ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

Sviluppo di applicazioni real-time multi-utente: un middleware basato sulla piattaforma Croquet

Elaborato in
Sistemi Embedded E Internet-of-things

Relatore
Prof. ALESSANDRO RICCI

Presentata da
ALESSANDRO MAZZOLI

Corelatore
Dott. SAMUELE BURATTINI

Anno Accademico 2021 – 2022

"Non andartene docile in quella buona notte,
I vecchi dovrebbero bruciare e delirare quando cade il giorno;
Infuria, infuria, contro il morire della luce."
Dylan Thomas

Indice

Introduzione

Il crescente aumento del numero di dispositivi connessi in rete e delle applicazioni che vengono sviluppate su di essa ha trasformato completamente lo standard delle applicazioni che una volta erano single-user, ovvero l'applicazione era locale all'utente e non interagiva con altri utenti, facendo quindi arrivare lo standard ad un modello multi-utente, dove si ha un ambiente collaborativo e condiviso in cui le azioni di un determinato utente vanno ad influire anche sugli altri. Questo aggiunge come requisito allo sviluppo un sistema di interconnessione degli utenti che permetta agli stessi di essere quindi connessi tra di loro per permettere la comunicazione.

Altra necessità al giorno d'oggi è inoltre che queste applicazioni multi-utente siano anche real-time, ovvero che si aggiornino in tempo reale e che di conseguenza l'azione di un singolo utente debba essere trasmessa e notificata a tutti gli altri in modo che possano gestirla di conseguenza, aggiungendo però come complessità il fatto che la latenza debba essere la più bassa possibile in modo da permettere un'alta reattività.

Basti pensare ad esempio alla quantità di videogiochi multi-player online che ci sono al momento, i quali necessitano che tutti i giocatori abbiano la versione della partita aggiornata in ogni istante e che le mosse che un giocatore fa debbano essere immediatamente visibili a tutti gli altri giocatori che visualizzeranno quindi il relativo cambiamento nell'ambiente di gioco.

Tali applicazioni riscontrano inoltre un utilizzo in una vasta gamma di ambiti, quali per esempio quello dell'Internet-of-Things, grazie al quale abbiamo un mondo dove quasi tutti gli oggetti sono connessi in rete e necessitano di comunicare tra di loro per condividere i dati raccolti dai sensori ed inviare strategie di attenuazioni alle componenti adibite. Questo richiede quindi un sistema di interconnessione e di scambio di dati che sia real-time e dove possono partecipare un numero elevato di componenti.

Infatti il numero di utenti o componenti che utilizzano questi servizi contemporaneamente è sempre più alto ed è quindi necessario capire come rendere scalabili tali applicazioni, in modo da non incontrare problemi dovuti al numero degli utenti, e questo va ulteriormente a complicare la sviluppo di queste applicazioni.

Per soddisfare queste necessità si è quindi gradualmente passati da architetture monolitiche, dove tutto il core del sistema si trovava all'interno di un server, a sistemi distribuiti basati su micro-servizi, i quali possono essere distribuiti e replicati, permettono quindi la scalabilità e l'efficienza.

Tuttavia esistono anche soluzioni architetturali e framework che risultano essere dedicati all'implementazione di tale tipo di applicazioni, il quale obiettivo è quindi quello di semplificarne e velocizzarne lo sviluppo, tramite l'utilizzo di pattern o architetture mirate all'implementazione di un tipo specifico di tali applicazioni.

Un esempio di tali soluzioni è Croquet, che permette la creazione di applicazioni real-time multi-utente, quindi con un ambiente che ne risulta condiviso tra tutti gli utenti connessi, senza la necessità di scrivere alcun codice per la parte server o per la gestione delle comunicazioni tra i peer, ma solamente sviluppando il modello del client, ovvero la parte in esecuzione sulla macchina dell'utente.

Questa tesi andrà quindi ad effettuare uno studio sul funzionamento e sull'architettura di Croquet proseguendo quindi con lo sviluppo di un middleware che permetta la creazione di applicazioni real-time multi-utente indipendenti dal linguaggio di sviluppo, le quali si potranno quindi appoggiare sull'architettura di Croquet per funzionare e comunicare, avendo quindi a disposizione tutte le garanzie che vengono offerte da Croquet e dalla sua architettura.

La tesi è cosi organizzata:

- **Capitolo I**: Viene introdotto il contesto delle applicazioni real-time multi-utente, con un particolare focus sui suoi ambiti di applicazione al giorno d'oggi e con le relative problematiche da gestire.

- **Capitolo II**: Viene introdotto Croquet descrivendo la sua storia e si arriverà quindi alla versione odierna di cui si analizzerà l'architettura e il funzionamento.

- **Capitolo III**: Viene effettuata un'analisi di come sia possibile l'integrazione di applicazioni sviluppate con Croquet con altri linguaggi di programmazione attraverso la creazione di un middleware che permetta di esportare la parte di View che verrà trasformata in un' entità Environment.

- **Capitolo IV**: Viene descritta l'implementazione del Proxy in Javascript che fungerà da middleware permettendo l'integrazione del modello Croquet con delle View sviluppate in altri linguaggi di programmazione.

- **Capitolo V**: Viene descritta l'implementazione della libreria client per l'utilizzo di tale middleware nel linguaggio Java, la quale si occuperà della creazione dell'Environment.

- **Capitolo VI**: Viene infine spiegata la metodologia di sviluppo di applicazioni basate su tale middleware tramite l'utilizzo di un esempio di tali applicazioni, creando quindi sia la parte di Model che andrà sviluppata con Croquet sia la parte client che esporrà l'Environment e andrà sviluppata sfruttando la libreria client in Java.

Elenco delle figure

Elenco dei codici

Capitolo 1

Panoramica

Un' applicazione real-time multi-utente consiste in un software al quale più utenti sono connessi ed interagiscono con l'applicazione contemporaneamente.

La caratteristica che differenzia questo tipo di applicazione è che gli utenti generalmente osservano un interfaccia che viene costantemente aggiornata nel momento in cui viene effettuata un'azione da uno qualsiasi degli utenti che stanno partecipando (anche l'utente stesso).

Si ha quindi come requisito quello di mettere in connessione tutti gli utenti in modo che ogni azione compiuta da uno dei client sia ricevuta nel più breve tempo possibile da tutti gli altri utenti. Questo permette quindi che una volta ricevuta la notifica dell'evento i client possano farne la relativa valutazione ed eseguire la relativa computazione.

Quello che si ha è quindi un applicazione con un ambiente condiviso tra tutti gli utenti, dove ognuno di essi può effettuare interazioni con tale ambiente ma soprattutto deve anche poter vedere le interazioni degli altri utenti. Inoltre quello che risulterà condiviso è anche lo stato attuale dell'applicazione, ovvero l'insieme delle componenti e delle modifiche che sono state effettuate.

1.1 Applicazioni

In un contesto come il mondo odierno tali applicazioni hanno subito un incremento esponenziale di utilizzo da parte sia delle persone sia di componenti software e hardware autonome che rimangono connesse. Infatti ci troviamo in una situazione in cui l'utilizzo di tali applicazioni fa parte della quotidianità delle persone, basti pensare a tutte le applicazioni che utilizziamo ogni giorno e che richiedono un funzionamento che rientra nelle casistiche precedentemente spiegate.

Ad esempio tutte le applicazioni di messaggistica e i vari social network rientrano in questa categoria, ma le applicazioni con tale meccanismo non si

limitano a questo, infatti rientrano in questa categoria anche i videogiochi, in particolare quelli in realtà aumentata che richiedono che tutti gli utenti collaborino in un mondo condiviso dove possono avere interazioni sia con il mondo sia con gli altri utenti.

Un altro esempio rilevante di utilizzo di tali applicazioni nella quotidianità delle persone sono tutti i servizi ad esempio di documenti condivisi, dove più persone possono procedere ad effettuare modifiche a tali documenti nello stesso momento, avendo quindi un effettivo ambiente condiviso che possono visualizzare.

Tuttavia tali applicazioni non vengono utilizzate solamente da persone ma possono essere impiegate anche in ambito dei sistemi distribuiti, o ancora più importante nel mondo dell'Internet of Things, dove insiemi di sensori ed attuatori comunicano in tempo reale i dati rilevati dall'ambiente ed elaborano strategie di attenuazione da applicare. Anche tali applicazioni quindi hanno bisogno di avere un ambiente che sia condiviso tra di esse e che permetta la comunicazione di eventi e dati in modo che il sistema possa funzionare correttamente.

1.1.1 Internet-of-Things

Come già anticipato abbiamo quindi che tale tipo di applicazioni risulta applicarsi bene all'ambito del cosiddetto Internet-of-Things, ovvero l'Internet delle cose, dove le *cose* sono intese come tutti quei dispositivi che sono connessi in rete.

Al giorno d'oggi infatti viviamo in una situazione dove il numero di oggetti *smart*, ovvero che non si limitano al funzionamento locale ma che hanno a disposizione funzionalità aggiuntive, quali l'essere connessi ad internet e comunicare tra di loro, è in un trend di crescenza esponenziale.

Le cosiddette *smart things* non risultano altro che essere tutti quei dispositivi embedded che abbiamo a disposizione nel mondo. Tra i più rilevanti sicuramente abbiamo gli smartphone, che sono posseduti da praticamente tutte le persone, i quali possiedono sensori che procedono alla rilevazioni da dati che sono quindi inviati in una rete per essere poi utilizzati. Inoltre le funzionalità offerte da tali device sono praticamente illimitate, dalla gestione delle chiamate all'interfacciamento con altri sistemi intelligenti.

L'idea alla base dell'Internet of Things è quindi quella di rendere digitale qualsiasi oggetto e ne possiamo notare l'effettiva implementazione ad esempio nell'evoluzione che stanno avendo gli elettrodomestici in questo momento, infatti ormai non è una novità trovare tali dispositivi che permettano di connettersi alla rete in modo anche da poter essere controllati da remoto dai nostri

stessi smartphone oppure da siti web. Tutte queste funzionalità aggiuntive risultano quindi essere una conseguenza della digitalizzazione delle cose.

Si ha quindi appunto anche in questo ambito la necessità che questi sistemi composti da innumerevoli componenti hardware e software ne risultino tra di loro interconnessi, avendo quindi bisogno di canali di comunicazioni interni ed esterni tramite i quali si possono interfacciare altri sistemi, andando quindi a migliorarne l'interoperabilità.

1.1.2 Videogiochi

Altra realtà importante, forse anche quella che si è specializzata in questo ambito da più tempo risulta essere quella dei videogiochi. Non tanto quando i videogiochi erano locali, ovvero che venivano eseguiti su una macchina e sulla quale l'utente giocava senza avere interazioni esterne, avendo quindi i cosiddetti giochi single-player, ma bensì quando tali giochi si sono evoluti ed hanno implementato delle funzionalità che permettevano quindi di giocare nella stessa partita a più giocatori.

Si è quindi giunti ad un modello di videogiochi definiti multi-player, con un particolare focus sul fatto che i molteplici player della partita non devono effettivamente essere connessi alla stessa macchina ma bensì ci deve essere una interconnessione tra tutte le piattaforme. Abbiamo infatti al giorno d'oggi che praticamente tutti i videogiochi hanno questa funzionalità, addirittura vengono anche prodotti videogiochi che risultano non avere nemmeno una modalità di gioco solitaria, ovvero dove gioca solo l'utente localmente, ma necessitano per forza di una connessione ad internet che permette il gioco online.

Oltretutto con l'espansione del Web e dei miglioramenti che hanno subito i browser, ma in generale la potenza di calcolo degli elaboratori odierni e della velocità delle connessioni, sempre più giochi vengono sviluppati direttamente per girare nei browser degli utenti, andando quindi a costruire un applicazione che non ha nemmeno bisogno di essere installata.

Ed ovviamente tutte le tipologie di questi videogiochi che sono multi-player online hanno la necessita di interconnettere tutti gli utenti, i quali devono avere un ambiente di gioco condiviso che si aggiorna in tempo reale in base alle azioni degli altri utenti.

1.1.3 Realtà aumentata

Un'altra applicazione molte valida ed in grande espansione al momento sono tutti quei sistemi di realtà aumentata che permettono quindi di interagire agli utenti con un mondo che ha tutti gli elementi del mondo fisico effettivo ma che in più possiede anche proprietà ed elementi che sono solamente virtuali.

Si ha quindi anche in questo caso la necessità di un ambiente virtuale complesso, che mappa il mondo reale e ne aggiunge elementi e funzionalità delle quali va tenuta traccia in modo che tutti possano vederle ed usufruirne. Anche in questo caso quindi bisogna fare si che lo stato degli oggetti condiviso sia coerente in ogni client in modo che non vi siano errori.

1.1.4 Documenti condivisi

Un ultimo esempio di applicazioni di uso comune che utilizza tale tipo di tecnologia sono tutti quei software per la gestione di documenti condivisi, che possono essere quindi acceduti e modificati in contemporanea da più persone. Si pensi ad esempio ad un applicazione come Google Docs, che permette la fruizione online di un servizio di editing di testo che estende le funzionalità di un semplice editor aggiungendone la possibilità di condividere tale documento con altri utenti. I quali potranno sia visualizzarli ma soprattutto potranno averne accesso in scrittura e quindi apporre modifiche nello stesso momento in cui anche un altro utente lo sta facendo.

Anche in questo caso si ha necessita di avere dei dati e degli oggetti che risultano condivisi tra tutti i client che stanno utilizzando tale software, in particolare nell'esempio di un documento condiviso su Google Docs si ha che tutti gli utenti che stanno scrivendo su tale documento devono vedere in tempo reale le modifiche effettuate dagli altri e viceversa, in modo che il documento rimango sincronizzato in ogni stato per tutti i client.

1.2 Problematiche

Dopo aver quindi introdotto tali applicazioni e il loro largo utilizzo si procede ad esplorarne i requisiti e le problematiche principali che sorgono quando si vogliono sviluppare tali applicativi.

Si ha infatti che tali applicazioni hanno uno sviluppo che non risulta per nulla semplice, infatti ci sono molte problematiche e requisiti che vanno tenuti in considerazione quando si sviluppano applicazioni di questo tipo.

Alcuni tra i più importanti elementi da tenere in considerazione sono quindi i seguenti[16]:

- **Networking**: ovvero la metodologia utilizzata per lo scambio dei messaggi tra gli utenti e le entità server dell'applicazione.
- **Sincronizzazione dello stato**: ovvero la gestione dello stato dell'applicazione che poiché distribuito su molteplici client bisogna assicurarsi che non vi siano incoerenze e che ogni utente abbia una versione corretta dello stato attuale in qualsiasi momento.

- **Persistenza**: ovvero il salvataggio in modo permanente dello stato dell'applicazione in modo che i client nel momento della connessione possano riprendere a lavorare dallo stato in cui è stato lasciato, o ad esempio in caso di disconnessione e riconnessione possano riprendere da uno stato congruo e non da uno che non è aggiornato all'ultima versione.

1.2.1 Networking

La gestione delle comunicazioni è tra le parti più importanti dello sviluppo di un applicazione multi-utente real-time proprio perché come già accennato in precedenza l'applicativo deve rendere possibile una comunicazione tra gli utenti in tempo reale, cosi che ognuno di essi possa ricevere gli aggiornamenti o le azioni compiute da altri utenti ed elaborale istantaneamente.

Vanno quindi utilizzate opportune tecniche e protocolli di scambio di messaggi IPC (Inter-Process Communication) che facciano in modo di ridurre al minimo la latenza e il tempo di ricezione dei messaggi ma che allo stesso tempo rendano più semplice e veloce lo sviluppo dell'applicazione.

Tra i protocolli più utilizzati per tali applicazioni troviamo quelli di tipo **Publish/Subscribe**.

Protocolli Publish/Subscribe

In un modello di protocollo di tipo **Publish/Subscribe** come da figura 1.1 un client ha la possibilità di pubblicare messaggi che verranno ricevuti da tutti gli altri client che si sono sottoscritti a quel tipo di messaggio.

Molto spesso in questo tipo di protocolli si ha una componente che funge da intermediario, che prende il nome di **Broker**, ovvero l'entità che si occupa dello smistamento e del filtraggio delle informazioni in modo da farle ricevere solamente a chi è sottoscritto a quel tipo di messaggio.

Un protocollo di questo tipo aiuta a separare i vari componenti del nostro sistema e a renderli tra di loro indipendenti nei seguenti modi[1]:

- **Spaziale**: chi si iscrive per ricevere determinate informazioni non ha bisogno di sapere l'indirizzo IP di chi pubblica i messaggi, ne tanto meno chi pubblica messaggi ha bisogno di sapere l'indirizzo IP dei destinatari.

- **Temporale**: i client non devono essere per forza online nello stesso momento, si possono utilizzare delle politiche di caching in modo che i messaggi vengano ricevuti alla ri-connessione.

- **Sincronismo**: la pubblicazione e la ricezioni di messaggi non sono operazioni bloccanti poiché si tratta comunque di un protocollo di IPC asincrono.

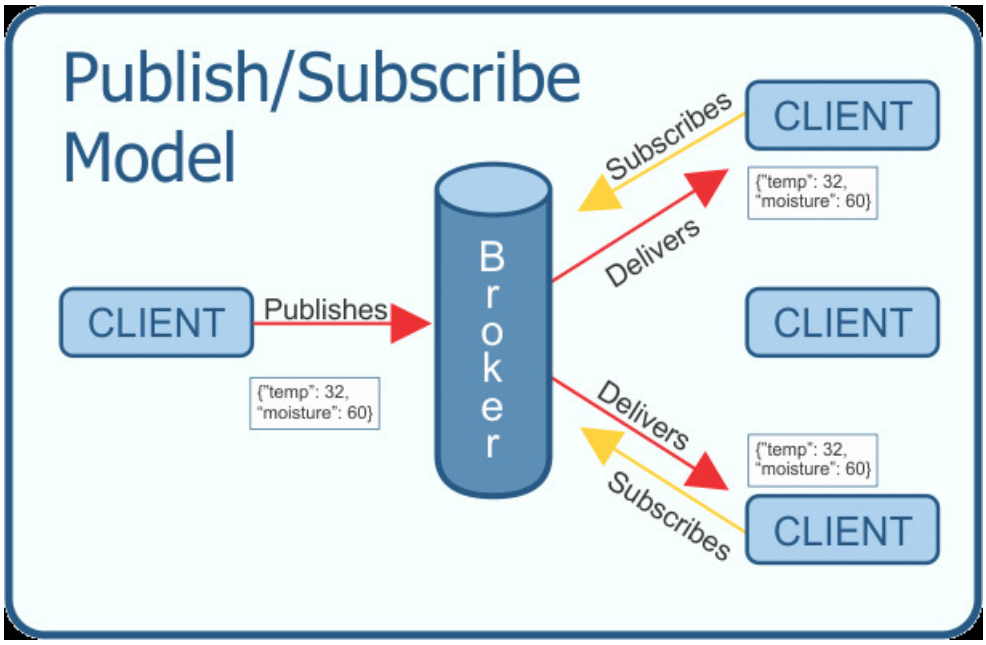

Figura 1.1: Architettura di un protocollo di tipo Publish/Subscribe[1]

.

