Alma Mater Studiorum
Università degli Studi di Bologna

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Scienze di Internet

Tesi di Laurea in Laboratorio di Programmazione Internet

Progettazione e Sviluppo di un Multiplayer Online Game su Reti Peer-to-Peer

Candidato:
Fabrizio Zagaglia

Relatore:
Chiar.mo Prof. Stefano Ferretti

Anno Accademico 2009/2010 - Sessione III

Dedico questa tesi alla mia famiglia

Indice

Introduzione

Astraendo da quelli che sono gli aspetti generali e multidisciplinari che si possono individuare nel processo produttivo di un videogioco (si pensi agli elementi artistici, dalla grafica alla musica, o alla necessità di solidi sistemi economici e di marketing), dal punto di vista tecnico, tra le caratteristiche che contraddistinguono maggiormente un software videoludico, vi sono, da una parte, la necessità di un feedback pressoché immediato alle azioni eseguite dal giocatore, e, dall'altra, la forte interazione tra gli elementi che popolano l'ambiente di gioco.

Questi aspetti implicano una serie di problematiche che, nell'ambito specifico dei giochi Multiplayer online, vengono in qualche modo amplificate: diventano quindi necessarie soluzioni che, a livello di architettura hardware e software, differiscono da quelle richieste, ad esempio, da un videogioco Singleplayer.

Un primo aspetto da analizzare è individuato dalla scelta dell'architettura di rete, tipicamente tra il modello Client-Server ed il modello Peer-To-Peer. Entrambi hanno vantaggi e svantaggi e risultano preferibili l'uno all'altro a seconda del contesto, disponibilità hardware, tipologia e specifiche di gioco.

Un sistema centralizzato Client-Server, da un lato sopperisce al problema di *inconsistenza* dei dati, dal momento che è il Server stesso a mantenere lo stato e a processare la logica di gioco, ma dall'altro risulta meno flessibile,

portabile e scalabile. Il Server inoltre rappresenta uno *SPOF*[1].
Con "*consistenza*" si intende la necessità di avere uno stato di gioco definito da dati integri e valorizzati allo stesso modo su ogni singolo client, dove con "*stato di gioco*" si intende una rappresentazione, una sorta di "*istantanea*", dello stato (posizione, rotazione, azioni, eventi, variabili, etc.) di ogni singolo elemento presente nell'ambiente di gioco.

In un sistema decentralizzato Peer-To-Peer, ogni nodo è paritetico e può fungere quindi contemporaneamente sia da Client che da Server.

Su ogni Peer è presente una copia dello stato di gioco, e nel caso di sistemi Peer-To-Peer "puri", ogni Peer porterà avanti la logica di gioco basandosi sul proprio Data Layer, introducendo quindi problematiche di consistenza e sincronizzazione temporale.
Senza nessun sistema di controllo, infatti, ogni Peer, specie in quelle situazioni ed in quegli eventi soggetti a processi aleatori o notevolmente sensibili alla sincronizzazione temporale, può presentare uno stato di gioco differente rispetto agli altri, e tale errore è destinato a propagarsi aggiornamento dopo aggiornamento, portando ad una divergenza dei risultati potenzialmente molto elevata (i.e. in un Peer la vittoria è assegnata al Player A, mentre in un altro Peer la vittoria è assegnata al Player B).

Obiettivi di progetto

In questa tesi di progetto si é deciso di affrontare lo sviluppo di un videogioco Multiplayer basato sull'architettura Peer-to-Peer. La soluzione adottata, però, non è riconducibile al modello Peer-to-Peer "Puro", in quanto i nodi non sono propriamente paritetici: ogni Peer infatti, pur conoscendo lo stato di gioco globale, ne gestirà direttamente solo una partizione, aggiornando di

[1]Single point of failure, un generico elemento il cui malfunzionamento compromette l'intero sistema del quale fa parte.

fatto solo gli elementi di propria appartenenza.

