander.com/scala/scala-akka-actors-ask-examples-future-await-timeout-result
- Daniel Westheide, *Promises and Futures in practice*, http://danielwestheide.com/blog/2013/01/16/the-neophytes-guide-to-scala-part-9-promises-and-futures-in-practice.html

RISORSE SU PLAY FRAMEWORK

- Peter Hilton, Erik Bakker, Francisco Canedo, *Play for Scala*, Manning, *early access* dal 2012
- *Play API*, http://www.playframework.com/documentation/api/2.1.1/scala/index.html
- *WebSockets*, http://www.playframework.com/documentation/2.1.1/ScalaWebSockets

- *Routing*, http://www.playframework.com/documentation/2.1.1/ScalaRouting
- JSON, http://www.playframework.com/documentation/2.1.1/ScalaJson
- Pascal Voitot, *Understanding Play2 Iteratees for Normal Humans*, http://mandubian.com/2012/08/27/understanding-play2-iteratees-for-normal-humans/

RISORSE SU EXT JS

- *Ext JS 4.1.0 – Sencha Docs*, http://docs.sencha.com/extjs/4.1.0/
- *MVC Application Architecture – Sencha Docs*, http://docs.sencha.com/extjs/4.1.0/#!/guide/application_architecture

RISORSE SU WEBSOCKET

- Malte Ubl, Eiji Kitamura, *Introducing WebSockets: Bringing Sockets to the Web – HTML5 Rocks*, http://www.html5rocks.com/en/tutorials/websockets/basics/
- *Can I use Web Sockets?*, http://caniuse.com/websockets
- Einar Otto Stangvik, *wscat*, *utility* a linea di comando per lavorare con i *WebSocket*, https://github.com/einaros/ws

RISORSE SU WEBRTC

- Sam Dutton, *Getting Started with WebRTC – HTML5 Rocks*, http://www.html5rocks.com/en/tutorials/webrtc/basics/
- Muaz Khan, Libreria *RTCMultiConnection*, https://github.com/muaz-khan/WebRTC-Experiment/tree/master/RTCMultiConnection

- Dean Bubley, *WebRTC forecasts upgraded: mobile support accelerating*, http://webrtcstats.com/webrtc-forecasts-upgraded-mobile-support-accelerating/
- *Phono*, produttore commerciale di *softare* per la telefonia *web*, http://phono.com/webrtc
- *Sharefest*, piattaforma di scambio *file* P2P via *web*, https://www.sharefest.me/
- Tsahi Levent-Levi, *WebRTC Doesn't Fit iOS – or Does it?*, http://bloggeek.me/webrtc-fit-ios/