1.2.2 Sincronizzazione dello stato

La sincronizzazione dello stato è forse la tematica più importante insieme alla gestione del networking quando si parla di applicazioni multi-utente real-time. Infatti considerato lo scopo di tali applicazioni si ha la necessità che in ogni istante tutti gli utenti abbiano accesso gli stessi dati e che non ci siano incongruenze tra i suddetti nei client poiché incongruenze di questo tipo potrebbero portare a situazioni non previste e di conseguenza ad errori nel funzionamento del sistema.

Una tecnica per la sincronizzazione di dati su più client è una metodologia che viene utilizzata dalle architetture basate su micro-servizi che permette di mantenere i dati sincronizzati su più componenti. Infatti tali applicazioni poiché sono strutturate per essere divise in servizi autonomi hanno bisogno di replicare i dati, o per lo meno parti dei dati, in modo da non dover per forza ogni volta richiederli alle altre componenti che potrebbero non essere disponibili.

Quello che si fa in questi casi è prevedere che quando uno dei componenti modifica i propri dati invii un evento di notifica il quale contenuto rappresenti quale modifica è stata effettuata. In questo modo che chi è interessato alle modifiche su quei dati si metterà in ascolto per questi eventi e una volta ricevuti

possa replicare la modifica nel suo contesto locale ed avere anch'esso i dati aggiornati[21].

1.2.3 Persistenza

La persistenza è un altro dei pilastri fondamentali delle architetture per applicazioni real-time multi-utente, infatti si ha la necessità di mantenere salvati i dati delle sessioni, in particolar modo garantire che quando un utente si disconnette, esso non debba perdere tutti i dati e deve quindi poterli recuperare nel momento in cui si riconnette, potendo ricostruire cosi uno stato conforme a tutti gli altri client nel suo contesto locale.

La necessità è quindi quella di salvare in modo persistente i dati in modo che essi siano sempre accessibili da tutti.

Una tecnica molto basilare è quella di avere un componente del sistema che sia adibito allo storage dei dati delle sessioni, che rimane quindi sempre sincronizzato in modo da avere in un qualunque momento i dati aggiornati così da renderli sempre disponibili ai vari client che li necessitano. Tale necessità può essere dovuta sia a problemi di connessione ma anche all'unione alla sessione di un nuovo utente che non ha nessun dato nel momento del suo avvio.

Capitolo 2

Croquet

Croquet è una piattaforma per la creazione di applicazioni multi-utente real-time, quindi basata sul modello di ambiente condiviso, grazie alla quale invece di dover scrivere codice sia per l'applicazione a livello client, sia a livello delle componenti server e per quelle di rete, gli sviluppatori hanno come unico compito quello di scrivere il codice dei client, il quale sarà eseguito all'interno di una macchina virtuale condivisa (VM) che si trova in esecuzione su ogni peer della sessione, e che grazie all'architettura di Croquet rimane costantemente sincronizzata in ognuno di essi sia a livello temporale di esecuzione sia per quanto concerne la sincronizzazione dello stato[19].

Croquet ha come fondamento quindi quello in cui ogni peer ha in esecuzione una macchina virtuale che ne risulta identica bit-a-bit a tutte le altre, in modo da fornire l'illusione che essa sia una sola macchina virtuale condivisa.

La sincronia è inoltre garantita dal controllo del tempo delle macchine virtuali dei vari peer e dell'intera sequenza di eventi che vengono trasmessi sulla rete da parte dell'architettura di Croquet.

Questo garantisce un modello che riduce notevolmente la complessità della fase di sviluppo di tali applicazioni poiché non ci si deve preoccupare di tutte le problematiche di rete, di scambio di messaggi per la sincronizzazione e soprattutto dello sviluppo di componenti server per mettere in comunicazione i vari client, i quali sono problemi in cui invece si incorre per sviluppare tali applicazioni utilizzando le classiche architetture client-server.

Scopo di Croquet è quindi quello di permettere lo sviluppo di applicazioni basate su un ambiente condiviso in modo semplice e veloce, con una logica di sviluppo efficiente e particolare che verrà presentata e spiegata nelle sezioni successive.

2.1 Storia

Croquet nasce come un architettura software nel 2003[22], la quale era implementata con il linguaggio Squeak, che altro non è che un estensione del linguaggio Smalltalk[24].

Il suo scopo era quello di essere una piattaforma di sviluppo per ambienti collaborativi, con un particolare focus agli ambienti 3D condivisi, nei quali ogni utente poteva interagire e vedere sia gli altri utenti sia le azioni che essi stavano compiendo. Inoltre il sistema permetteva anche di avere molteplici stanze virtuali, le quali davano accesso a diversi servizi e tali stanze erano tra di loro interconnesse tramite portali, che permettevano quindi lo spostamento tra le suddette stanze. In sostanza quello che Croquet voleva realizzare era una rappresentazione a livello fisico del Web, ovvero delle connessioni (link) che si hanno tra tutte le risorse online.

Il sistema era basato sul protocollo TeaTime[20] che garantiva queste funzionalità di interazione tra utenti in tempo reale in uno stesso spazio 3D che era quindi condiviso, il quale veniva gestito invece a livello grafico tramite la libreria OpenGL.

Una considerazione che si può fare quindi su Croquet è che effettivamente lo si può considerare come il precursore di quello che oggi viene definito Metaverso, poiché aveva già una visione di un ambiente tridimensionale condiviso dove molteplici utenti potevano entrare a farne parte ed interagire sia con il mondo condiviso sia con gli altri utenti.

2.2 Tempo virtuale

Il modello di funzionamento attuale di Croquet, cosi come quello pensato alle sue origini si basava appunto su un concetto di tempo virtuale. L'idea è quindi quella di avere un sistema distribuito in cui ogni componente lavora non basandosi sul tempo reale ma su un tempo virtuale che rimane sincronizzato tra tutti i peer e che può quindi scorrere indipendente da quello reale[15].

Questo permette di ottenere una sincronia anche a livello di computazione, infatti si ha che l'esecuzione di operazioni e la ricezione e l'invio di eventi si baseranno anch'essi su tale tempo.

Quello che si ottiene quindi è una metodologia per la creazione di sistemi distribuiti, dove si vanno ad imporre dei vincoli di temporizzazione e di controllo del flusso per ottenere un controllo sulla sincronizzazione.

Tale protocollo quindi risulta avere un applicazione molto valida nel caso si voglia ottenere una sincronizzazione forte in sistemi distribuiti, quindi ad esempio come meccanismo di coordinazione di simulazioni distribuite su più peer.

Inoltre questo controllo temporale se unito ad un controllo sulle modifiche porta ad un controllo completo sullo stato dell'applicazione, il cui tempo quindi è orchestrato e che può effettuare anche correzioni nel caso di malfunzionamenti di tale meccanismo di sincronizzazione.

Nel modello architetturale di Croquet precedente il protocollo utilizzato per la gestione del tempo virtuale era il precedentemente citato TeaTime[20].

2.3 Croquet oggi

Il sistema attuale di Croquet è stato cambiato rispetto al precedente, infatti non si pone più come obiettivo la creazione di un mondo 3D condiviso e interconnesso ma bensì la creazione di applicazioni con ambienti condivisi che rientrano quindi nella tipologia di applicazioni real-time multi-utente ma senza doverne specificare il contesto.

Infatti non si ha ora la necessità di utilizzare librerie grafiche particolari, bensì il framework è pensato per lo sviluppo di ambienti condivisi che rimangono sincronizzati e che cooperano tra di loro. Lo si può considerare quindi come uno strumento generico per la creazione di tali tipi di applicazioni, sopra al quale poi si può costruire tutto l'ambiente effettivo.

Il framework attuale è completamente implementato in JavaScript e può essere eseguito sia in ambiente browser sia tramite Node.js. Il tutto è stato reso possibile dallo standard ECMAScript che garantisce conformità di esecuzione anche su piattaforme diverse, non rendendo così lo sviluppo delle applicazioni dipendente dalla piattaforma in cui la si sviluppa.

Inoltre come già anticipato si sta avendo un trend sempre più crescente delle cosiddette applicazioni strutturate come Software-as-a-Service, che vengono quindi rese disponibili tramite la rete e in particolare tramite il Web. Basta quindi avere un browser ed una connessione ad internet per poter utilizzare queste applicazioni tramite il proprio browser, ed anche Croquet ha seguito questa linea di sviluppo, infatti essendo la libreria scritta in JavaScript ed andando in esecuzione sul browser del client non ci sarà bisogno di installare nulla.

Per come è quindi strutturato il framework, l'unica parte da sviluppare è la parte che viene eseguita sulle macchine degli utenti e risulta inoltre facilitata anche la distribuzione del software poiché basta una qualsiasi pagina web di tipo statico che fornisca il codice che viene quindi scaricato dal browser ed eseguito al suo interno.

2.4 Architettura

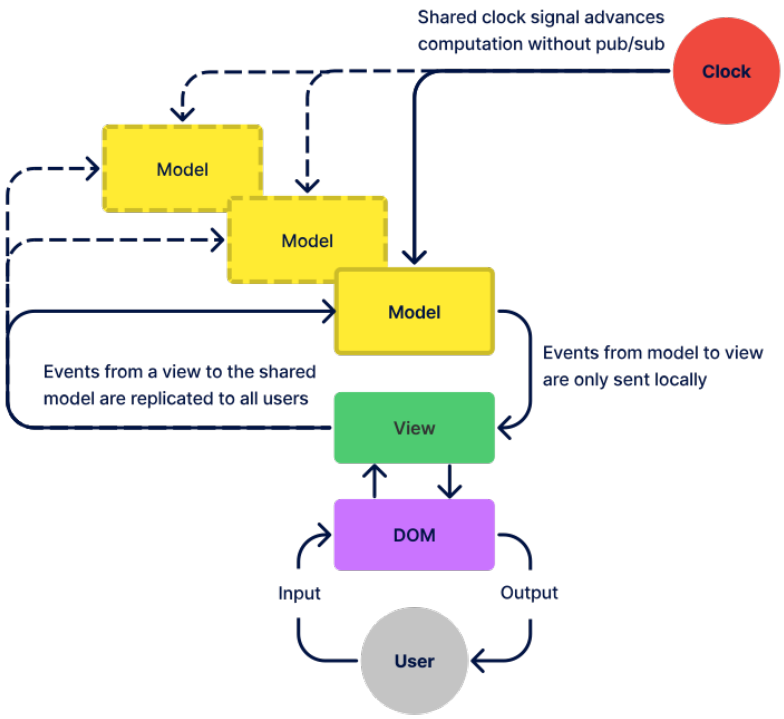

Figura 2.1: Architettura di Croquet[2]

.

Si passa ora allo studio dell'architettura su cui è costruita la libreria di Croquet. In particolare essa basa le sue applicazioni su due componenti fondamentali:

- **Model**: ovvero la parte condivisa, la quale è in esecuzione nella macchina virtuale di ogni singolo peer che rimane costantemente sincronizzata.
- **View**: ovvero la parte non condivisa, che per ciascun client l'interazione con l'utente stesso ma anche gli eventi da inviare alla parte di Model.

Tutta la computazione dell'applicazione e i relativi cambiamenti di stato vengono eseguiti solamente nei peer connessi. Quando uno di questi peer ha bisogno di effettuare un cambiamento di stato, ad esempio come conseguenza di un azione di input da parte dell'utente esso procede ad informare tutti i peer trasmettendo un messaggio ai server di sincronizzazione di Croquet. Questi server una volta ricevuto il messaggio ci inseriranno il timestamp, ovvero il tempo virtuale del momento dell'arrivo, dell'evento e lo "rifletteranno" verso tutti i peer connessi alla sessione senza neanche avere la necessità di analizzare il contenuto dei messaggi. Proprio per questo loro meccanismo di funzionamento i server di Croquet vengono chiamati **Reflector**.

L'insieme di questi 3 componenti costituisce quindi quella che l'architettura attuale di Croquet che prende il nome di **Model-View-Reflector**[2] e che viene mostrata dalla figura 2.1.

Si andrà ora quindi ad esplorare i singoli componenti e a descrivere il loro funzionamento.

2.4.1 Model

Il Model, ovvero il modello, è la componente attiva dell'applicazione. Esso consiste nell'insieme degli oggetti che sono effettivamente replicati e mantenuti sincronizzati tra tutti i peer nella stessa sessione.

Ma il suo scopo non si limita al contenimento e al salvataggio dei dati bensì quello di cui si occupa anche il Model è l'effettiva esecuzione dell'applicazione a livello interno. Si ha infatti che tutta la computazione relativa alla simulazione viene effettuata da questi oggetti, i quali risiedono in maniera replicata in ciascuna delle macchine virtuali degli utenti.

La sincronia della parte di Model di una applicazione sviluppata con Croquet viene gestita come già anticipato dai Reflector, i quali selezioneranno i Model da tenere sincronizzati osservando a quale sessione dell'applicazione essi stanno partecipando. In questo modo una volta avviata l'applicazione ed effettuato il join della sessione, tutti i client verranno messi in comunicazione in modo da garantire la sincronia delle macchine virtuali e quindi della parte di Model di ogni singolo peer.

Avendo quindi appurato che la parte di Model non concerne solo lo stato a livello di dati ma anche tutta l'effettiva computazione dell'applicazione si può osservare come esso basi il suo meccanismo di esecuzione di istruzioni su due principi fondamentali:

1. Utilizzando la schedulazione di azioni specificandone l'offset temporale.

2. Reagendo alla ricezione di eventi generati dalle View.

Schedulazione azioni futuri

La schedulazione di azioni da svolgere nel futuro può essere fatta tramite le API di gestione delle operazioni futuri offerte dal framework e saranno gestite internamente alla macchina virtuale allo scorrere del tempo della simulazione, il quale è gestito dai Reflector che fanno progredire il tempo in maniera sincronizzata tra tutti i peer.

In questo modo il tempo della simulazione risulta essere lo stesso in ogni istante in ciascuno dei peer. Il tempo del modello sarà approfondito in una sezione apposita successivamente.

Gestione eventi

Per quanto riguarda invece la gestione degli eventi, le parti del Model rispondono a quelli generati dalla View che saranno però prima inviati ai Reflector che si occuperanno quindi di marchiarli con il tempo della simulazione del momento in cui sono arrivati (che viene appunto mantenuta all'interno del Reflector) e successivamente li invieranno a tutti i Model connessi in modo che la loro esecuzione sia sempre sincronizzata basandosi sullo scorrere del tempo della simulazione[5].

Ovviamente anche la parte di Model ha la possibilità di generare eventi, in particolare tale possibilità viene sfruttata per emettere eventi di notifica quando vengono effettuate modifiche ai dati, in modo che la parte di View che è in ascolto possa riceverle e di conseguenza aggiornarsi in base ai nuovi dati.

Model multipli

Per semplificare la creazione di applicazioni con Croquet è stato scelto di non limitare l'utente ad avere una sola classe dove fosse racchiuso tutto il modello, bensì è possibile definirne molteplici in modo da avere una separazione dei ruoli che ognuno di essi deve svolgere e dei relativi dati contenuti in ciascuno di tali componenti.

Questo aiuta quindi la decomposizione nella scrittura di un modello Croquet, potendone definire ogni singola componente a se stante e creando quindi una struttura a moduli dove ogni elemento ha un proprio scopo e un proprio funzionamento. Tale funzionamento può essere sia indipendente, quindi non necessità di comunicare con altri componenti del modello, sia di tipo dipendente, ovvero esiste della cooperazione tra componenti del modello che interagiranno quindi utilizzando anch'essi gli eventi.

Registrazione dei Model

La registrazione dei modelli è un operazione necessaria che va effettuata all'avvio dell'applicazione prima della procedura di join di una sessione. Essa consiste nella registrazione all'interno della macchina virtuale dei vari Model implementati, in modo che il serializzatore sappia poi come ricrearne delle istanze a partire dagli snapshot e viceversa generarle nel momento in cui viene scelto per fare l'upload dello snapshot.

Inoltre come verrà spiegato successivamente nella sezione dedicata alla Sessione, i dati di registrazione dei modelli verranno utilizzati per controllare che tutti i client della stessa sessione stiano usando lo stesso identico modello.

Gestione numeri casuali

Un altra componente molto importante che è gestita dal framework di Croquet è la generazione di numeri casuali nella parte di Model. Infatti tutto il modello deve essere sempre identico bit-a-bit in tutte le macchine su cui viene eseguito e deve quindi rimanere sincronizzato in ogni momento, tuttavia se il modello avesse bisogno in qualche modo di generare dei numeri random questi sarebbero differenti in ogni macchina e quindi andrebbero a rompere la sincronia che c'è tra tutti i modelli.

Per ovviare a questo problema Croquet ha implementato la gestione della randomness nella loro macchina virtuale in modo che alla generazione di un numero random all'interno del model essa ritorni lo stesso numero su ogni macchina connessa[6].

Ovviamente questa considerazione non vale anche per la parte della View poiché essa è la parte non condivisa e quindi è corretto che la generazione del numero random sia differente da macchina a macchina.

2.4.2 View

Le View costituiscono la parte non replicata di un applicazione sviluppata con Croquet, ovvero quelle che risiedono nella macchina dell'utente ma non nella macchina virtuale gestita da Croquet. Infatti esse non hanno la necessità di essere sincronizzate ma come unico requisito devono poter aver accesso ai dati del modello.

La creazione di tali View infatti viene gestita nel momento in cui si effettua il join di una sessione ed essa è indipendente dalla gestione della parte del Model, infatti il modello verrà inizialmente generato ed una volta ristabilito lo stato attuale della sessione si potrà procedere alla generazione della parte di View.

Comunicazione con il Model

Come già accennato la parte di View comunica con quella di Model solamente tramite eventi. Infatti la sincronia dei modelli nelle macchine virtuali è garantita dal fatto che le modifiche vengano effettuate solamente dal modello stesso alla ricezione di eventi tramite i Reflector che garantiscono l'esecuzione di eventi nello stesso istante tra tutti i peer, e per questo motivo risulta proibito quindi che la View effettui modifiche al modello in locale senza passare tramite gli eventi.

Tuttavia alla View viene concesso l'accesso diretto ai dati del modello ma solamente con lo scopo di lettura, in questo modo essa può visualizzare lo stato attuale e di conseguenza rappresentarlo graficamente.