In questo modo viene contenuto l'impatto del problema di consistenza, dal momento che, desincronizzazioni temporali a parte, tutti gli stati di gioco convergeranno ad uno stato virtualmente condiviso comune. Rispetto al modello "Puro", si è però anche più sensibili a problemi derivanti dal *cheating*[2] e alla formazione di bottleneck attorno all'insieme di dati gestiti da un Peer poco performante. Il che significa l'eventuale implementazione di politiche di distribuzione di calcolo mirate alla riduzione del contesto dei dati che appartengono ai nodi computazionalmente più lenti o con maggiore latenza, in favore di quelli più performanti.

Il gioco realizzato appartiene alla tipologia degli "*Shooter 3D*", e prende spunto, sia dal punto di vista delle meccaniche di gioco che dal punto di vista dello stile grafico, da *Spectre* (si veda il capitolo 1): ogni giocatore controlla un proprio "*Tank*" ed è chiamato distruggere i "*Tank*" nemici controllati dagli altri giocatori.

Nel prossimo capitolo verranno introdotti quelli che sono i sistemi, le architetture e le tecnologie attualmente più utilizzate a seconda dei diversi contesti e delle diverse specifiche di gioco.

Nei capitoli successivi, invece, verranno analizzate nel dettaglio le fasi di progettazione e le scelte implementative del progetto di tesi.

[2]Un sistema grazie al quale è possibile aggirare le regole di gioco traendone vantaggio

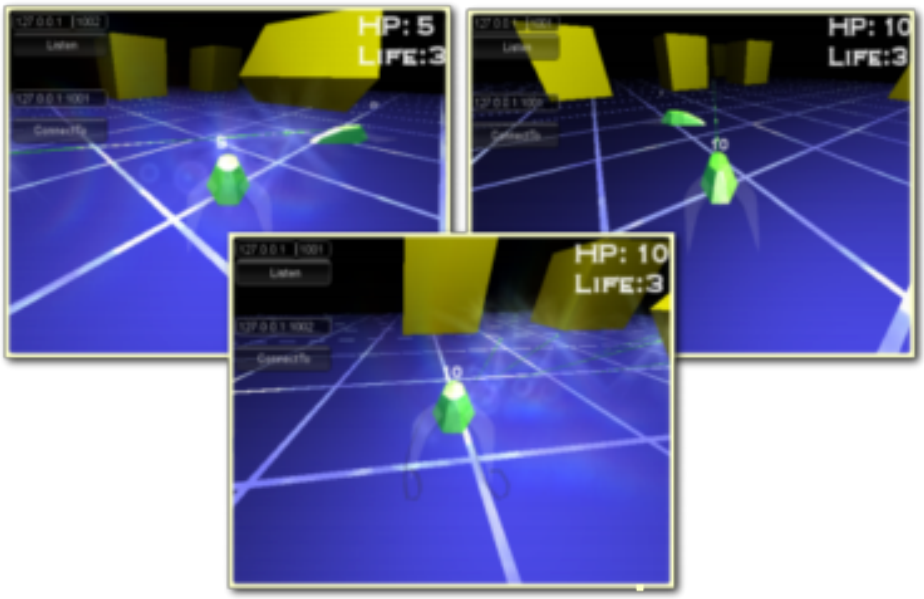

Figura 1: Screenshots

Capitolo 1

Stato dell'arte

1.1 Storia dei MOGs

Il concetto di videogioco Multiplayer (Multiplayer Online Game, MOG) è mutato notevolmente nell'arco della sua evoluzione.
Inizialmente era confinato ad una rappresentazione testuale di *wargames*[1], giocato via *BBS*[2] ed organizzato e gestito da amministratori fisici che, manualmente, calcolavano lo stato di gioco risultante dalle richieste inoltrate dai giocatori, distribuiti sul tutto il globo, ore o addirittura giorni prima.

Presto tale processo venne automatizzato, dando via alla nascita dei *MUD*[3].

Parallelamente, se si estende il discorso anche ai giochi "visuali" e *real-time* (in contrapposizione a quelli testuali), tale concetto è stato dapprima

[1]Un gioco accostabile alla categoria dei giochi strategici, le cui regole sono mirate alla simulazione o alla rappresentazione di uno scenario di guerra, ed il cui scopo è individuato nella vittoria di una battaglia, raggiunta attraverso determinate tattiche e strategie.