Ricezioni eventi dal Model

Anche la View come il modello è predisposta per la ricezione di eventi, in particolare quelli generati dalla parte di Model, che si trova in esecuzione nella macchina virtuale locale dell'utente. Infatti essa una volta effettuate modifiche ai dati dovrà notificare la View di tale cambiamento in modo che essa possa aggiornarsi in conformità ai cambiamenti senza dover utilizzare delle tecniche di polling per notare dei cambiamenti.

Generalmente una sola View può sottoscriversi a tutti gli eventi del Model, tuttavia è una buona pratica costruire delle sottoclassi multiple della View in modo da rispecchiare tutte le varie componenti del Model[10].

Il concetto di View in Croquet

Il concetto di View di Croquet tuttavia non richiede che effettivamente ci sia una parte visiva, infatti la classe base della View fornisce solamente le API di pubblicazione/sottoscrizione ad eventi in modo da poter reagire quando avviene un cambiamento nella parte di Model e di proporne altrettanti tramite la pubblicazione di eventi scatenata dall'utente.

A sostegno di ciò infatti non vengono fatte assunzioni sul framework che vada usato nella parte di View da parte di Croquet, poiché l'interfaccia di View è stata appunto definita in modo generico.

2.4.3 Eventi

Nell'architettura di Croquet le due entità principali, ovvero **Model** e **View** comunicano tra di loro tramite eventi. Infatti entrambe le classi mettono a disposizione le stesse API per la comunicazione, la quale è basata su un meccanismo di tipo Publish/Subscribe.

Molto importante è tenere in considerazione come Croquet gestisca la pubblicazione degli eventi in base alla parte in cui ci troviamo.

Nel caso in cui una pubblicazione venga effettuata da un Model verso una View allora l'evento sarà gestito solo localmente e sarà recapitato alle View in ascolto.

Nel caso in cui la pubblicazione sia fatta dalla View verso il Model allora essa verrà inviata al server di replicazione (**Reflector**) che successivamente la invierà a tutti i Model connessi, compreso quello relativo alla View che ha inviato l'evento. In questo modo il tempo di arrivo dell'evento è lo stesso su tutti quanti i peer.

Per la gestione di eventi che invece sono locali alla componente, come nel caso da Model a Model allora anche in questo caso esso viene gestito solamente internamente senza il passaggio da nessun server. Alla pubblicazione sarà quindi schedulata l'esecuzione della funzione di handle che è stata registrata per rispondere all'evento appena pubblicato.

Come nel caso Model to Model anche gli eventi da View a View vengono gestiti solo localmente con la stessa strategia spiegata sopra.

Inoltre Croquet permette ovviamente che più Model o View possano sottoscriversi allo stesso evento e garantisce che ognuna delle componenti che sia sottoscritta ad un determinato evento riceva la notifica all'arrivo di tale evento.

2.5 Reflector

I Riflettori (Reflector) sono le uniche componenti server di Croquet, infatti sono dei servizi stateless pubblici che sono allocati e distribuiti nel mondo.

Il loro scopo come già accennato è quello di occuparsi dello smistamento e del ridirezionamento dei messaggi che sono stati inviati dai client per poi rifletterli verso tutti i peer della sessione a cui il messaggio si riferisce, aggiungendone il relativo tempo della simulazione nel momento della ricezione, il quale risiede appunto dentro il Riflettore.

Tuttavia lo scopo dei Riflettori come già accennato è anche quello di tenere al suo interno il tempo relativo alla simulazione, mandando dei segnali a tutti i peer allo scorrere del tempo in modo che essi possano fare proseguire di conseguenza il tempo interno della macchina virtuale così che possano essere eseguite le operazioni schedulate e gli eventi ricevuti con quel determinato tempo.

The Croquet Reflector Network (CRN)

La rete composta dai Reflector di Croquet (CRN) offre prestazioni che garantiscono una latenza molto bassa anche con reti di tipo Edge e 5G, sia

pubbliche che private[3].

Il suo scopo è quello di eliminare i server dedicati e ancora più in generale il codice lato server per lo sviluppo di applicazioni real-time multi-utente, rendendo possibile la comunicazione e la gestione di tutto l'applicativo solamente basandosi sullo sviluppo del client e appoggiandosi appunto sulla suddetta rete di riflettori.

Poiché essi non devono fare alcun tipo di computazione pesante, se non il ridirezionamento dei messaggi, ne risultano molto leggeri per quanto riguarda il consumo di risorse e questo rende possibile il deploy di innumerevoli di questi servizi in tutto il mondo mantenendo un costo molto basso.

Avendo quindi una distribuzione uniforme di riflettori in tutto il globo questo permette di ridurre ulteriormente la latenza delle applicazioni che si basano su Croquet, infatti all'avvio la procedura di connessione ai server di replicazione verrà fatta cercando il riflettore più vicino al client.

Per quanto riguarda la sicurezza tutti i messaggi che passano per i riflettori sono crittografati utilizzando la crittografia end-to-end ed in questo modo tutto il sistema ne risulta sicuro.

2.5.1 Sessione

Una sessione consiste in un istanza di un applicazione sviluppata tramite Croquet in cui un insieme di client che eseguono tale applicazione partecipano e si scambiano messaggi tra di loro interagendo.

Nel caso di Croquet durante l'avvio dell'applicazione, in particolare nel momento del Join di una sessione andrà specificato come parametro il nome di tale sessione e la password di accesso che sarà la base della chiave crittografica usata per la comunicazione con i Reflector.

Questo rende quindi possibile generare istanze multiple dell'applicazione così da effettuare una divisione degli utenti connessi in più gruppi.

Nel caso la sessione non esista sarà il primo client a connettersi a tale sessione a crearla decidendone così anche la password.

Tuttavia l'identificativo vero e proprio della sessione non consiste nel suo nome ma bensì in un valore alfanumerico che viene calcolato da Croquet all'avvio dell'applicazione, il quale è generato a partire dai seguenti dati:

- il nome della sessione.
- un hash generato da tutte le classi del modello e dalle costanti definite.

Questo permette di accedere alla stessa sessione solamente i client che hanno in esecuzione lo stesso esatto codice sorgente e viene così assicurata che la risposta di questi modelli agli eventi ricevuti e quelli schedulati sia esattamente la stessa,

il quale è uno dei prerequisiti per garantire la perfetta sincronizzazione tra tutti i Model.

Questo identificativo verrà quindi poi utilizzato per l'accesso ai server di replicazione di Croquet.

2.5.2 Gestione del tempo

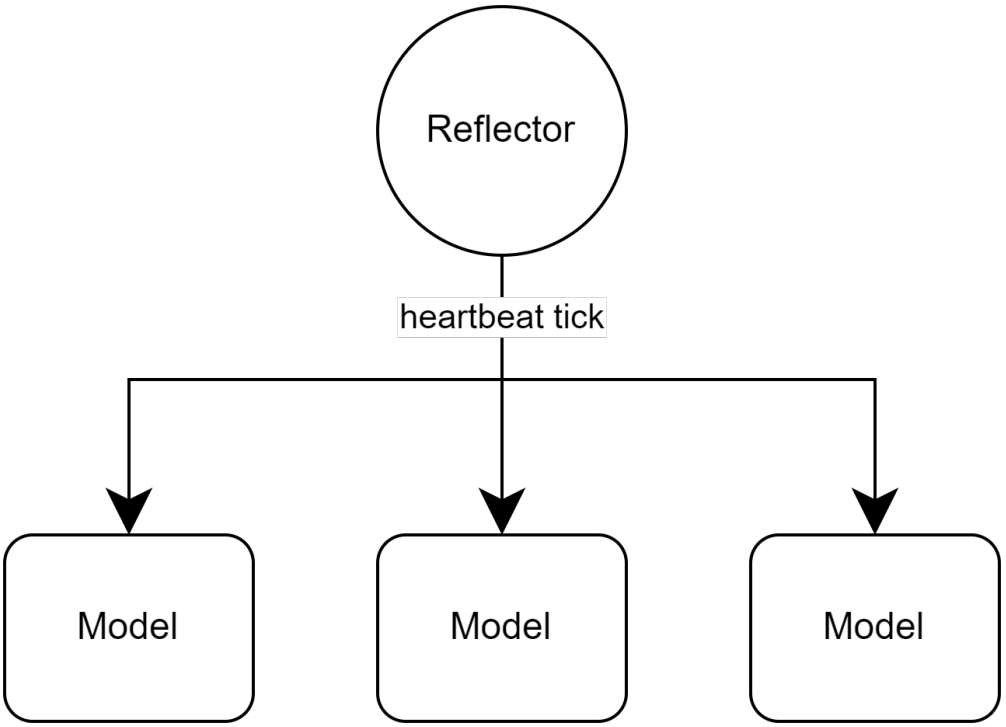

Figura 2.2: Gestione del tempo del modello

.

Il modello di Croquet non è al corrente del tempo reale del mondo, l'unica cosa che conosce è il tempo della simulazione, ovvero il tempo trascorso dall'inizio della sessione che non sarà altro che un tempo virtuale su cui si basa la macchina virtuale di Croquet[8].

A dirigere il tempo di tutti i client connessi abbiamo il Reflector, che alla creazione di una sessione inizierà a tenere traccia del tempo trascorso e questo dato costituirà il tempo della simulazione relativo a quella sessione, che altro non sarà che il suo tempo virtuale.

Le macchine virtuali aspetteranno quindi le notifiche da parte del Reflector per fare procedere il tempo virtuale interno cosi che possano essere eseguite le operazioni schedulate oppure gli eventi in coda che hanno un timestamp superiore al tempo corrente.

Infatti il Reflector avrà anche come compito quello di inviare dei tick di **Heartbeat** ai modelli connessi, ad una frequenza specificata che varia tra le 20 e le 60 volte al secondo, i quali una volta ricevuti faranno procedere il tempo locale della macchina virtuale che procederà quindi ad eseguire eventuali funzioni schedulati o eventi in coda. Tale procedimento è illustrato dalla figura 2.2.

Inoltre come già accennato più volte precedentemente il riflettore ha anche come lavoro quello di ricevere gli eventi dalle View e dopo averli taggati con il tempo della simulazione nel momento dell'arrivo (che nel Reflector continua a scorrere) provvederà quindi ad inoltrarli a tutti i modelli che li metteranno in coda, in attesa del progredire del tempo nella macchina virtuale per poterli eseguirli.

2.6 Snapshot

Per risolvere la problematica della persistenza dei dati, Croquet si basa sui cosiddetti **Snapshot**, ovvero delle istantanee dello stato attuale dell'applicazione che vengono periodicamente salvate su dei server appositi in modo da poter ripristinare lo stato in qualunque momento[9].

La gestione delle istantanee è effettuata dai riflettori, che periodicamente si occupano di selezionare uno tra i peer per ottenere una copia del suo stato attuale (che sarà esattamente uguale a tutti gli altri peer) per poi farne l'upload su dei server di salvataggio dati in modo da renderli disponibili ai nuovi utenti o in caso di disconnessione per poter ricostruire l'esatto stato della macchina virtuale.

Infatti oltre al salvataggio degli Snapshot Croquet salva anche gli eventi e il relativo tempo della simulazione in cui sono arrivati, questo permette una ricostruzione esatta dello stato dell'applicazione in un qualsiasi momento.

2.6.1 Ricostruzione dello stato

Il processo di ricostruzione dello stato in caso di un nuovo client che si unisce alla sessione o in caso di riconnessione è il seguente[9]:

1. Il modello viene inizializzato con i dati dell'ultimo snapshot disponibile.
2. Il riflettore invierà al modello tutti gli eventi che sono stati inviati nella sessione a partire dal timestamp relativo all'ultimo snapshot.

3. Il modello a questo punto simulerà l'arrivo di tutti gli eventi con il progredire del tempo virtuale ed eseguirà quindi le relative funzioni di handle e poiché il modello è lo stesso di tutti gli altri peer una volta simulati tutti questi eventi si ritroverà in uno stato esattamente identico a quello degli altri client.

4. Viene a questo punto inizializzata la View in modo che possa sottoscriversi agli eventi ed incominciare ad interagire.

2.7 Modello di sviluppo

Si procederà ora a mostrare un esempio di applicazione sviluppata con Croquet per capirne al meglio i meccanismi e le API esposte.

L'applicazione che si andrà ad analizzare prevede una contatore condiviso che continua ad incrementarsi ogni secondo, e che può essere resettato da uno qualsiasi degli utenti in un qualsiasi momento.

Il requisito è quindi avere questo contatore condiviso che si dovrà vedere progredire allo stesso modo su ogni client connesso, in modo che tutti visualizzino lo stesso valore come se appunto la macchina virtuale fosse una unica.

Prima di procedere alla creazione delle applicazioni sarà necessario illustrare le API esposte da Croquet in JavaScript.

2.7.1 Croquet API

Si procede quindi esponendo quali sono le API offerte da Croquet per lo sviluppo di tali applicazioni, con particolare focus sulla gestione degli eventi e del tempo.

API Eventi

Nelle API per la gestione degli eventi bisogna comprendere come Croquet ha pensato di incapsulare tale meccanismo, partendo innanzitutto dai parametri che costituiscono un evento:

- **Scope**: ovvero il contesto dell'evento
- **Event**: ovvero l'evento concreto
- **Data**: ovvero i dati dell'evento (opzionali)

Il parametro **Scope**, in questo caso, aiuta a semplificare la decomposizione degli eventi per contesto. Ad esempio nel caso in cui un applicazione sia divisa in più modelli si potrebbe usare il nome del modello come scope per gestire solamente gli eventi relativi ad esso.
Per la pubblicazione degli eventi l'API messa a disposizione è[4]:

- **publish(scope, event, data)**: permette di effettuare una pubblicazione di un evento con i parametri specificati.

Per la sottoscrizione ad eventi le API messe a disposizione sono:

- **subscribe(scope, event, handler)**: permette di sottoscriversi agli eventi con lo scope e l'event specificato, registrando una funziona come handler che verrà eseguita ogni qual volta che sia ricevuto tale evento.
- **unsubscribe(scope, event)**: permette di rimuovere una sottoscrizione così che non si ricevano più tali eventi.
- **unsubscribeAll()**: permette la rimozione di tutte le sottoscrizioni.

API Tempo

Il framework di Croquet mette a disposizioni anche API per la gestione del tempo, soprattutto in merito all'esecuzione temporizzata di funzioni in modo da programmarla ad un determinato tempo nel futuro facendo si che venga eseguita in contemporanea su tutti i peer.
Tali API sono le seguenti[8]:

- **this.now()**: utilizzata nel Model permette di ottenere il tempo attuale della simulazione.
- **this.future(delay).methodName()**: permette di schedulare l'esecuzione di un metodo ad uno specifico offset di tempo in millisecondi.

Anche le View hanno a disposizione queste API, tuttavia esse sono basate sul tempo locale della macchina su cui vengono eseguite e non su quello della simulazione.

2.7.2 Counter Model

```
// CounterModel.js
import { Model } from '@croquet/croquet'

export class CounterModel extends Model {

    init(options) {
        super.init(options)
        this.counter = 0
        this.future(1000).tick()
        this.subscribe("counter", "reset", this.resetCounter)
    }

    tick() {
        this.counter += 1
        this.publish("counter", "updated")
        this.future(1000).tick()
    }

    resetCounter() {
        this.counter = 0;
        this.publish("counter", "updated")
    }

}
```

Per la parte del modello in questo caso non si hanno componenti molteplici, infatti l'unica cosa che deve gestire il Model è l'incremento del contatore e il reset di tale valore quando l'utente lo desidera. Questo rientra perfettamente nelle casistiche in cui è sufficiente una sola classe per il Model.

Tale classe prenderà il nome quindi di **CounterModel** e al suo interno avrà come variabili solamente quella contatrice che ovviamente viene inizializzata a 0 all'avvio della sessione.

Incremento automatico

Viene poi programmata l'esecuzione futura (con delay ad 1 secondo) del metodo *tick* del modello, il quale provvederà ad incrementare il contatore e notificherà il cambiamento del valore così che la View possa sottoscriversi e ricevere l'evento.

Alla fine di tale cambiamento quindi il modello provvederà a ri-schedulare l'esecuzione del metodo *tick* con un altro secondo di delay. In questo modo

si otterrà un meccanismo di schedulazioni future della funzione di *tick*, la quale verrà invocata ad ogni secondo basandosi sugli aggiornamenti del tempo virtuale che vengono inviati dai riflettori.

Tutta questa parte quindi lavora in modo automatico su ogni client e rimane costantemente sincronizzata dato che il tempo dei modelli è virtuale ma soprattutto viene mantenuto in sincronia in modo che le funzioni schedulate vengano eseguite nello stesso esatto momento su tutte le macchine.

Reset da parte dell'utente

Si è giunti ora alla parte di reset del contatore, la quale deve necessariamente venire dalla parte dell'utente, ovvero della View. Viene quindi fatta una sottoscrizione con scope "counter" e tipo di evento "reset" e si registra la funzione *resetCounter* per far si che venga eseguita ogni qual volta che venga ricevuto uno di questi eventi.

Tale funzione quindi una volta invocata resetterà il contatore e notificherà anche in questo caso il cambiamento in modo che la View possa sapere che il valore è stato modificato senza dover fare polling.

2.7.3 Counter View

```
// CounterView.js
import { View } from "@croquet/croquet";

export class CounterView extends View {

    constructor(model) {
        super(model);
        this.model = model;
        this.subscribe("counter", "updated", this.onCounterUpdated);
    }

    onCounterUpdated() {
        console.log("Counter updated to", this.model.counter);
    }

    resetCounter() {
        this.publish("counter", "reset");
    }

}
```

Si procede ora quindi alla parte di View, la quale dovrà mostrare il valore del contatore che si auto incrementa (in questo caso è stato scelto un semplice print alla console). Inoltre dovrà permettere all'utente di resettare il contatore.

Viene così costruita la classe **CounterView**, che una volta salvato il riferimento al modello per avere l'accesso in lettura ai dati, procederà a sottoscriversi alle notifiche di cambiamento del valore del contatore, registrando quindi la funzione *onCounterUpdated* a tale evento in modo che venga invocata ad ogni cambiamento pubblicato dal modello. Tale funzione come da requisiti stamperà il nuovo valore del contatore.

Per quanto riguarda il reset del contatore è stato implementato il metodo adibito a tale evento, infatti una volta che l'utente lo invocherà tramite ad esempio il click di un bottone, esso pubblicherà l'evento di reset del counter, il quale sarà inviato ai riflettori che lo marchieranno con il timestamp e successivamente lo invieranno a tutti i modelli della sessione attualmente connessi, che quindi azzereranno il loro contatore interno e notificheranno la View di questo cambiamento.