[2]Bulletin board system, un sistema software, utilizzato principalmente a cavallo tra gli anni '80 e '90, che permetteva di scambiare dati, da semplici messaggi di testo ad interi file, attraverso la linea telefonica.

[3]Multi User Dungeon, giochi di ruolo testuali multi-utente

confinato al singolo device fisico: due o più giocatori dovevano essere presenti fisicamente nella stanza per poter controllare il proprio personaggio e ricevere il feedback visivo e sonoro corrispondente alle proprie azioni. Si pensi ai *Coin-op* presenti nelle sale giochi, o allo stesso *Pong*, (Atari, 1972) che permetteva a due giocatori di sfidarsi (o cooperare) direttamente sullo stesso device.

La comparsa dei primi giochi Multiplayer online e *real-time* è da ricercare nell'ambito universitario: *Empire* di John Daleske e *Spasim* di Jim Bowery sono tra i primi giochi multi-utente basati sul sitema *PLATO*. Si susseguono, poi, numerose versioni di Empire ambientate nell'universo di *Star Trek* (Xtrek, Netrek, Decwars, WinTrek, etc.): queste si basavano sulla rappresentazione dello spazio bidimensionale, individuato da una griglia nella quale era possibile spostare in tempo reale la propria nave nelle 8 direzioni disponibili.

Particolarmente significativo dal punto di vista storico è stato *Maze War*[4] (1974), videogioco seminale che ha introdotto numerose dinamiche di gameplay successivamente riprese e tutt'ora presenti nella maggior parte dei titoli videoludici odierni. E' considerato infatti il primo FPS 3D Multiplayer.

L'avvento del Personal Computer incentivò la richiesta di un supporto al "NetPlay", cioè una modalità alternativa a quella classica Singleplayer, che consentisse di giocare in Multiplayer via LAN o Internet. *Spectre* (1991) per *Apple Macintosh* fu tra i primi a dare credito a tale richiesta, permettendo a 8 giocatori simultanei di scontrarsi in un'arena tridimensionale vettoriale.

Ma il punto di svolta più significativo è probabilmente rappresentato da *DooM* (1993) di *ID Software*, che, grazie alla distribuzione capillare di una sua versione Shareware, diventò popolare in brevissimo tempo. Concretizza alcune idee legate al mutiplayer già presenti a livello embrionale in Maze War

[4]http://en.wikipedia.org/wiki/Maze_War

e le fissa ad un nuovo standard qualitativo. Per diversi anni a seguire venne utilizzato come termine di paragone.

E' solo verso la fine degli anni '90, parallelamente alla crescente diffusione di Internet su larga scala, che iniziano a definirsi tipologie di gioco *real-time* esclusivamente orientate al Multiplayer: se qualche anno prima questo aspetto era spesso e volentieri relegato a rare modalità di gioco alternative a quella principale Singleplayer, da qui in poi entra prepotentemente tra le necessità più sentite dai giocatori PC, prima, e console poi.

E' il caso di FPS principalmente orientati al Multiplayer come *Quake 3 Arena* (1999) di *ID Software* e *Unreal Tournament* (1999) di *Epic Games*.

In questo periodo nasce ed inizia a prendere piede anche uno dei generi di gioco che ha caratterizzato, e monopolizzato, l'intero decennio '00: è con *Ultima Online* (1997, Origin), infatti, che si inizia a parlare di MMORPG[5], di *avatar* e di mondi persistenti. L'apice del genere, in termini di popolarità (e quindi di ricavi economici) è sicuramente individuabile in *World of Warcraft* (2004, Blizzard).

Tornano inoltre in auge tipologie di gioco più lente e visivamente astratte, in qualche modo direttamente derivabili dalle prime forme testuali di wargame multigiocatore; videogiochi direttamente giocabili da qualsiasi browser, che in parte ricalcano gli stilemi proposti dai MMORPG dal punto di vista delle meccaniche *social* e *addictive* e che fanno della loro estrema accessibilità (basse richieste Hardware, nessun costo d'ingresso o di abbonamento) uno dei maggiori motivi di successo. Tra questi si possono ricordare *Ogame* (Gameforge, 2002), *Travian* (Travian Games, 2005) e *FarmVille* (Zynga, 2009). Quest'ultimo ha raggiunto un'elevata popolarità utilizzando sinergica-

[5]Multiplayer Online Role-Playing Game, gioco di ruolo ambientato su un mondo virtuale persistente

mente la base d'utenza di *Facebook* come sistema di marketing e di diffusione.