2.7.4 Counter Main

```
// CounterMain.js
import { CounterModel } from "./CounterModel"
import { CounterView } from "./CounterView"
import { Session } from "@croquet/croquet"

const apiKey = ""
const appId = ""
const sessionId = ""
const password = ""

Session.join({
    apiKey: apiKey,
    appId: appId,
    name: sessionId,
    password: password,
    model: CounterModel,
    view: CounterView
})
```

A questo punto non rimane che definire un punto di avvio della applicazione.

Il suo scopo sarà quindi quello di permettere di fare il Join di una sessione e richiede i seguenti dati[7]:

- **API Key**: ovvero la chiave di accesso alla rete di riflettori di Croquet.
- **App ID**: ovvero l'identificatore dell'applicazione che deve essere univoco e deve seguire le convenzioni di naming delle applicazioni android (com.example.myapp)
- **Name**: identifica la sessione relativa all'applicazione nel caso essa possa avere più sessioni.
- **Password**: utilizzata per crittografia end-to-end di tutti i dati in transito nella rete di Croquet per questa sessione.

Ora non resta che richiedere il join della sessione tramite il relativo metodo statico della classe Session, specificando i suddetti parametri e passando inoltre la classe principale del Model da istanziare e anche quella della View.

2.7.5 Counter Main Headless

```
// CounterMainHeadless.js
import { Session } from "@croquet/croquet"
import { CounterModel } from "./CounterModel.js"
import { CounterView } from "./CounterView.js"

const apiKey = ""
const appId = ""
const sessionId = ""
const password = ""

Session.join({
    apiKey: apiKey,
    appId: appId,
    name: sessionId,
    password: password,
    model: CounterModel,
    step: "manual"
}).then(({ id, step, model, view, leave }) => {
    setInterval(step, 100)
    view.subscribe("counter", "updated", (data) => {
        console.log("Counter updated: ", model.counter)
    })
})
```

Inoltre è possibile effettuare il join di una sessione in maniera headless tramite la specifica del parametro step a manual, andando così a sostituire il main loop predefinito di Croquet.

In questo caso non sarà necessario specificare la View e bisognerà richiamare la funzione di step manualmente per far si che la View riceva gli eventi. Se non si specifica una View ne verrà creata una vuota ma che sarà possibile sfruttare per interagire con il Model. Infatti nell'esempio proposto viene sfruttata la view vuota per sottoscriversi agli eventi di modifica del contatore e viene definito un handler a runtime che stamperà il valore del contatore al cambio del valore.

Capitolo 3

Croquet come Middleware: Idea

Si andrà di seguito ad analizzare come è stata sviluppata l'integrazione di modelli di applicazioni Croquet con altri linguaggi tramite la creazione di un middleware.

3.1 Obiettivo

Come già accennato l'obiettivo di questa tesi è stato quello di poter sfruttare l'architettura di Croquet con altri linguaggi che non siano Javascript. Quello che si è quindi dovuto capire è stato come implementare questa integrazione poiché Croquet ha un funzionamento molto particolare dato che il codice deve essere eseguito su una macchina virtuale Javascript. Quindi questo ci forza in ogni caso ad avere almeno una componente nel nostro sistema che sia scritta in JavaScript e che esegua del codice relativo a Croquet.

L'idea è quindi stata quella di creare un middleware che esegue il codice di Croquet e che però ne esponga le funzionalità a molteplici client. Quello che si voleva ottenere quindi era un sistema composto da un middleware che permettesse l'accesso a Croquet ad altre applicazioni che non siano scritte in JavaScript, in modo da poter creare un ambiente condiviso tra più entità software che comunicano e condividono tra loro dati ed eventi appoggiandosi sulla rete di Croquet.

3.2 Middleware

L'idea alla base dell'integrazione era quindi quella di sviluppare come già accennato un middleware che permettesse l'accesso al sistema di Croquet. Quello che quindi si è cercato di capire era come strutturare tale applicativo.

Dopo un analisi si è giunti alla soluzione che prevede una componente software che fungerà da middleware e la quale sarà scritta per forza in JavaScript e che si occuperà sia dell'esecuzione del modello scelto sia della comunicazioni con le entità esterne che richiedono l'accesso all'ambiente di Croquet.

La metodologia che quindi si è scelta per fornire l'accesso a Croquet e realizzare questo middleware è stata la creazione di un Proxy lato JavaScript che supportasse la comunicazione con più client e che potesse quindi fornire accesso a determinate API di Croquet.

Figura 3.1: Comunicazione Proxy-Client

.

3.3 Integrazione Model

Il primo approccio è stato quello di cercare di integrare completamente sia la parte di Model che di View su un altro linguaggio. Si cercava quindi di integrare anche lo sviluppo del Model nel linguaggio del client, e per fare ciò sono stati svolti vari esperimenti per architetture a livello concettuale ed anche prove pratiche tramite l'integrazione del Model in linguaggio Java.

L'obiettivo era quindi quello di poter sviluppare un applicazione Croquet completamente in Java senza dover passare da Javascript, tuttavia l'implementazione della libreria core di Croquet non è open-source e perciò non si poteva reperire il meccanismo e le API alla base del framework.

Di conseguenza si è cercato quindi di sviluppare un Model dinamico in Javascript la cui configurazione sarebbe stata determinata dal client che dopo essersi connesso avrebbe mandato a Javascript tutte le sue sottoscrizioni e i suoi comandi di future.

L'idea era quindi quella di realizzare un Model che fosse sempre sincronizzato tra Javascript e il linguaggio client mantenendo l'uguaglianza tramite messaggi scambiati tramite qualche tipo di IPC come da figura 3.2. Questi messaggi avrebbero incluso aggiornamento di dati, eventi, notifiche di esecuzioni di future, etc...

Tuttavia questo limitava di molto le funzionalità implementabili nel linguaggio client perché quello che quindi veniva replicato erano solamente le

sottoscrizioni e le schedulazioni future programmate alla creazione del Model (nella funzione di init) mentre quelli fatti a runtime non sarebbero stati gestibili.

Anche l'aggiunta di un componente che gestiva le richieste di sottoscrizioni ed eventi futuri a runtime si è rivelato fallimentare poiché al riavvio non si poteva stabilire lato client dove ci si era fermati prima della disconnessione.

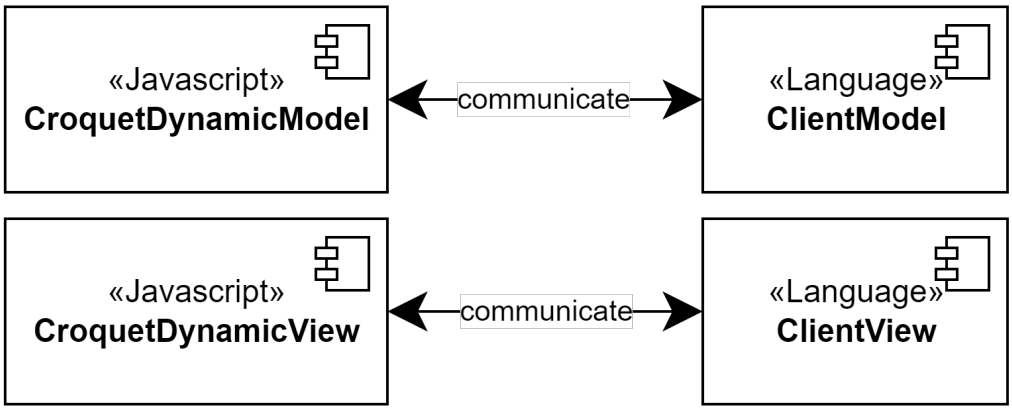

Figura 3.2: Idea integrazione Model sul client

.

L'integrazione del Model lato client si è quindi rivelata fallimentare e si è proceduto allo sviluppo di una procedura per l'integrazione della parte di View lato client in modo che essa possa avere sempre accesso a dati ed eventi.

3.4 Integrazione View

Scartata quindi l'idea del replicare l'intera struttura Model e View in un altro linguaggio si è iniziata una valutazione dell'integrazione solamente della parte di View. Essa infatti risulta molto più semplice da gestire, in primis non viene eseguita in una macchina virtuale e questo ci da libero accesso lato javascript alla sua creazione e alla sua interazione in ogni modo.

L'idea è quindi quella di creare una View di base che faccia da Proxy per accesso al modello tramite qualche tipo di protocollo IPC. In questo modo una volta instaurata la connessione con un client si potrebbe sviluppare una procedura di messaggi che permettono l'accesso a tutte le funzionalità della View anche dal client.

Inoltre questo tipo di architettura avrebbe permesso anche la gestione di più client, tra loro indipendenti, in un contesto locale come da figura 3.3.

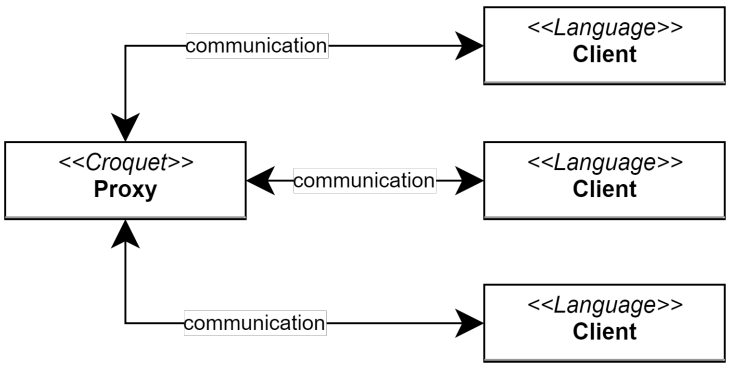

Figura 3.3: Comunicazione Proxy con client multipli

.

3.4.1 Analisi View Croquet

Bisogna comunque fare una riflessione su quello che è il concetto di View di Croquet, infatti chiamarla tale è abbastanza riduttivo poiché essa non si limita a rispondere a degli eventi per aggiornare un interfaccia utente e ad interagire con l'utente stesso, bensì tale View non ha neanche come requisito il dover essere associata ad un qualche tipo di interfaccia.

Si ha quindi che tale componente di Croquet ha si le funzionalità di una View, quindi ad esempio la sottoscrizione ad eventi per aggiornarsi in tempo reale ma ha anche funzionalità che sono esterne al comune concetto di View, quali l'accesso diretto ai dati del modello ma soprattutto la possibilità di pubblicare eventi che andranno ad influire lo stato del modello, avendo quindi una funzione seppur minima di controllo.

Analizzando quindi meglio il funzionamento e il livello di accesso e funzionalità che tale componente permette si può notare come essa renda accessibile praticamente tutto l'ambiente condiviso a chi la sta utilizzando.

Si è quindi scelto che una volta esportata questa componente sarebbe stata rinominata come un entità di tipo **Environment**, che come già detto quindi permette l'accesso sia allo stato dell'ambiente, sia agli eventi che vengono inviati in tale ambiente e sopratutto alla pubblicazione di eventi nel suddetto environment.

Si andrà quindi ad analizzare l'integrazione parlando di integrazione dell'Environment lato client.

3.4.2 Environment

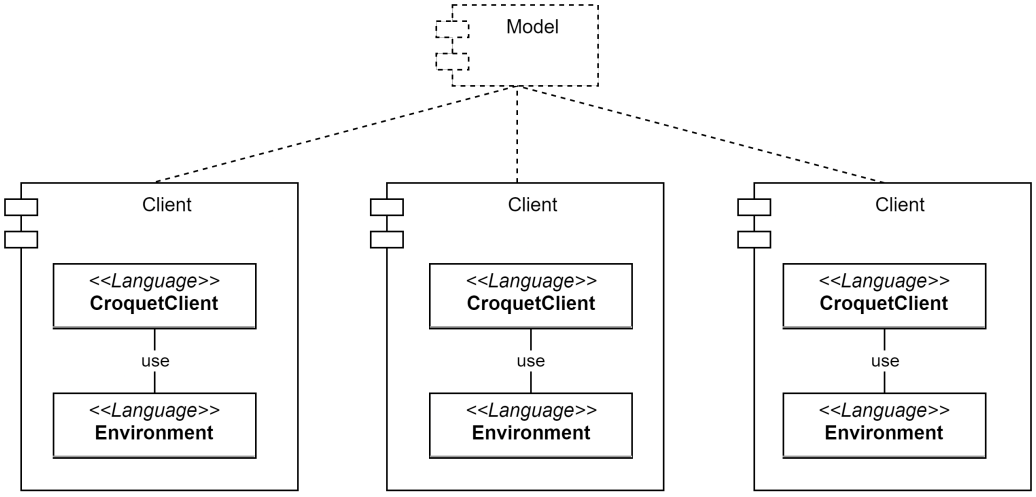

Figura 3.4: Funzionamento dal punto di vista dei client

.

Dalla precedente analisi si è giunti all'idea di non considerare la parte di View come tale ma più come un entità di tipo **Environment**.

Quindi l'idea dell'integrazione come da figura 3.4 era quella di creare in qualche modo un architettura che permettesse ai client di rendere trasparente il fatto che non fossero connessi al Model tramite qualche IPC ed esporre le funzionalità di Publish/Subscribe ed accesso ai dati tramite un Environment. Il quale si sarebbe servito dell'apposito client tramite il quale avrebbe mantenuto in sincronia sia dati che eventi.

Inoltre poiché Croquet basa il suo fondamento sul fatto che il Model ne risulta una macchina virtuale il cui scopo è simulare un unica macchina condivisa è possibile sfruttare questa caratteristica e notare che ogni client a questo punto sarebbe isolato come se fosse una singola istanza dell'applicazione comprendente sia Model che View.

3.4.3 Proxy

Quello che mancava era quindi una procedura di sincronizzazione tra JavaScript e il linguaggio del client. Si è optato quindi per la creazione di un Proxy in JavaScript che supportasse la connessione tramite protocollo di tipo IPC per comunicare in modo che il client potesse connettersi ad esso.

La struttura sarebbe stata quindi quella dove il Proxy JavaScript avrebbe mandato in esecuzione il Model (che andrà quindi programmato in Javascript utilizzando la libreria di Croquet) e avrebbe connesso ad esso una View di tipo dinamico che adatti le sue sottoscrizioni in base a quelle ricevute dai client tramite IPC.

L'idea viene espressa bene dalla figura 3.5.

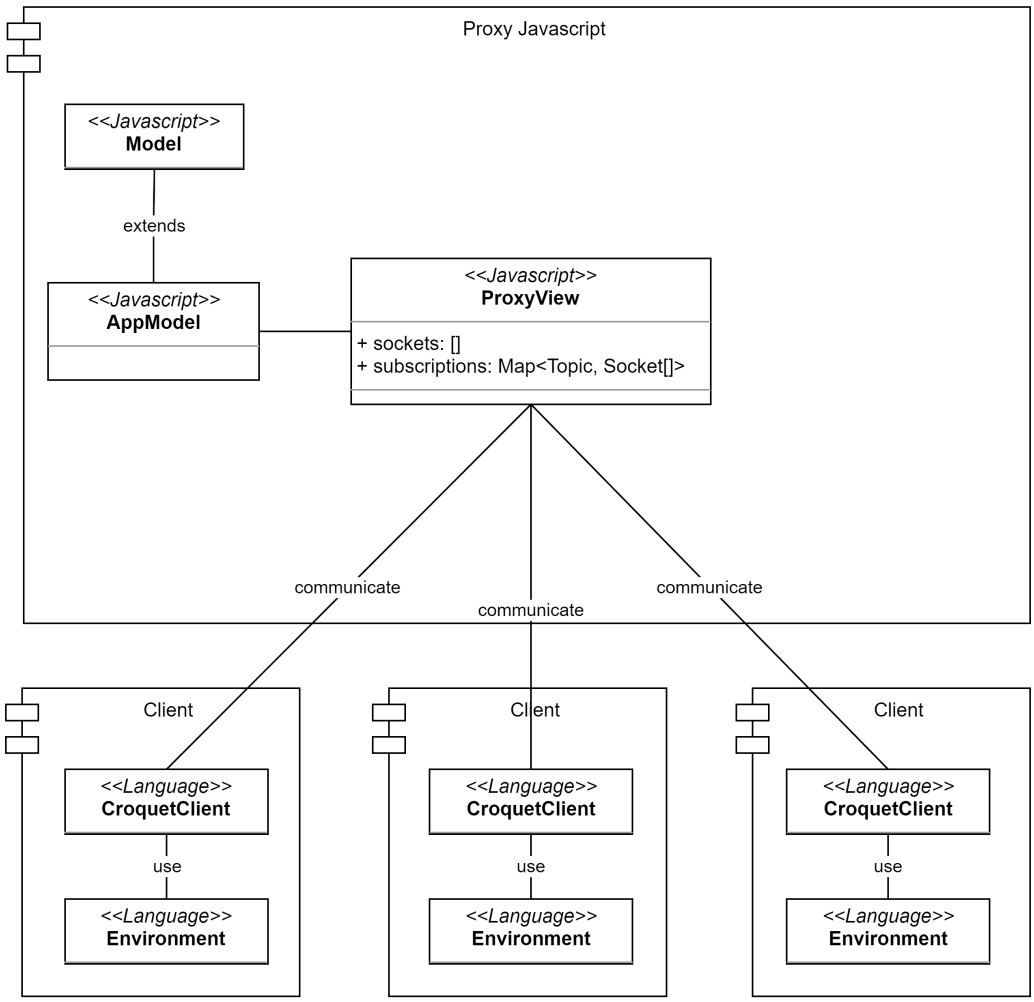

Figura 3.5: Funzionamento generale proxy-client

.

3.5 Protocollo

Si procede quindi ad analizzare cosa il protocollo di sincronizzazione richiedeva e come è stato deciso di implementarlo.
I problemi da risolvere erano principalmente 3:

- **Sincronizzazione Dati**: ovvero quando il client si connette al proxy deve poter ricevere lo stato attuale dei dati del Model per salvarselo internamente e quindi simulare di avere accesso diretto ai dati del Model che è in esecuzione su JavaScript.

- **Aggiornamento Dati**: ovvero quando il Model in JavaScript effettua delle modifiche allo stato queste modifiche devono essere in qualche modo comunicate al client che le deve replicare così da poter garantire il contenimento dello stato esatto del Model.

- **Gestione Eventi**: ovvero il client deve poter pubblicare eventi ma soprattutto deve poter sottoscriversi ad eventi e ricevere le notifiche quando questi eventi vengono pubblicati dal Model in modo da risultare reattivo ad essi.

Di seguito verrà spiegato più in dettaglio ognuno di questi problemi e come esso sia stato risolto.

3.5.1 Sincronizzazione Dati

Il primo problema era quello della sincronizzazione dei dati, ovvero del portare lo stato dei dati dal Model Javascript fino al client quando avviene una nuova connessione. Era quindi necessario trovare un formato di scambio dati che permettesse al Proxy di inviare lo stato attuale al client.

Si è optato quindi per utilizzare il formato JSON che permette un espressività e una leggibilità molto superiore a quella del concorrente XML. In questo modo alla connessione del client il proxy avrebbe dovuto fare un istantanea dello stato attuale dei dati per poi inviarlo in formato JSON al client, il quale avrebbe potuto ricostruire l'oggetto in memoria facendone un parsing.

Questo messaggio prende il nome di **data** in quello che sarà il protocollo completo di sincronizzazione e da come si può notare dalla figura 3.6 che mostra la procedura di connessione è preceduto da altri messaggi che sono:

- **connect**: ovvero l'apertura della connessione.

- **connection-ready**: inviato dal proxy una volta pronto ad accogliere il client che ha fatto richiesta.

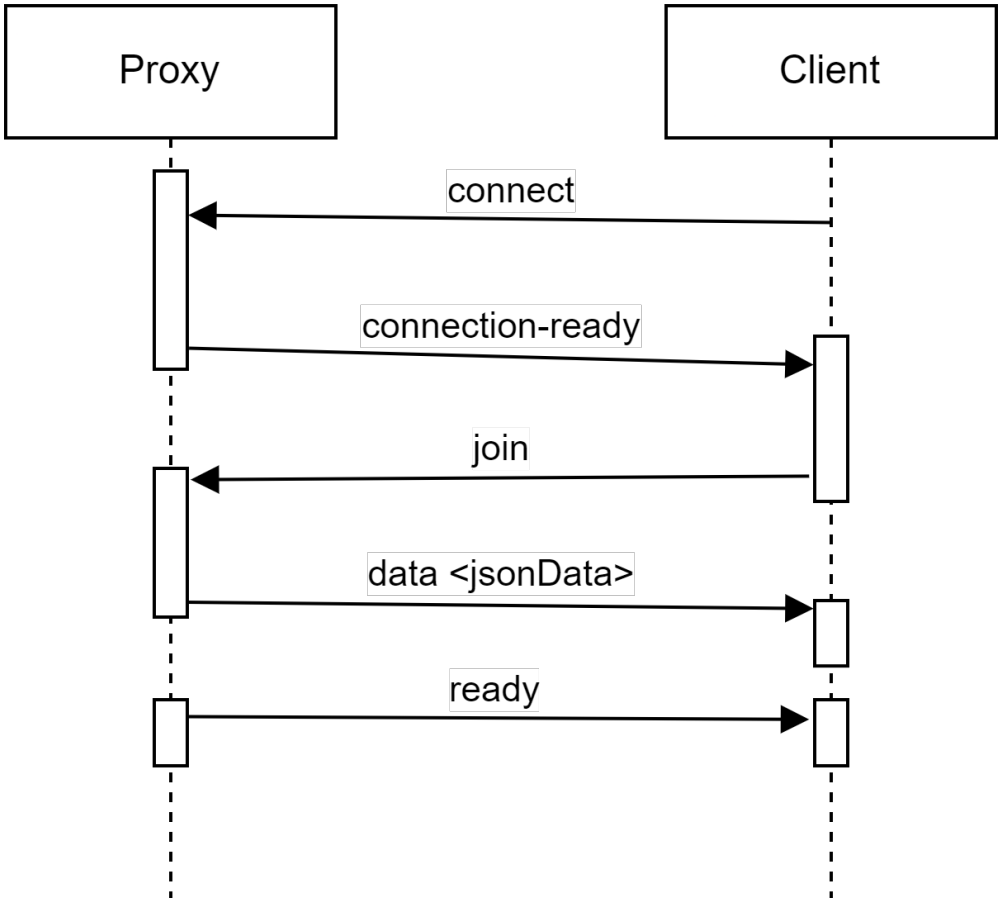

Figura 3.6: Procedura di connessione al Proxy

.

- **join**: inviato dal client per richiedere di partecipare.

Il messaggio è inoltre seguito da quello di **ready** che viene inviato dal proxy e segnala al client che ora è pronto a ricevere tutti i messaggi futuri del protocollo

3.5.2 Aggiornamento Dati

Un altro problema è stato quello di mantenere i dati uguali in ogni istante tra Javascript e il client, infatti le modifiche ai dati sono effettuate dal Model e quindi la ProxyView non può sapere quali sono.

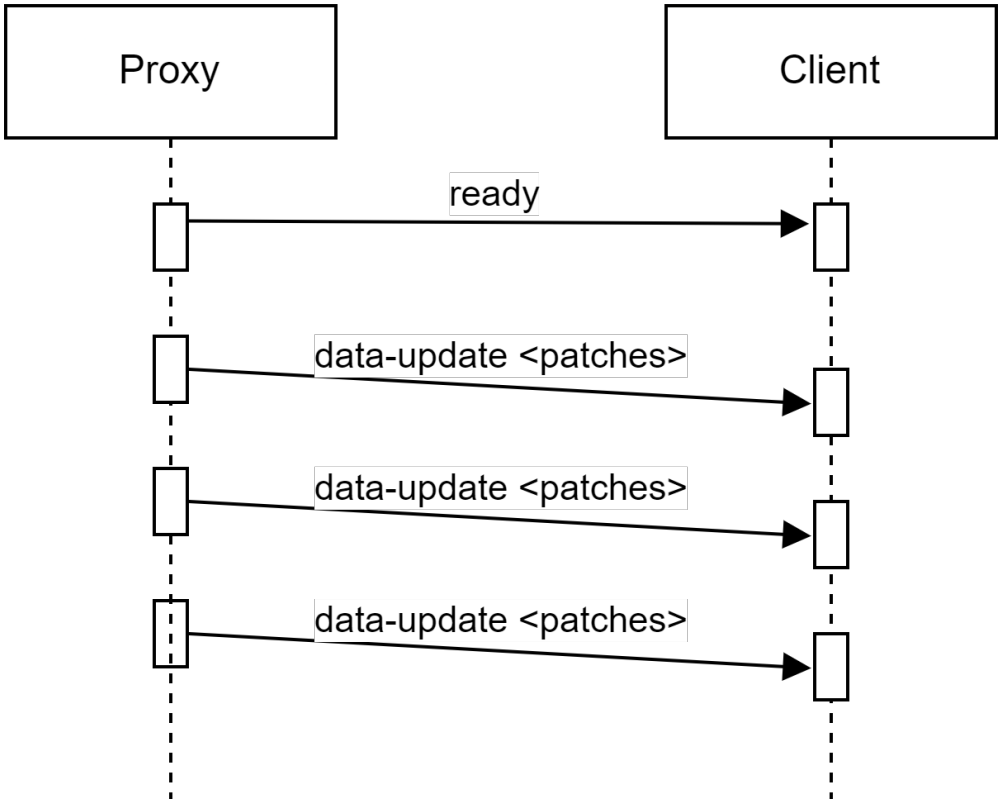

Figura 3.7: Procedura di aggiornamento dati

.

L'idea è stata quindi quella di sfruttare il fatto che la ProxyView abbia accesso diretto al model per poter creare un observer sui dati in modo da essere notificati quando avvengono delle modifiche su di essi.

A questo punto si è scelto di sfruttare il formato JsonPatch[17] che permette di serializzare qualsiasi operazioni di modifica su oggetti anche complessi. In questo modo la ProxyView nel momento in cui viene notificata di una modifica ai dati può estrapolarne le modifiche creando quindi delle JsonPatch per poi inviarle al client, il quale le processerà in modo da avere i dati aggiornati.

Questo tipo di messaggio prende il nome di **data-update** in quello che sarà il protocollo completo di sincronizzazione e viene inviato quindi in un qualsiasi momento da parte del Proxy come da figura 3.7.

3.5.3 Gestione Eventi

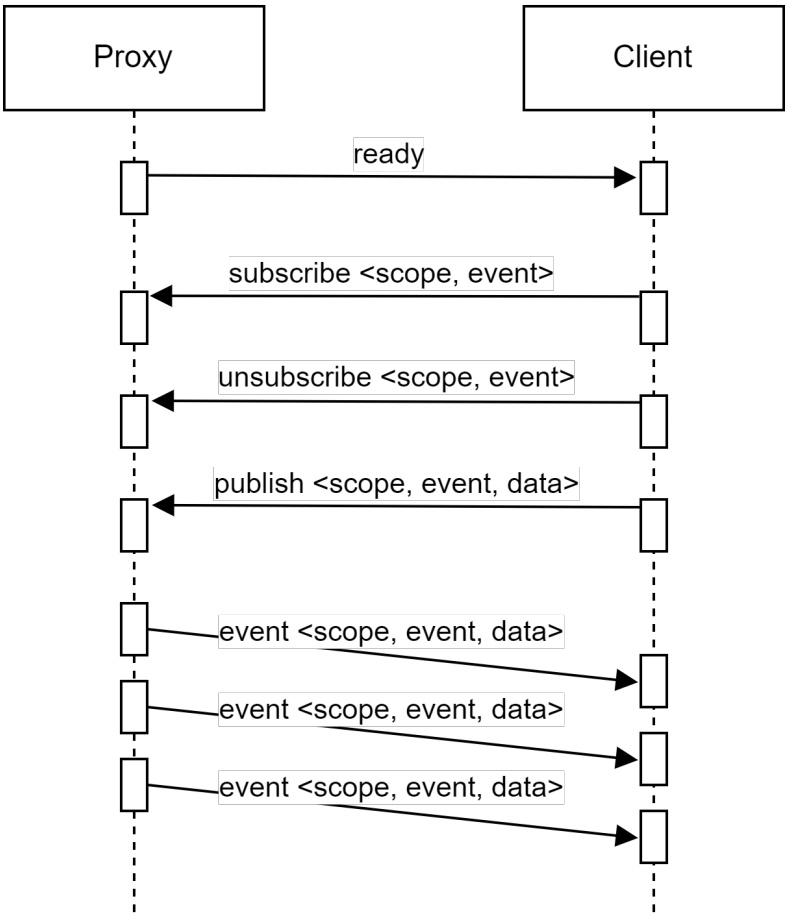

Figura 3.8: Procedura di gestione eventi

.

A questo punto l'ultimo problema da risolvere era la gestione degli eventi, sia dal punto di vista delle sottoscrizioni sia da quello delle pubblicazioni.

Pubblicazioni

Per la parte delle pubblicazioni è stato molto semplice, quando il client deve effettuare una publish allora viene inviato un messaggio con lo scope e l'event e una volta che la ProxyView l'ha ricevuto lo pubblica a sua volta lei stessa, in modo che arrivi ai riflettori e che venga successivamente ricevuta da tutti i Model della sessione.

Questo tipo di messaggio prende il nome di **publish** in quello che sarà il protocollo completo di sincronizzazione.

Sottoscrizioni

La gestione delle sottoscrizioni risulta leggermente più complessa. Quello che deve fare il client è inviare un messaggio alla ProxyView specificando a quale scope ed event sottoscriversi e salvarsi l'handler da eseguire all'arrivo di quell'evento in una mappa che correla dei canali (scope + event) alla funzione di handle di tale evento.

Questo messaggio prende il nome di **subscribe** in quello che sarà il protocollo completo di sincronizzazione.

Rimuovere Sottoscrizioni

Ovviamente è disponibile anche la possibilità di annullare una sottoscrizione e per effettuarla basta che il client invii il relativo messaggio specificando lo scope e l'evento da cui rimuovere la sottoscrizione.

Questo messaggio prende il nome di **unsubscribe** in quello che sarà il protocollo completo di sincronizzazione.

Eventi

A questo punto rimane solo la gestione degli eventi dal punto di vista della ProxyView, che quando riceverà dei messaggi di subscribe nel caso non sia sottoscritta a quel canale provvederà a farlo.

Si salverà in oltre una mappa di canali che per ognuno di essi si salva quali client sono sottoscritti ad essi. In questo modo quando arriverà un evento potrà controllare quali client siano connessi e inoltrargli l'evento tramite un apposito messaggio che prende il nome di **event** in quello che sarà il protocollo completo di sincronizzazione.

Per la gestione della rimozione delle sottoscrizione molto semplicemente viene rimosso il client dalla lista dei client che sono sottoscritti a quel determinato canale e nel caso in cui non ci siano più client ad ascoltare quel determinato evento allora viene rimossa definitivamente la sottoscrizione chiamando il metodo unsubscribe delle API di Croquet.

3.6 Analisi Protocollo

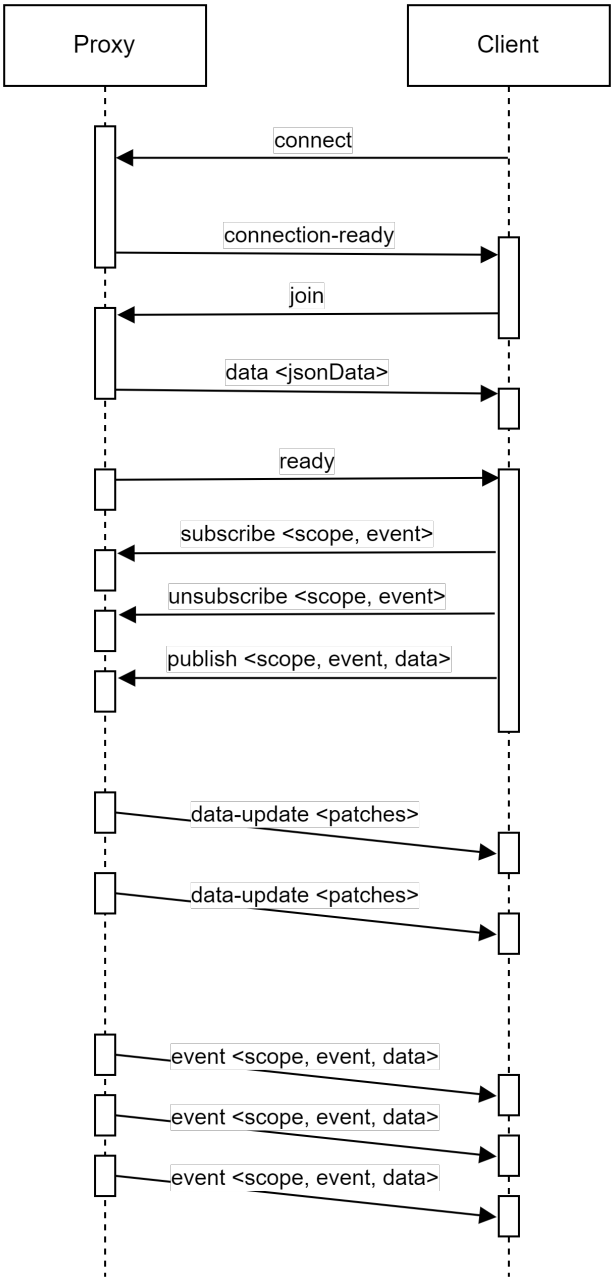

Figura 3.9: Protocollo completo

.

Dopo aver quindi spiegato e risolto tutte le problematiche relative alla sincronizzazione tra la parte di Javascript e quella del client, mostrando quindi tutti i tipi di messaggi che vengono scambiati per ottenerla, si procede ad analizzare le possibilità offerte.

3.6.1 Possibilità

Per quanto riguarda le funzionalità che l'Environment costruito nel linguaggio client espone, esse sono esattamente le stesse che scrivere una View direttamente in Javascript. Si ha quindi una garanzia di funzionamento di tutte le API fondamentali che le View di Croquet hanno a disposizione. Infatti non si avrà nessun problema a sottoscriversi ad eventi registrando ad essi delle callback, le quali saranno invocate nell'esatto momento in cui le riceverà la ProxyView su Javascript, la quale provvederà ad inoltrarle istantaneamente a tutti i client.

Anche le funzionalità di pubblicazione sono assicurate poiché basta inviare un messaggio alla ProxyView che provvede a pubblicarlo immediatamente.

Infine anche i dati è stato appurato che rimangano sempre sincronizzati grazie all'invio dello stato iniziale e di tutte le patch a runtime.

A questo punto l'unica differenza tra avere una View direttamente in Javascript o sfruttare la procedura di sincronizzazione implementando un client in qualsiasi linguaggio sarà semplicemente la latenza della comunicazione tra i due processi, che essendo entrambi locali sarà praticamente 0.

3.6.2 Isolamento

Inoltre come già accennato in precedenza anche con la figura 3.4 l'obiettivo che è stato raggiunto è di isolare completamente la parte relativa alla View di Croquet nei client, trasformandola opportunamente in un oggetto di tipo Environment che non è a conoscenza della provenienza dei dati e degli eventi, ma che semplicemente sfrutterà il suo client interno che implementa la procedura di sincronizzazione per mantenere congruo lo stato del modello e per gestire gli eventi.

Infatti per ognuno dei client la sorgente di accesso al modello potrebbe essere un istanza di Model JavaScript diversa, dato che per come è strutturato Croquet ogni singola istanza del Model ne risulta perfettamente uguale alle altre, ed è quindi indistinguibile capire se ognuno di questi client sia connesso allo stesso modello o meno.

Quindi come conclusione si ha che tutti i possibili client locali non avranno ripercussioni dal fatto che siano tutte View, opportunamente ristrutturate, di una stessa istanza del Model.

3.6.3 Multi-Client locale

Considerando quindi il suddetto isolamento garantito lato client si può quindi generalizzare ancora di più le possibilità offerte dalla libreria client. Infatti essa potrebbe essere sfruttata per generare un qualsiasi numero di entità che interagiscono con il mondo di Croquet come entità autonome effettuando operazioni che non sono dipendenti dall'utente.

Non si ha infatti la necessità che sia l'utente stesso a interagire come client poiché si è giunti alla conclusione che il concetto di View espresso da Croquet è un qualcosa di più astratto e generico.

Si può ad esempio avere componenti che gestiscano delle entità del mondo condiviso in modo autonomo, utilizzando intelligenza artificiale in modo da creare delle entità con cui gli utenti reali possono interagire senza dover svolgere la computazione dell'intelligenza artificiale in locale ma bensì comunicando con questi agenti che altro non sono che altri client.

Per concludere quindi ora si ha la possibilità di sviluppare tali componenti senza per forza doversi appoggiare sul linguaggio JavaScript.

3.7 Modello di sviluppo

Si passa ora alla spiegazione del come funzionerebbe lo sviluppo di applicazioni che utilizzano tale middleware.

Quello che un utente dovrebbe fare è quindi innanzitutto lavorare allo sviluppo della parte di Model in JavaScript e poi mandarlo in esecuzione attaccando ad esso la View che funge da Proxy e che quindi rimarrà in ascolto per le connessioni dei client.