1.2 Architetture distribuite per il supporto ai MOGs

Tale eterogeneità non rende possibile l'affermarsi di soluzioni generiche ottimali per ogni tipologia di gioco, ogni tipo di architettura e di tecnologia: i requisiti ed i sistemi di sincronizzazione utilizzati per realizzare un RPG *play-by-post* sono, ad esempio, notevolmente differenti rispetto a quelli utilizzati invece per sviluppare un FPS o un MMORPG.

L'architettura *Client-Server* (Figura 1.1) in ambito videoludico è attualmente predominante: viene utilizzata in FPS Multiplayer come *Quake* o *Unreal Tournament*, o in MMORPG come *Ultima Online* o *World of Warcraft*. Nei primi la simulazione e l'aggiornamento dello stato di gioco sono a carico di un processo fisicamente esterno ai client. Poiché la durata delle partite di uno Shooter è piuttosto bassa (mediamente circa dieci minuti), e non è richiesto nessun tipo di sistema di permanenza dei dati (se non a livello di statistiche esterne), i requisiti per "hostare" un server dedicato sono tali da permettere la proliferazione di server non ufficiali gestiti direttamente dai giocatori, sia in LAN che in Internet [Abrash 1996].
Per quanto riguarda invece i generi di gioco ambientati in un mondo virtuale persistente, il numero di giocatori e di items da gestire a carico di ogni singolo server è notevolmente più alto, e dal momento che il concetto di partita individuata da un inizio ed una fine è assente, in favore di un'esperienza di gioco potenzialmente infinita, i requisiti Hardware per aggiornare e mantenere i dati relativi all'ambiente e alle entità che lo popolano sono tali da rendere necessaria l'esistenza di singoli server di gioco "ufficiali".

L'architettura Peer-To-Peer (Figura 1.2), anche se utilizzata in diversi titoli, specialmente RTS (*Warcraft II* (Blizzard, 1995) o *Demigod* (Gas Powered Games, 2009)), è invece meno diffusa. Inoltre, mentre il modello

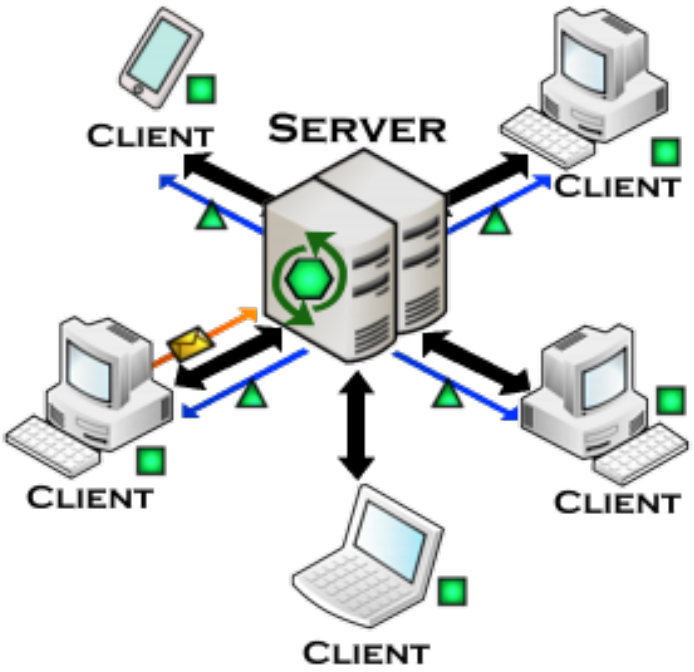

Figura 1.1: Modello Client-Server: lo stato di gioco completo (esagono verde) è mantenuto e aggiornato da un singolo Server centralizzato. Ogni Client è connesso esclusivamente al Server e mantiene una propria visione dello stato di gioco, tipicamente incompleta (quadrato verde), aggiornata grazie ai messaggi inviati, autonomamente o come diretta conseguenza di un input esterno, dal Server (triangolo verde).