Una volta sviluppata la parte del Model essa sarà quindi disponibile per l'accesso ad ogni tipo di client.

Servirà quindi sviluppare la parte client, che si appoggerà su un sdk che esporrà come oggetto quello di tipo Environment che è stato introdotto e spiegato nelle sezioni precedenti. Questo oggetto Environment una volta inizializzato si troverà in uno stato di connessione al Proxy che garantirà l'accesso al modello di Croquet ma che rimarrà invisibile all'utente.

Quello che quindi un utente potrà fare lato client una volta inizializzato l'Environment è utilizzarlo per la gestione degli eventi, che concerne sia la sottoscrizione e le rimozione delle sottoscrizioni, sia la pubblicazione di eventi. Ovviamente viene garantito anche l'accesso ai dati del modello che risulta una parte fondamentale per lo sviluppo di tali applicazioni.

Questa metodologia astrae quindi i dettagli implementativi che il client utilizza, rendendolo quindi indipendente dalla struttura di tale componente.

Infatti si potranno avere architetture software di qualsiasi tipo, le quali semplicemente avranno a disposizione l'accesso all'ambiente condiviso tramite le API offerte dall'Environment.

Capitolo 4

Implementazione Middleware

Si procede quindi spiegando come è stata implementata la struttura del Middleware in Javascript, costruita in modo da esporre tramite Socket l'accesso al sistema di Croquet e il cui compito è anche quello di gestire l'esecuzione del modello.

Il tutto è stato strutturato in modo da poter gestire anche la presenza di più classi di Model per quanto riguarda la parte di JavaScript evitando di averne solamente una che col crescere della grandezza dell'applicazione rischia di diventare incomprensibile. Si può avere quindi la possibilità di modularizzare un applicazione in più sotto-modelli, come consigliato anche da Croquet stesso.

Si andrà quindi di seguito ad analizzare come sia stato sviluppata la parte JavaScript, ovvero quella responsabile dell'esecuzione del modello e che farà da middleware per l'accesso a Croquet per tutti i client.

4.1 Tecnologie Utilizzate

Prima di procedere ad illustrare l'implementazione del protocollo lato Javascript verranno mostrate le librerie utilizzate per implementarlo, spiegandone l'utilità che hanno avuto e quali problemi sono stati risolti grazie ad esse.

4.1.1 Node.js

Node.js è un runtime JavaScript costruito sul motore JavaScript V8 di Chrome[18]. Esso permette l'esecuzione di applicazione JavaScript in ambienti server invece che all'interno del browser. Risulta quindi possibile utilizzarlo per l'esecuzione di codice JavaScript da riga di comando, non dovendo quindi essere eseguito per forza in un browser, cosa che impedirebbe l'accesso da altri processi.

Nel caso del middleware di Croquet è stato utilizzando in modo che esso venga eseguito tramite tale software in modo che venga creato un processo a se stante che si occupi della sua esecuzione e che lo renda quindi accessibile da altri processi, che saranno quindi i processi client che vogliono usufruire di tale middleware.

4.1.2 Socket.io

Per la parte di comunicazione tra Javascript e il client è stata scelta la libreria **Socket.io** che permette un canale di comunicazione bidirezionale basato su eventi ed implementato tramite le Websocket[23].

La libreria offre molti vantaggi, innanzitutto garanzie a livello di connessione, dove nel caso cada il collegamento verrà ristabilito con una politica di retry, inoltre supporta anche altri protocolli di fallback nel caso Websocket non sia disponibile (ad esempio usando polling HTTP).

Un ulteriore vantaggio che tale libreria offre è la compatibilità con le librerie che fungono da client di Socket.io per il protocollo, infatti ne esistono praticamente per ogni linguaggio di programmazione e in questo modo è possibile scrivere l'implementazione del client molto facilmente utilizzando suddette librerie.

4.1.3 Fast-Json-Patch

Come libreria per implementare l'osservazione dei dati del Model e ricavarne le JsonPatch di modifica si è optato per **fast-json-patch**[25] che offre prestazioni di livello superiore alle concorrenti.

L'unica problematica incontrata con questa libreria è che promette di poter osservare dei dati e definire una callback opzionale da triggherare ad ogni cambiamento, tuttavia questa callback, se definita, non viene mai avviata. Questo risulta tuttavia un problema già noto agli sviluppatori ma ancora in corso di risoluzione[13].

Per ovviare a questo problema è stato scelto che invece che inviare i messaggi di **data-update** ai client nel momento in cui cambiavano i dati, si è scelto di inviarli prima dell'invio degli eventi. In questo modo poiché i client reagiscono solamente agli eventi e non sono pensati per eseguire il polling dei dati, quando dovranno ricevere un evento riceveranno prima le patch dei dati e poi l'evento stesso.

A questo punto quando eseguiranno l'handler dell'evento si troveranno in un contesto in cui le patch sono già state ricevute ed applicate e quindi si hanno i dati aggiornati.

4.2 Architettura Model

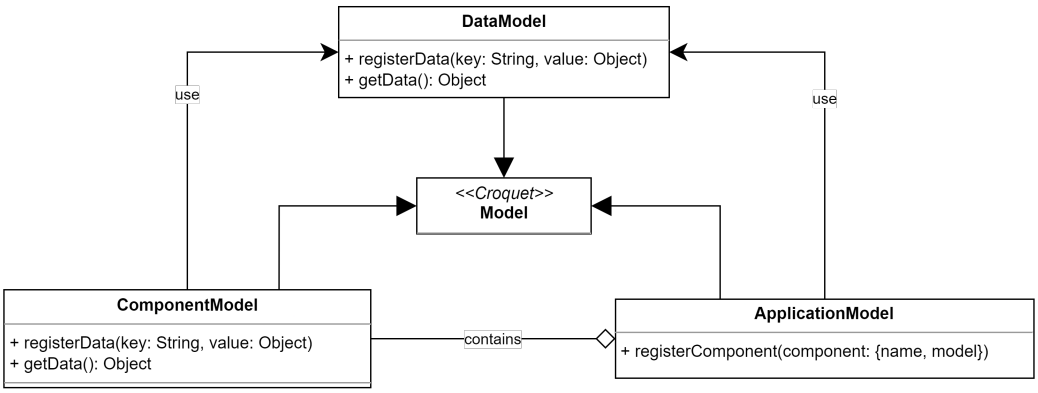

Figura 4.1: Architettura del Model Javascript

.

Si procede quindi ora a descrivere come è stata gestita l'architettura a livello di Model lato Javascript, la quale è illustrata dalla figura 4.1.
In particolare essa è stata divisa in 3 componenti:

- **DataModel**: che si occuperà del mantenimento e della gestione dei dati che costituiscono lo stato dell'applicazione, in modo da avere una sorgente unica di accesso a tali dati.

- **ComponentModel**: che sarà l'entità fondamentale che andrà estesa per generare una componente del modello, che potrà essere indipendente o meno dalle altre.

- **ApplicationModel**: che altro non è che la componente del modello principale che quindi dovrà registrare tutte le sotto-componenti in modo che quando verrà avviata sarà a sua volta in grado di mettere in esecuzioni tutti i modelli sviluppati dall'utente.

4.2.1 DataModel

```
// DataModel.js
import { Model } from "@croquet/croquet"
export class DataModel extends Model {

    init() {
        this.data = {}
    }

    addData(key, value) {
        this.data[key] = value
    }

    getData() {
        return this.data
    }
}

DataModel.register("DataModel")
```

La classe **DataModel** è una componente Model completamente gestita dal middleware il cui scopo è quello di essere l'unica sorgente dei dati, ovvero l'unico oggetto che contiene l'intero stato dell'applicazione, in questo modo non essendoci dati sparsi in vari componenti del Model si può facilitare l'osservazione dei cambiamenti dello stato del modello.

Essa infatti contiene solamente un campo *data* che sarà un oggetto che conterrà a sua volta tutti gli oggetti che i vari componenti avranno bisogno di salvare.

La funzionalità di aggiunta dati è esposta tramite il metodo *addData* che permette di aggiungere un dato specificandone le chiave e il valore iniziale.

Successivamente per le modifiche basterà chiamare invece il metodo *getData*, che restituirà l'oggetto completo e sul quale si potrà quindi effettuare le modifiche agli oggetti interni.

Poiché ogni classe che estende Model deve essere registrata per poterla serializzare ai fini di snapshotting e dato che questa classe non viene utilizzata direttamente dall'utente si è optato per effettuare la registrazione direttamente dentro il suo file in modo che venga eseguita quando viene importata dalle altre classi del progetto.

4.2.2 ComponentModel

```
// ComponentModel.js
import { Model } from "@croquet/croquet";

export class ComponentModel extends Model {

    init(options) {
        super.init(options);
        this.dataModel = options.dataModel
    }

    registerData(key, value) {
        this.dataModel.addData(key, value)
    }

    registerData(dict) {
        Object.keys(dict).forEach(key => {
            this.dataModel.addData(key, dict[key])
        })
    }

    getData() {
        return this.dataModel.getData()
    }
}
```

Continuando si giunge quindi al ComponentModel, che sarà la classe che andrà estesa per creare i vari componenti del modello.

Il suo scopo è quello di facilitare l'accesso ai dati contenuti nel DataModel che gli viene passato come parametro di inizializzazione alla sua creazione e ne ri-espone le API tramite i metodi *registerData* e *getData* che rispecchiano le API esposte dal DataModel ma opportunamente rinominate, poiché semanticamente è più corretto dire che una componente registri i suoi dati nel modello piuttosto che aggiungerli.

Per semplificare la registrazione di dati complessi viene anche offerta la possibilità di registrare direttamente un oggetto intero che verrà scomposto e verrà registrata ogni coppia chiave-valore singolarmente.

Inoltre esponendo questi metodi per l'accesso ai dati esso incapsula e nasconde il fatto che al suo interno sia contenuto il rifermento al DataModel, che risulterà quindi invisibile a chi sviluppa.

4.2.3 ApplicationModel

```
// ApplicationModel.js
import { Model, Session } from "@croquet/croquet";
import { DataModel } from "./data-model.mjs";
import { ProxyView } from "./proxy-view.mjs";
import { Server } from "socket.io";

export class ApplicationModel extends Model {

    init(options) {
        super.init(options);
        this.dataModel = DataModel.create()
    }

    registerComponent(component) {
        component.model.register(component.name)
        component.model.create({ dataModel: this.dataModel })
    }

    static setupSocket(socket, view) {
        socket.on('join', () => {
            view.addSocket(socket)
        })
        socket.on('disconnect', () => {
            view.removeSocket(socket)
        })
        socket.emit("connection-ready")
    }

    static startProxy(options = {}) {
        this.register(this.prototype.constructor.name)
        this.joinSession(options).then(({ view, step }) => {
            setInterval(step, 100)
            let server = this.createServer(options.port || 3000)
            server.on("connection", (socket) => {
                setupSocket(socket, view)
            })
        });
    }
}

ApplicationModel.register("ApplicationModel")
```

Infine si ha la classe **ApplicationModel** che serve a creare il Model principale che sarà poi mandato in esecuzione. Essa permette la registrazione dei vari componenti, ovvero delle classi sviluppate dall'utente che estendono **ComponentModel**, tramite il metodo *registerComponent*.

A questo metodo andrà passato come parametro un oggetto che specifichi le seguenti informazioni:

- **name**: ovvero il nome con cui registrare il componente (poiché la minificazione del codice potrebbe cambiarlo).

- **model**: ovvero la classe del componente.

Tale metodo quindi provvederà sia a registrare la classe per la serializzazione sia a crearne l'istanza che andrà in esecuzione.

La suddetta classe inoltre si occuperà di inizializzare il DataModel e di passarne il riferimento a tutti i componenti che si registrano.

Tale classe, dispone in ultimo di un metodo statico startProxy che permette l'avvio del proxy, che andrà chiamato sulla classe creata estendendola, il quale si occuperà di registrare l'ApplicationModel creato dall'utente e farà il join della sessione, inizializzando la ProxyView.

Infine si metterà in ascolto sulla porta specificata in modo da intercettare le nuove connessioni e passarle alla View, ed inoltre farà il setup per l'handle dei messaggi relativi alla procedura di connessione quali il messaggio di **join** e quello di disconnessione.

Una volta effettuato questo setup verrà quindi inviato il messaggio di **connection-ready** così che il client sappia che ora può iniziare ad inviare i messaggi della procedura di connessione.

Anche in questo caso la registrazione di tale classe viene effettuata all'interno del file, in modo che Croquet sappia come serializzarla e rendendo l'utente che sviluppa utilizzando tale framework indipendente da tale registrazione

4.3 Architettura View

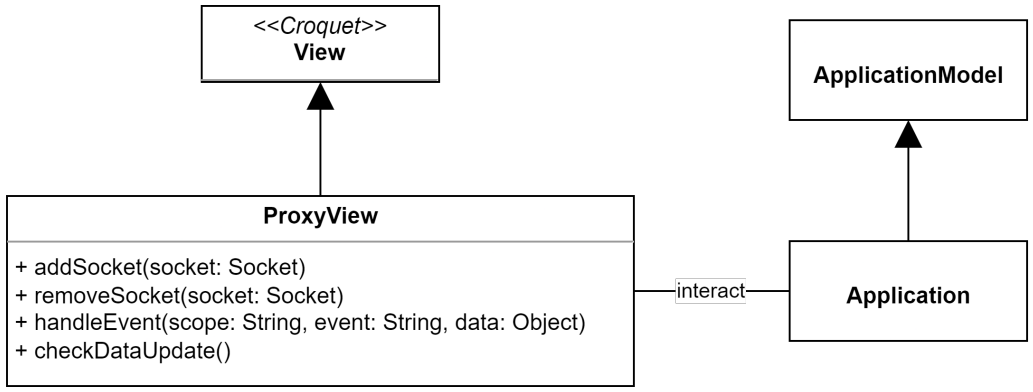

Figura 4.2: Architettura della View Javascript

.

Si è giunti quindi ora a parlare dell'architettura che concerne la parte di View dal lato Javascript, ovvero quella che fungerà da Proxy verso i client. Essa interagirà con il Model dell'applicazione effettiva e comunicherà con i client seguendo i meccanismi spiegati per la procedura di sincronizzazione.

Avrà quindi un ruolo fondamentale per far si che il client abbia accesso all'ambiente condiviso.

4.3.1 ProxyView

In questa componente si ha quindi la classe **ProxyView**, la quale si occuperà di gestire tutte le socket connesse al middleware, ovvero tutti i client in esecuzione, e implementerà il protocollo di sincronizzazione spiegato nel capitolo precedente.

Come primo elemento per la View, poiché bisogna tenere in considerazione quali client sono sottoscritti ad un determinato evento, è stata definita una classe **Channel** che rappresenta il canale dell'evento, ovvero l'insieme dei dati che identificano una sottoscrizione, che nel caso di Croquet è dato dalla coppia *scope* ed *event*.

Inoltre è stato definito un membro key dentro la classe del canale per far si che esso sia univoco e possa essere utilizzato come indice di una mappa, la quale come già anticipato si occuperà di correlare ogni canale all'insieme di client che sono sottoscritti a quel determinato canale.

Tale classe ha un implementazione banale e viene quindi omessa la sua implementazione.

```
// ProxyView.js
import { View } from "@croquet/croquet"
import jsonpatch from 'fast-json-patch';
import { Channel } from "./Channel.js";

export class ProxyView extends View {

    constructor(model) {
        super(model)
        this.model = model
        this.observer = jsonpatch.observe(this.model.dataModel.data)
        this.subscriptions = {}
    }

    addSocket(socket) {
        socket.emit("data", this.model.dataModel.data)
        socket.on("subscribe", (scope, event) => {
            this.registerSocketToSubscription(new Channel(scope,
                event), socket)
        })
        socket.on("unsubscribe", (scope, event) => {
            this.removeSubscription(new Channel(scope, event), socket)
        })
        socket.on("publish", (scope, event, data) => {
            this.publish(scope, event, data)
        })
        socket.emit("ready")
    }

    removeSocket(socket) {
        this.removeAllSocketSubscriptions(socket)
    }

    handleEvent(scope, event, data) {
        if (this.isThereDataUpdate()) {
            this.notifyDataUpdateToSockets()
        }
        this.notifyEventToSubscribedSockets(scope, event, data)
    }

    createSubscription(subscription) {
        this.subscriptions[subscription.key] = []
        this.subscribe(subscription.scope, subscription.event,
```

```
            (data) => this.handleEvent(subscription.scope,
                subscription.event, data)
        )
    }

    addSocketToSubscription(subscription, socket) {
        this.subscriptions[subscription.key].push(socket)
    }

    isAlreadySubscribed(subscription) {
        return this.subscriptions[subscription.key] !== undefined
    }

    removeSubscription(subscription, socket) {
        const index =
            this.subscriptions[subscription.key].indexOf(socket)
        if (index > -1) {
            this.subscriptions[subscription.key].splice(index, 1)
            if (this.subscriptions[subscription.key].length == 0) {
                delete this.subscriptions[subscription.key]
                this.unsubscribe(subscription.scope,
                    subscription.event)
            }
        }
    }

    removeAllSocketSubscriptions(socket) {
        for (const key in this.subscriptions) {
            this.removeSubscription(new Channel(key), socket)
        }
    }

    registerSocketToSubscription(subscription, socket) {
        if (!this.isAlreadySubscribed(subscription)) {
            this.createSubscription(subscription)
        }
        this.addSocketToSubscription(subscription, socket)
    }

    notifyEventToSubscribedSockets(scope, event, data) {
        const channel = new Channel(scope, event)
        if (this.subscriptions[channel.key] !== undefined) {
            this.subscriptions[channel.key].forEach(socket => {
                socket.emit("event", scope, event, data)
```

```
            })
        }
    }

    isThereDataUpdate() {
        return this.observer.length > 0
    }

    notifyDataUpdateToSockets() {
        const patches = jsonpatch.generate(this.observer)
        this.sockets.forEach(socket => {
            socket.emit("data-update", patches)
        })
    }

}
```

In tale classe, si può notare come alla sua costruzione essa crei un observer sui dati del modello, in modo da avere un controllo sulle modifiche che vengano fatte su di esso per poterle poi esportare in modo da inviarle ai client.

Il metodo principale di questa classe è quello di *addSocket*, che viene richiamato quando c'è una nuovo client che si vuole connettere ed in particolare viene chiamato dopo che il client ha inviato il messaggio di **join**.

Essa costituisce il fulcro della procedura di sincronizzazione, infatti inizialmente invierà alla socket lo stato attuale dei dati, poi farà il setup dei vari messaggi che può ricevere dal client (subscribe, unsubscribe, publish) e infine invierà il comando di **ready** per notificare il client che è pronto ad accettare tutti i messaggi del protocollo cosi che esso possa iniziare a sottoscriversi ad eventi ed a pubblicarne.