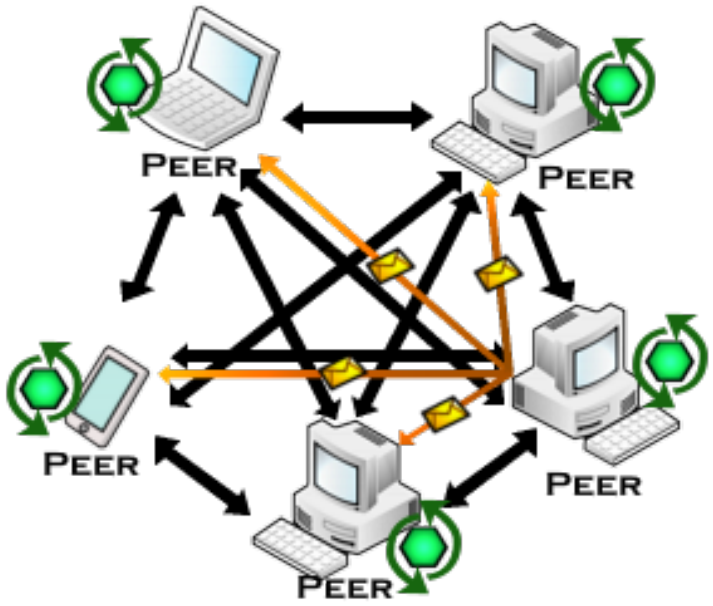

Figura 1.2: Modello Peer-To-Peer: ogni Peer, connesso direttamente o indirettamente a tutti gli altri Peer, mantiene e aggiorna una propria visione dello stato di gioco completo (esagono verde), secondo gli eventi ricevuti.

Client-Server segue una serie di pattern architetturali ormai consolidati, rintracciabili nella maggior parte dei titoli Multiplayer, questo non accade per il modello Peer-To-Peer, che tendenzialmente vede approcci e scelte implementative differenti anche in tipologie di gioco relativamente simili.
In *Age of Empire*, gioco strategico di *Ensemble Studios (1997)*, è utilizzata un'architettura Peer-To-Peer ibrida, caratterizzata da un Peer appartente alla Membership, eletto "Host" che fa da "Game Master" e che si occupa di inizializzare la partita e gestire tutte le situazioni critiche e concorrenziali [Bettner 2001].
Altri paradigmi ibridi, basati sul concetto di "regione" o di "area di interesse", sono stati invece proposti per far fronte al problema di scalabilità individuato nel modello Client-Server per quelle tipologie di gioco, come i MMOG, contraddistinte da un elevato numero di giocatori.
Ad esempio in [Rhalibi 2005] viene discussa un'architettura nella quale un Server centrale controlla una serie di "regioni" composte da singole reti Peer-To-Peer che fanno capo ad un proprio "Supernode", e suddivise in funzione della loro posizione geografica all'interno dell'ambiente virtuale di gioco (Figura 1.3).

In ultima analisi, i fattori che influenzano la scelta dell'architettura derivano da molteplici aspetti, dalla tipologia di gioco alla disponibilità hardware. L'approccio Peer-To-Peer è tutt'ora meno diffuso e standardizzato rispetto a quello centralizzato Client-Server, e spesso si presenta in veste "ibrida", caratterizzata cioè da una gerarchizzazione dei nodi e dalla formazione di "isole" centralizzate. Inoltre a suo supporto, tipicamente viene introdotto un sistema Client-Server apposito per la gestione delle lobby, il player discovery ed il *MatchMaking*.

1.3 Sincronizzazione Temporale

In un'applicazione di rete non è possibile garantire la continua consistenza dei dati senza che vi sia una qualche forma di sincronizzazione temporale:

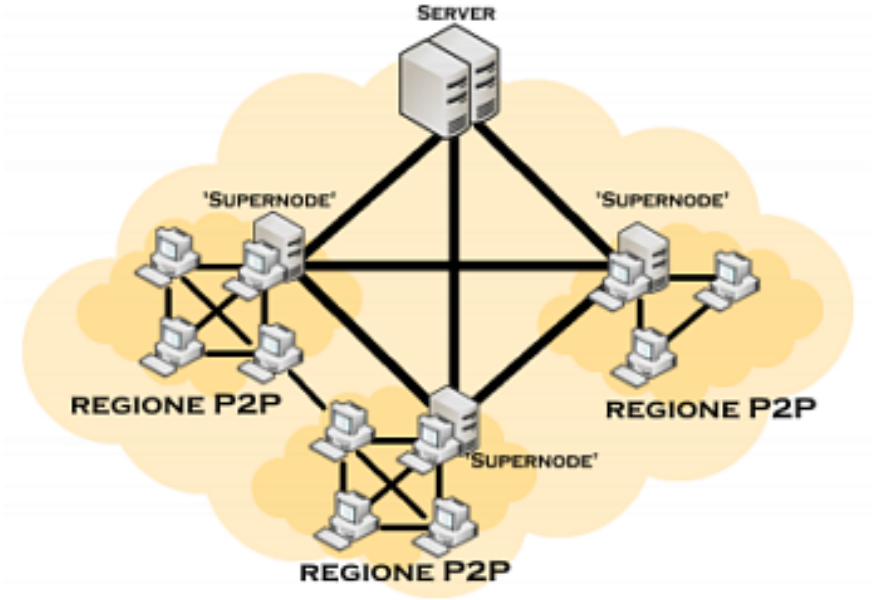

Figura 1.3: Modello ibrido proposto in [Rhalibi 2005] ed in [Luo 2010], e discusso in [Due Billing 2006]

i dati che individuano lo stato di un dominio, sono per definizione mutevoli, sono cioè destinati a cambiare valore nel corso del tempo.
Senza una corrispondenza tra un determinato dato e l'istante di tempo nel quale quest'ultimo assume un determinato valore, l'informazione apportata perde di significato.
Nelle applicazioni *real-time*, tale aspetto è destinato ad avere un impatto considerevole, dal momento che i cambi di stato tipicamente avvengono con una frequenza decisamente alta. Nell'ambito videoludico, inoltre, tra la sincronizzazione temporale e la sincronizzazione dello stato, si instaura una terza problematica: Il Cheating (si veda la sezione 5.1).
Una situazione ideale di perfetta consistenza, nella quale ogni giocatore vede esattamente lo stesso stato di gioco globale allo stesso preciso istante di tempo assoluto, è fisiologicamente impossibile da realizzare causa latenza di rete.

Per gestire la problematica di sincronizzazione è necessario valutare i termini di un compromesso tra la reattività di risposta e la consistenza ad un determinato istante di tempo: un approccio "*conservativo*" è implementato in sistemi di sincronizzazione "*Stop-And-Wait*", come il *Lock-Step* (Figura 1.4) e la *Bucket Synchronization* (Figura 1.5), ed è caratterizzato, da una parte da una forte consistenza a livello temporale, poiché ogni azione viene processata solo quando questa è, teoricamente, conosciuta da tutti i Peer appartenenti alla Membership, e, dall'altra, come diretta conseguenza, da un tempo di reazione alto.
Per questo motivo, questo genere di sincronizzazioni vengono utilizzate soprattutto in quei giochi, come gli RTS, caratterizzati da un Gameplay relativamente lento.
É possibile comunque sfruttarle anche in tipologie di gioco più "veloci", attivandole solo in particolari contesti di concorrenza e all'interno della stessa area di interesse (ad esempio nella gestione di una collisione tra due oggetti in avvicinamento).