Altro elemento importante della classe è il metodo di *handleEvent*, che viene richiamato ogni qual volta venga ricevuto un evento a cui almeno un client è sottoscritto, che lo analizzerà e lo ridirezionerà a tutti client che si sono sottoscritti ad esso.

Inoltre prima di inviare un determinato evento occorre controllare se ci sono state modifiche ai dati, e nel caso vengano rilevate allora vengono notificate a tutti i client connessi tramite il messaggio di **data-update**. In questo modo prima di ricevere un evento riceveranno tutti quanti le patch di aggiornamento dei dati, cosicché nel momento in cui verranno notificati dell'evento a cui sono sottoscritti si troveranno di una situazione dove non sono presenti incongruenze dei dati.

Si è cosi ottenuta una perfetta sincronia dello stato del modello tra JavaScript e il client.

Capitolo 5

Implementazione Client

Si è quindi giunti all'implementazione di un client, per il quale è stato scelto il linguaggio Java, ma che tuttavia non risulta vincolante poiché se si vuole utilizzare un qualsiasi altro linguaggio basta implementarne la parte di Environment che utilizzi un apposito client del Middleware, la cui procedura, che è quella spiegata nel capitolo precedente, risulta molto semplice da implementare dato che si tratta solo della gestione di pochi tipi di eventi che vengono trasmessi, risultando quindi anche molto efficiente dal punto di vista computazionale.

Quello che ne risulta è quindi che quello che si andrà ora a mostrare è soltanto l'esempio che riguarda l'integrazione con il linguaggio Java.

Di seguito si procederà quindi descrivendo le componenti principali che costituiscono la libreria sviluppata per il client, la cui struttura completa è illustrata dalla figura 5.1.

Come già introdotto in precedenza quello che viene esportato nel client è l'entità **Environment**, che deve quindi avere accesso sia ai dati dell'ambiente condiviso, ovvero agli **Environment Data**, sia la possibilità di sottoscrizione e pubblicazione di eventi in tale ambiente.

5.1 Tecnologie utilizzate

Prima di procedere ad illustrare l'implementazione del protocollo lato Java, ovvero client, e della costruzione, attorno ad esso, del modello di sviluppo ad Environment, verranno mostrate le librerie e tecnologie utilizzate per implementarlo, spiegandone l'utilità che hanno avuto e quali problemi sono stati risolti grazie ad esse.

5.1.1 Socket.io-client

Per connettersi al Proxy, il quale è stato sviluppato utilizzando la libreria Socket.io per ascoltare le connessioni è stato quindi necessario utilizzare la stessa libreria però dedicata al ruolo di client. Essa infatti permette di connettersi ad un server sviluppato con Socket.io e quindi di utilizzare la stessa metodologia di sviluppo basata su eventi[14].

5.1.2 Jackson

Per quanto riguarda la gestione dei dati in JSON e quindi il relativo parsing ed interpretazione si è scelto di utilizzare la libreria **Jackson**, che risulta una delle migliori per quanto riguarda il parsing di JSON in ambito Java[11].

Infatti essa ha reso possibile la ricostruzione di oggetti a partire dalla loro rappresentazione in formato JSON ma anche tutte le operazioni di conversione da oggetti a JSON.

5.1.3 Json-Patch

Per applicare le JsonPatch che vengono ricevute dal Middleware durante l'esecuzione è stata utilizzata la libreria **json-patch** del pacchetto **java-json-tools** che ha quindi permesso di applicare le modifiche su oggetti Java a partire dalla specifica JsonPatch[12].

Tale libreria si è quindi rivelata utile per l'implementazione della parte del protocollo che riguardi i messaggi di **data-update**, ovvero quelli che contengono le Json Patches da applicare allo stato attuale per essere congruo con quello in esecuzione nel middleware.

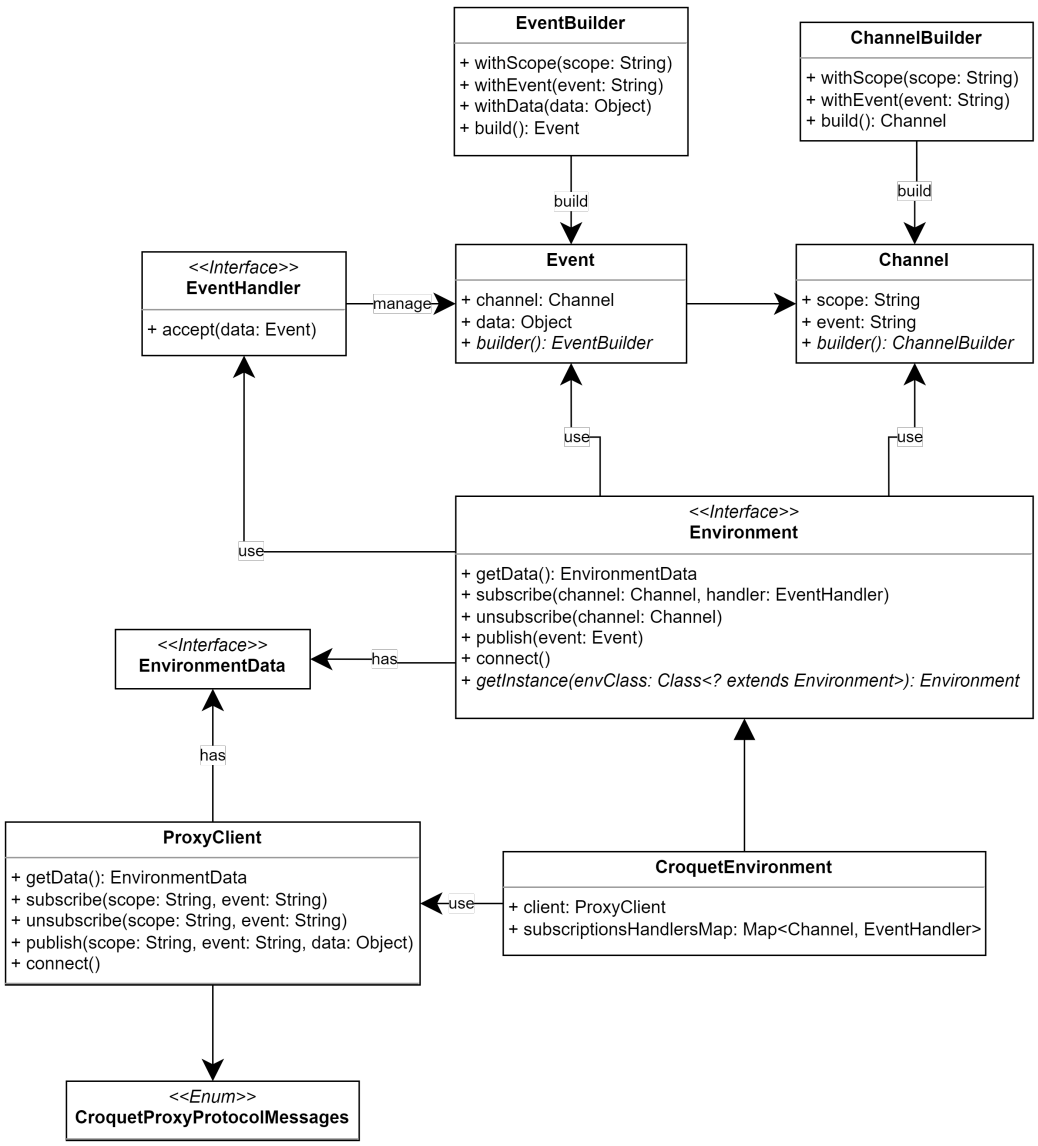

Figura 5.1: Architettura del client

.

5.2 Environment Data

```
public interface EnvironmentData {

    ObjectMapper mapper = new ObjectMapper();

    static <T extends EnvironmentData> T loadFromJson(String
        jsonString, Class<T> clazz) {
        return mapper.readValue(jsonString, clazz);
    }

    static <D extends EnvironmentData> D applyPatch(D data, String
        jsonPatch) {
        JsonPatch patch = mapper.readValue(jsonPatch,
            JsonPatch.class);
        JsonNode result = patch.apply(mapper.convertValue(data,
            JsonNode.class));
        return (D) mapper.treeToValue(result, data.getClass());
    }

}
```

L'interfaccia **EnvironmentData** è quella che si occupa della gestione dei dati che costituiscono lo stato del modello. Sarà infatti quella da implementare per creare una sorgente dati che deve rispecchiare quella sul Model sviluppato in Javascript.

Una volta implementata infatti andranno inseriti come campi pubblici tutte le variabili che vanno a comporre lo stato complessivo del modello.

Non devono inoltre essere creati dei costruttori poiché la classe non sarà mai istanziata dall'utente ma sarà sempre ricostruita a partire dai dati ricevuti dal middleware tramite **Reflection**[26].

Infatti l'interfaccia offre sia la funzionalità di caricamento di un istanza, a partire dalla rappresentazione JSON che viene ricevuta al join della sessione dal Proxy, tramite il metodo *loadFromJson*, sia il metodo *applyPatch* che permette di applicare le patch ricevute tramite il messaggio **data-update** del protocollo, che presi in input i dati attuali e le patch restituisce l'oggetto modificato correttamente.

5.3 Eventi

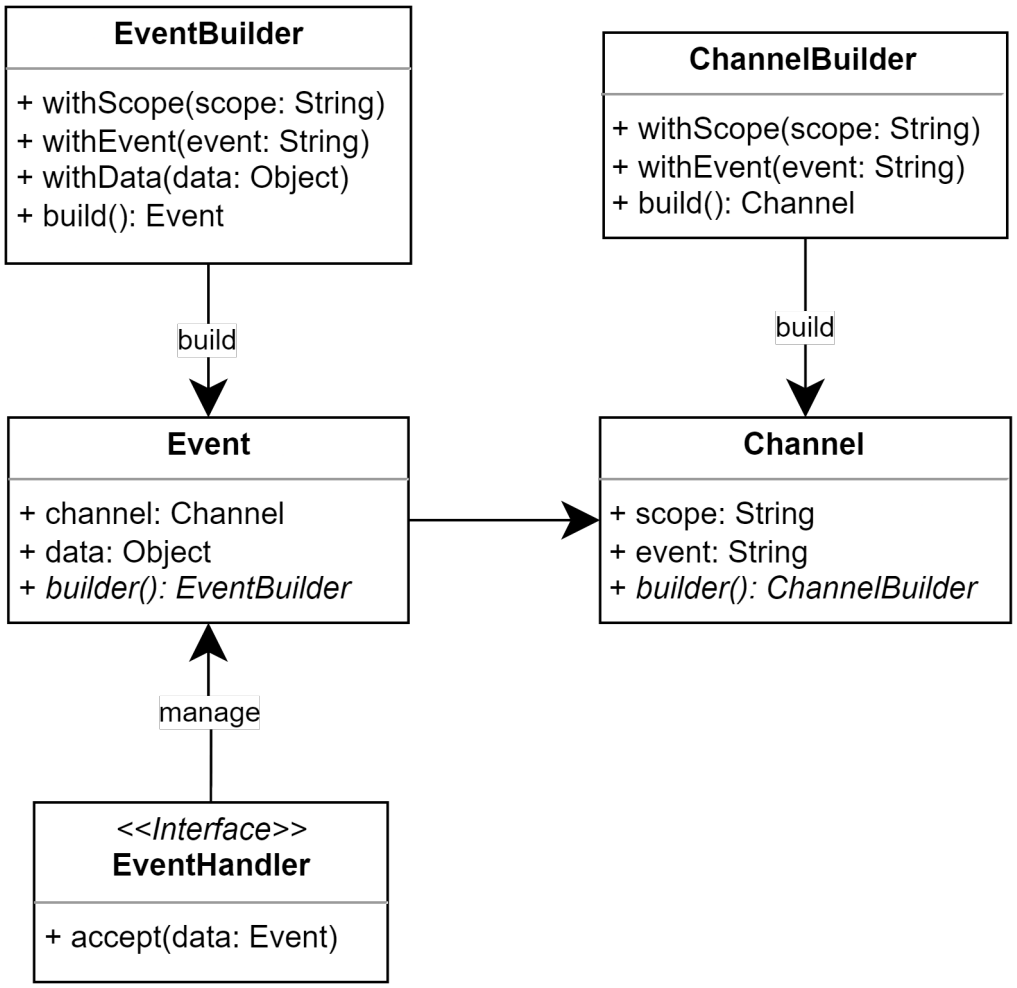

Figura 5.2: Gestione degli eventi nel client

.

Per quanto riguarda la gestione degli eventi nel client anche qui ritroviamo la gestione dei **Channel**, ovvero dei canali su cui vengono pubblicati gli eventi. Tale classe come già mostrato contiene semplicemente i dati relativi al canale, ovvero lo scope e l'evento. Sono stati poi ridefiniti i metodi *equals* e *hashCode* in modo da poter utilizzare anche nel client il canale come indice di una mappa, che correla quindi il Channel ad un **EventHandler**, che nel nostro caso altro non sarà che una **Consumer** di oggetti di tipo **Event**, i quali sono la rappresentazione di un evento, che quindi comprende il canale su cui è stato

oppure deve essere pubblicato e i dati, che ovviamente potrebbero essere nulli poiché non è obbligatorio che un evento ne abbia.

```
public class Channel {

    public final String scope;
    public final String event;

    public Channel(final String scope, final String event) {
        this.scope = scope;
        this.event = event;
    }

    public static ChannelBuilder builder() {
        return new ChannelBuilder();
    }

    @Override
    public boolean equals(Object o) { ... }

    @Override
    public int hashCode() { ... }

}
```

```
public class Event {

    public final Channel channel;
    public final Object data;

    public Event(final Channel channel, final Object data) {
        this.channel = channel;
        this.data = data;
    }

    public static EventBuilder builder() {
        return new EventBuilder();
    }

}
```

```
public interface EventHandler extends Consumer<Object>{}
```

5.4 ProxyClient

Si procede ora illustrando la classe il cui compito è quello di occuparsi della gestione del protocollo di sincronizzazione, che si connetterà quindi al Proxy e manterrà in sincronia dati ed eventi, gestendo anche il salvataggio degli handler di tali eventi.

```
public class ProxyClient<D extends EnvironmentData> {

    private final Class<D> dataClass;
    private D data;
    private final Socket socket;
    private boolean ready = false;

    // Map of event handlers for each channel
    private final Map<Channel, EventHandler> subscriptionHandlersMap
        = new HashMap<>();

    public ProxyClient(final Class<D> dataClass) {
        this.dataClass = dataClass;
        this.socket = IO.socket(create("ws://localhost:3000"));
        this.socket.on(CONNECTION_READY.message, (data) ->
            this.socket.emit(JOIN.message));
        this.socket.on(READY.message, (data) -> this.onReady());
        this.socket.on(DATA.message, (data) ->
            this.onData(String.valueOf(data[0])));
        this.socket.on(DATA_UPDATE.message, (patches) ->
            this.onDataUpdate(String.valueOf(patches[0])));
        this.socket.on(EVENT.message, (data) ->
            this.onEvent(String.valueOf(data[0]),
            String.valueOf(data[1]), data[2]));
    }

    public void connect() {
        this.socket.connect();
        while (!this.ready) {
            Thread.sleep(100);
        }
    }

    public D getData() {
        return this.data;
    }
```

```
    public void subscribe(final String scope, final String event,
        final EventHandler callback) {
        this.subscriptionHandlersMap.put(new Channel(scope, event),
            callback);
        this.socket.emit(SUBSCRIBE.message, scope, event);
    }

    public void unsubscribe(final String scope, final String event) {
        this.subscriptionHandlersMap.remove(new Channel(scope,
            event));
        this.socket.emit(UNSUBSCRIBE.message, scope, event);
    }

    public void publish(final String scope, final String event,
        final Object data) {
        this.socket.emit(PUBLISH.message, scope, event, data);
    }

    private void onReady() {
        this.ready = true;
    }

    private void onData(final String jsonData) {
        this.data = EnvironmentData.loadFromJson(jsonData,
            this.dataClass);
    }

    private void onDataUpdate(final String jsonPatch) {
        this.data = EnvironmentData.applyPatch(this.data, jsonPatch);
    }

    private void onEvent(final String scope, final String event,
        final Object data) {
        final Channel channel = new Channel(scope, event);
        final EventHandler handler =
            this.subscriptionHandlersMap.get(channel);
        handler.accept(new Event(channel, data));
    }

}
```

Tale classe è quindi il pilastro della libreria client, infatti essa implementa tutta la procedura di sincronizzazione. Il suo compito quindi è la connessione

al Proxy e successivamente il mantenimento dello stato del modello, tramite l'oggetto EnvironmentData che conterrà quindi sempre lo stato aggiornato. Infatti possiamo notare come nella sua costruzione essa si connetta al Proxy e faccia il setup per gestire tutti gli eventi.

Tale classe permette quindi anche la gestione di eventi, in primis partendo dalla possibilità di effettuare sottoscrizioni a canali registrando il relativo handler, in modo da poterlo richiamare quando si riceve un evento su quel canale. Viene inoltre esposta la funzionalità di pubblicazione di eventi.

I metodi più rilevanti in questa classe sono gli ultimi 3, ovvero quelli che gestiscono i messaggi ricevuti dal middleware.

Come primo metodo si ha quello di *onData* che viene invocato quando si riceve il messaggio **data** che contiene lo stato del modello nel momento della connessione in formato JSON, il quale verrà utilizzato per ricostruire l'oggetto *data* tramite il metodo *loadFromJson* dell'interfaccia EnvironmentData, che è stato spiegato in precedenza.

Come secondo metodo si ha quello di *onDataUpdate* che viene invocato quando si riceve il messaggio **data-update** che contiene le patch di aggiornamento dello stato del modello. Tale metodo si occuperà quindi di processarle e modificare di conseguenza lo stato dell'environment. Il metodo utilizzato per applicare le modifiche è quello statico *applyPatch* dell'interfaccia EnvironmentData che è già stato spiegato.

L'ultimo metodo di rilevante importanza è *onEvent* che viene invocato ogni qual volta si riceva un evento, il quale si occuperà di cercare tra i gli handler che sono stati registrati quelli che rispondono al canale dell'evento appena ricevuto. Successivamente quindi potrà procedere ad invocare tale handler.

5.4.1 CroquetProxyProtocolMessages

Per racchiudere tutti i tipi di messaggi del protocollo, dato che sono un numero limitato si è scelto di utilizzare una Enum che permetta di incapsulare in unico punto tutti i possibili messaggi. In questo caso se si presentano delle variazioni nei tipi di messaggi questo risulterà l'unico punto che necessita cambiare.

```
public enum CroquetProxyProtocolMessages {

    // Client -> Proxy
    JOIN("join"),
    SUBSCRIBE("subscribe"),
    UNSUBSCRIBE("unsubscribe"),
    PUBLISH("publish"),
```

```
    // Proxy -> Client
    CONNECTION_READY("connection-ready"),
    DATA("data"),
    DATA_UPDATE("data-update"),
    READY("ready"),
    EVENT("event");

    public final String message;
    CroquetProxyProtocolMessages(String message) {
        this.message = message;
    }

}
```

5.5 Environment

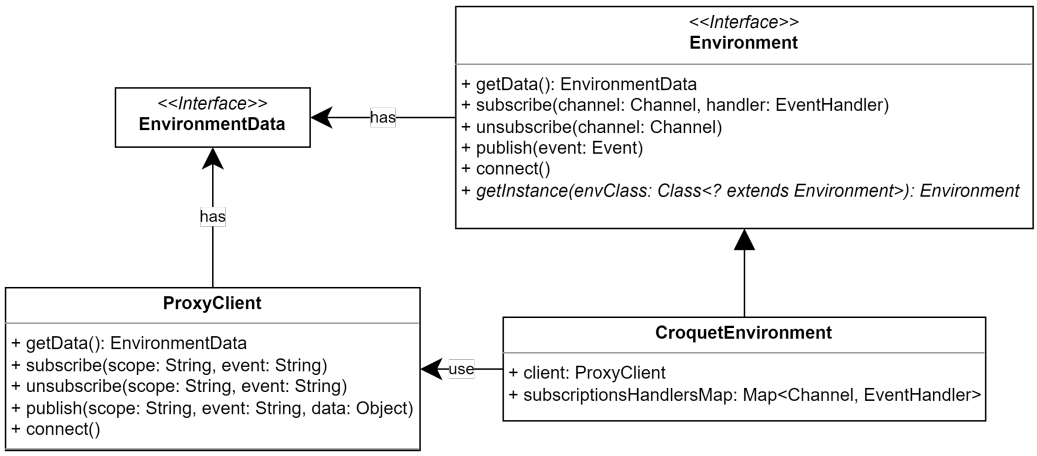

Figura 5.3: Gestione Environment

.

Come già introdotto nel capitolo dedicato allo studio dello sviluppo di tale middleware, quello che si vuole esportare sul client non è semplicemente una View, bensì un entità con maggiori funzionalità rispetto al concetto di View classico.

La conclusione di tale valutazione è quindi stata quella dell'entità Environment, che come già spiegato permette sia l'accesso ai dati dell'ambiente condiviso, ovvero gli **Environment Data**, sia la gestione degli eventi nell'environment.

```
public interface Environment<D extends EnvironmentData> {

    D getData();
    void subscribe(final Channel channel, EventHandler callback);
    void unsubscribe(final Channel channel);
    void publish(final Event event);
    void connect();

    static <E extends Environment<?>> E getInstance(Class<E> clazz) {
        // ...
    }

}
```

Tali funzionalità sono quindi state espresse tramite l'apposita interfaccia **Environment** che mette a disposizione tramite gli appositi metodi le funzionalità sopra elencate. Inoltre tale interfaccia ha il metodo statico *getInstance*, che specificata una classe che implementa un Environment si occupa di crearne un istanza e fare partire la connessione, e sarà proprio questo il metodo da chiamare per ottenere un oggetto Environment utilizzabile.

Quello che si è fatto è quindi un implementazione di tale interfaccia che quindi costituisse un environment basato su Croquet, che utilizzerà quindi il **ProxyClient** spiegato precedentemente.

La classe costruita quindi ha preso il nome di **CroquetEnvironment**, la quale implementa l'interfaccia **Environment** e semplicemente ridireziona le richieste che gli vengono fatto in quanto oggetto di tipo Environment al client che mantiene all'interno.

```
public class CroquetEnvironment<D extends EnvironmentData>
    implements Environment<D> {

    private final ProxyClient<D> client;

    public CroquetEnvironment(Class<D> dataClass) {
        this.client = new ProxyClient<D>(dataClass);
    }

    @Override
    public D getData() {
        return this.client.getData();
    }

    @Override
    public void subscribe(final Channel channel, EventHandler
        callback) {
        this.client.subscribe(channel.scope, channel.event, callback);
    }

    @Override
    public void unsubscribe(final Channel channel) {
        this.client.unsubscribe(channel.scope, channel.event);
    }

    @Override
    public void publish(final Event event) {
        this.client.publish(event.channel.scope, event.channel.event,
            event.data);
```

```
    }

    @Override
    public void connect() {
        this.client.connect();
    }

}
```

Capitolo 6

Metodologia di sviluppo

Si vuole di seguito mostrare in cosa consiste quindi la procedura di sviluppo di un applicazione che utilizzi il middleware creato.

Quello che sarà fatto quindi è utilizzare un applicazione di esempio per spiegare i vari procedimenti di sviluppo, partendo dalla creazione della parte di Model, che andrà scritta quindi in linguaggio JavaScript e utilizzerà le componenti sviluppate in tale linguaggio, arrivando infine alla parte del client, il quale utilizzerà la libreria Java appena creata.

L'obiettivo dell'applicazione che si vuole sviluppare è estendere il contatore visto in precedenza, aggiungendo come funzionalità un altro contatore, che però non viene incrementato in automatico ma solamente tramite eventi generati dall'utente.

Si ha quindi come requisiti quelli di avere sia un contatore che lavora in automatico e rimane sincronizzato tra tutti i client, che lo vedranno scorrere allo stesso modo e ognuno degli utenti potrà decidere di resettarlo tramite evento. In aggiunta a questo primo contatore ne deve essere presente un altro, il cui incremento e decremento è regolato in base ad aventi generati dall'utente. Tale contatore quindi non prevede un auto-incremento come il primo.

6.1 Extended Counter Proxy

Come prima cosa si andrà quindi ad illustrare e spiegare lo sviluppo della parte di modello che va scritto in JavaScript.

Per tale implementazione si andrà a sfruttare la decomposizione dei Model per poter modularizzare l'applicazione.

6.1.1 CounterModel

Di seguito quindi si implementerà il modulo adibito al contatore automatico che ne risulterà identico alla versione presentata nel secondo capitolo se non per le API dei dati che nascondo il DataModel interno.

```
// CounterModel.js
import { ComponentModel } from "../ComponentModel.js";

export class CounterModel extends ComponentModel {

    init(options) {
        super.init(options);
        this.registerData({ counter: 0 })
        this.future(1000).tick()
        this.subscribe("counter", "reset", this.resetCounter)
    }

    tick() {
        this.getData().counter++;
        this.publish("counter", "updated")
        this.future(1000).tick()
    }

    resetCounter() {
        this.getData().counter = 0;
        this.publish("counter", "updated")
    }

}
```

6.1.2 ManualCounterModel

Si passa quindi alla parte del contatore manuale, il quale risponderà agli eventi di incremento e decremento modificando la sua variabile *manualCounter* e alla modifica notificherà il cambiamento.

```
// ManualCounterModel.js
import { ComponentModel } from "../ComponentModel.js";

export class ManualCounterModel extends ComponentModel {

    init(options) {
```

```
        super.init(options);
        this.registerData({ manualCounter: 0 })
        this.subscribe("manual-counter", "increment", this.increment)
        this.subscribe("manual-counter", "decrement", this.decrement)
    }

    increment() {
        this.getData().manualCounter++;
        this.publish("manual-counter", "updated")
    }

    decrement() {
        this.getData().manualCounter--;
        this.publish("manual-counter", "updated")
    }

}
```

6.1.3 ExtendedCounterApplication

Bisogna quindi definire il Model principale, che estenderà quindi la classe ApplicationModel e che registrerà i vari componenti del modello.

```
// ExtendedCounterApplication.js
import { ApplicationModel } from "../ApplicationModel.js"
import { CounterModel } from "./CounterModel.js"
import { ManualCounterModel } from "./ManualCounterModel.js"


export class ExtendedCounterApplication extends ApplicationModel {

    init(options) {
        super.init(options)
        this.registerComponents([
            { name: "CounterModel", model: CounterModel },
            { name: "ManualCounterModel", model: ManualCounterModel }
        ])
    }

}
```

6.1.4 ExtendedCounterMain

Infine non resta che implementare il Main che non farà altro che richiamare il metodo startProxy specificandone i relativi parametri per l'avvio del server e per la connessione alla rete Croquet.

Una volta avviato, il modello entrerà in esecuzione anche se non ha dei client connessi.

```
// ExtendedCounterProxyMain.js
import { ExtendedCounterApplication } from
    "./ExtendedCounterApplication.js"

const apiKey = ""
const appId = ""
const name = ""
const password = ""


ExtendedCounterApplication.startProxy({
    apiKey: apiKey,
    appId: appId,
    name: name,
    password: password,
    port: 3000,
})
```

6.2 Extended Counter Client

Completata la descrizione dell'implementazione della parte che concerne il modello, si può procedere mostrando come viene realizzata l'implementazione del client per tale applicazione, la quale sfrutterà quindi la libreria spiegata nel capitolo precedente.

6.2.1 CounterData

Come primo punto andrà creata la classe che funge da holder per i dati del modello, che in questo caso sono il contatore automatico e quello manuale. Si avrà cosi la possibilità di avere un unico oggetto che contiene tutti i dati dell'environment del modello. Bisognerà quindi creare la classe **CounterData** che estende l'interfaccia **EnvironmentData** e dichiarare tali campi pubblici in modo che siano accessibili per la ricostruzione da JSON tramite Reflection.

```
public class CounterData implements EnvironmentData {
    public int counter = 0;
    public int manualCounter = 0;
}
```

6.2.2 CounterEnvironment

Successivamente rimane solo da definire l'Environment dell'applicazione creando l'apposita classe che estende il **CroquetEnvironment** e ne specifica il parametro che indica la classe holder dei dati dell'applicazione.

Sarà quindi necessario definire un costruttore vuoto in modo che la libreria possa poi usare la Reflection per istanziarlo.

A questo punto il CounterEnvironment è già pronto per il funzionamento, si procede quindi definendo i vari metodi di modifica del manualCounter al suo interno per incapsularli e avere un unica sorgente di eventi.

```
public class CounterEnvironment extends
    CroquetEnvironment<CounterData> {

    public CounterEnvironment() {
        super(CounterData.class);
    }

    public void resetCounter() {
        this.publish(Event.builder()
            .withScope("counter")
            .withEvent("reset")
            .build());
    }

    public void incrementManualCounter() {
        publish(Event.builder()
                .withScope("manual-counter")
                .withEvent("increment")
                .build());
    }

    public void decrementManualCounter() {
        publish(Event.builder()
                .withScope("manual-counter")
                .withEvent("decrement")
                .build());
```

```
    }

}
```

6.2.3 CounterMain

Come ultimo passaggio per vedere in azione il funzionamento complessivo, non resta che definire il Main che crei un istanza del **CounterEnvironment** tramite il metodo statico *getInstance* dell'interfaccia Environment, che provvederà a crearne l'istanza e ad avviare la connessione del client con il Proxy.

Infine bisogna solo effettuare le sottoscrizioni utilizzando le API di gestione eventi che sono esposti da tale interfaccia.

```
public class Main {

    public static void main(String[] args) {

        final CounterEnvironment environment =
            Environment.getInstance(CounterEnvironment.class);

        final Channel counterUpdatedChannel = Channel.builder()
                .withScope("counter")
                .withEvent("updated")
                .build();

        final Channel manualCounterUpdatedChannel = Channel.builder()
                .withScope("manual-counter")
                .withEvent("updated")
                .build();

        environment.subscribe(counterUpdatedChannel, event -> {
            System.out.println("Counter: " +
                environment.getData().counter);
        });

        environment.subscribe(manualCounterUpdatedChannel, event -> {
            System.out.println("Manual Counter: " +
                environment.getData().manualCounter);
        });
```

```
        // Read from the console and if it's a + or - then increment
           or decrement the manual counter
        final Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            final String input = scanner.next();
            if (input.equals("+")) {
                environment.incrementManualCounter();
            } else if (input.equals("-")) {
                environment.decrementManualCounter();
            }
        }

    }

}
```

In questo caso, il client sviluppato ha funzionalità di osservazione e di ascolto sull'environment, infatti si mette in ascolto sulle modifiche fatte sia al contatore automatico sia a quello manuale e ne stampa il valore ai relativi cambiamenti.

Inoltre le sue funzionalità riguardano anche l'agire sull'environment, infatti il loop alla fine non farà altro che rimanere in ascolto dalla console e nel caso si riceva un carattere "+" in input allora verrà richiamato il metodo del CounterEnvironment che è adibito all'incremento del contatore manuale. Nel caso invece si riceva un carattere "-", verrà invocato il metodo di decremento, mentre come ultimo caso se si riceve il carattere "x" allora verrà resettato il contatore automatico.

Si ottiene quindi che tale client ha sia funzionalità di osservazione dell'environment sia ne influisce pubblicando gli eventi relativi al contatore manuale.

Conclusioni

Le possibilità offerte dal framework e dall'architettura che mette a disposizione Croquet sono molto elevate e lo sviluppo di un middleware su di esso che permette l'acceso a tali applicazioni a componenti che ora possono essere sviluppate in un qualsiasi linguaggio di programmazione, purché implementino la procedura di sincronizzazione, ampliano ancora maggiormente ciò che si può fare con tale framework.

Sicuramente anche la metodologia di accesso che viene esportata sul client, ovvero il concetto dell' Environment che è condiviso tra tutti i client, aiuta a semplificare anche il modello di sviluppo. Può infatti essere utilizzato semplicemente come un database condiviso tra più entità che si affacciano sullo stesso ambiente, ma le sue funzionalità non si limitano a questo, poiché possono essere gestiti anche eventi che quindi connettono componenti di qualsiasi tipo.

Quello che rende particolare però Croquet è il fatto che la parte di controllo, ovvero quella che svolge computazione e modifiche non risiede solo nei client ma bensì ha anche una parte di modello che risulta standard e condivisa tra tutti i client. Ne risulta quindi che l'utilizzo di tale framework non è limitato allo scambio di eventi e di dati, bensì anche alla condivisione della computazione.

Si ha quindi ora uno strumento molto potente, i cui sviluppi futuri prevedono sicuramente lo sviluppo di ulteriori librerie client per il supporto di nuovi linguaggi di programmazione ed una fase di sperimentazione e di studio sulla coordinazione di entità distribuite che utilizzano l'accesso all'Environment. Inoltre l'avere a disposizione la possibilità di accesso all'ambiente condiviso da parte di più linguaggi di programmazione fornisce sicuramente un aiuto allo sviluppo di componenti facenti parte dello stesso sistema ma con funzionalità diverse che risultano più language-dependent. Questo semplifica quindi lo sviluppo di componenti che richiedono ad esempio l'utilizzo di librerie che esistono solo per determinati linguaggi.

Di rilevante importanza potrebbe essere anche un'analisi di un integrazione di tale middleware per lo sviluppo di sistemi ad agenti, i quali potrebbero quindi comunicare tra di loro sfruttando l'Environment offerto dalla libreria client.

Ringraziamenti

In primis voglio ringraziare la mia famiglia, che è sempre stata presente, che mi è stata vicina e non mi ha mai fatto mancare nulla.

Ringrazio inoltre chiunque mi abbia sostenuto e abbia incoraggiato il mio percorso, in particolare tutti quei professori che sin dalle superiori mi hanno saputo ispirare, facendomi appassionare sempre di più alla materia che ho sempre saputo sarebbe stato quello che volevo fare da grande.

Non mancano di certo i ringraziamenti a tutte le persone passate e presenti che abbiano contribuito alla mia crescita personale, la quale mi ha portato ad essere la persona che sono oggi e di cui vado fiero. Senza di voi non sarei mai stato in grado di raggiungere questo traguardo, grazie.

Il ringraziamento più speciale va però alle persone conosciute in questo percorso e in questa città, in particolare ai miei amici di Zucchero Sintattico, che reputo come una seconda famiglia. In ordine alfabetico: Alex, Gustavo, Kelvin, Luigi, Manuel, Stefano e Tommaso, senza dei quali l'esperienza non sarebbe stata minimamente la stessa. Grazie per esserci sempre stati, per avere condiviso questa esperienza con me e per avermi regalato gioie infinite.

Bibliografia

[1] Robert Bryce and Gautam Srivastava. The addition of geolocation to sensor networks, 01 2018.

[2] croquet.io. Croquet architecture. [Online; accessed September 9, 2022].

[3] croquet.io. The croquet reflector network(crn). [Online; accessed September 10, 2022].

[4] croquet.io. Events and publish/subscribe. [Online; accessed September 10, 2022].

[5] croquet.io. Model. [Online; accessed September 11, 2022].

[6] croquet.io. Randomness and determinism. [Online; accessed September 10, 2022].

[7] croquet.io. Session. [Online; accessed September 11, 2022].

[8] croquet.io. Simulation time and future sends. [Online; accessed September 10, 2022].

[9] croquet.io. Snapshots. [Online; accessed September 10, 2022].

[10] croquet.io. View. [Online; accessed September 11, 2022].

[11] github.com. Jackson project. [Online; accessed September 18, 2022].

[12] github.com. Java json patch. [Online; accessed September 23, 2022].

[13] github.com. Optional observe callback not called. [Online; accessed September 11, 2022].

[14] github.com. Socket.io-client java. [Online; accessed September 18, 2022].

[15] David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[16] jessmart.in. Concerns of real-time multi-user applications. [Online; accessed September 8, 2022].

[17] jsonpatch.com. Jsonpatch. [Online; accessed September 11, 2022].

[18] nodejs.org. Nodejs. [Online; accessed September 12, 2022].

[19] Yoshiki Ohshima, Aran Lunzer, Jenn Evans, Vanessa Freudenberg, Brian Upton, and David A Smith. An experiment in live collaborative programming on the croquet shared experience platform. 2022.

[20] David P Reed. Teatime: Designing the architectural framework for croquet. *PowerPoint presentation], Oct*, 2006, 2005.

[21] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[22] David A Smith, Alan Kay, Andreas Raab, and David P Reed. Croquet-a collaboration system architecture. In *First Conference on Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings.*, pages 2–9. IEEE, 2003.

[23] socket.io. Socket.io. [Online; accessed September 9, 2022].

[24] squeak.org. Squeak. [Online; accessed September 9, 2022].

[25] www.npmjs.com. Fast json patch. [Online; accessed September 9, 2022].

[26] www.oracle.com. Using java reflection. [Online; accessed September 14, 2022].