Un approccio "*ottimistico*", implementato ad esempio della *Time-Warp Synchronization*, prevede invece un' esecuzione degli eventi immediata: questi infatti saranno processati appena giunti a destinazione. Il vincolo temporale atto a garantire la consistenza, che nell'approccio "*conservativo*" è realizzato mediante fasi ed intervalli di tempo "*bloccanti*", è qui invece rappresentato dal processo di "*Roll-back*". Ad ogni evento viene infatti assegnato un *Time-Stamp*: quando un Peer riceve un evento contrassegnato da un *TimeStamp* meno recente rispetto a quello attuale, vengono prima cancellati tutti gli effetti scaturiti da eventi già processati riportanti un *TimeStamp* più recente, poi eseguito l'evento in questione, ed infine ri-processati quest'ultimi. Alcuni eventi possono però essere "*correlati*" tra loro: l'esistenza -o la non esistenza- di un evento "*chiave*" all'interno di una "*Correlation chain*" può alterare o compromettere il significato e la validità degli eventi ad esso correlati: in altre parole, compromettere la consistenza dello stato di gioco (Figura 1.6). Questo tipo di sincronizzazioni permette di avere una maggiore reattività di interazione rispetto a quelle basate sull'approccio "*conservativo*", ma, per garantire l'esistenza di una base dati sufficientemente ampia da permettere il recovery degli stati precedenti in fase di rollback, richiede tendenzialmente una quantità di memoria più elevata.

Per limitare questo problema, si può intervenire eliminando gli eventi obsoleti, cioè quegli eventi, non correlati, ridondanti e sostituibili quindi da eventi più recenti che, partendo dal medesimo stato iniziale, portano al medesimo stato finale.

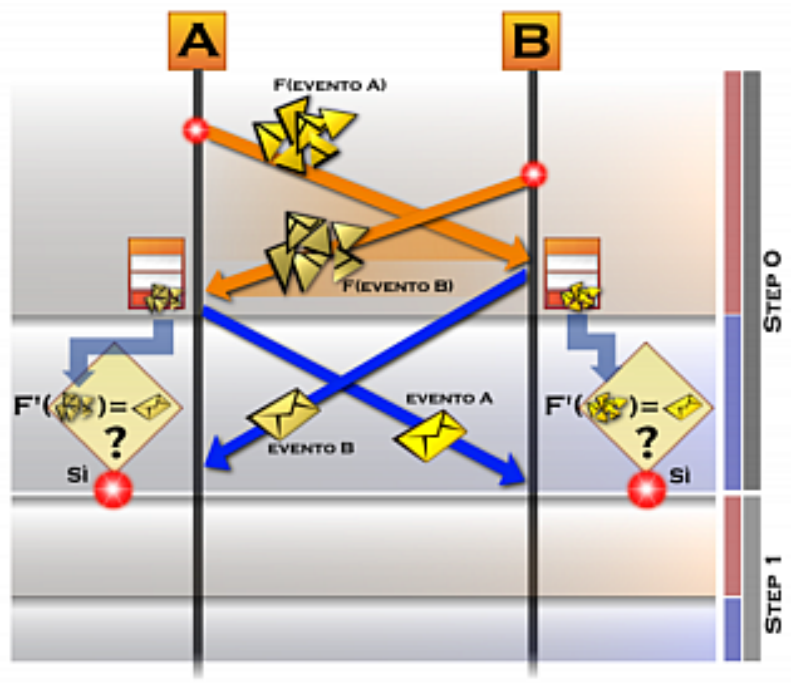

Figura 1.4: LockStep Synchronization. In questo sistema di sincronizzazione, l'esecuzione degli eventi è volutamente ritardata. Un Peer non processerà nessun evento fino a quando tutti gli altri Peer appartenenti alla Membership non saranno "pronti" per passare allo step successivo, ovvero fintanto che non riceverà le loro intenzioni, sottoforma di evento. Lo si può associare ad un sistema a turni, nel quale ogni attore è obbligato ad eseguire la propria mossa (eventualmente nulla) prima di passare al turno successivo.
Per evitare che un Peer malevolo possa modificare le proprie azioni in conseguenza alla ricezione degli eventi appartenenti agli altri Peer, ogni step viene diviso in due fasi: nella prima ogni Peer annuncia una propria "intenzione", inviando, ad esempio, l'Hash dell'azione voluta a tutti gli altri Peer appartenenti alla Membership. L'"intenzione" non identifica un'azione, nel senso che dal suo valore non è possibile ricavare tramite inferenza la semantica dell'azione stessa. E' solo nella seconda fase, infatti, che ogni Peer è tenuto a rivelare la propria azione, inviando, agli altri Peer, il relativo evento. Questo infine sarà eseguito solo se, applicando la funzione inversa a quella utilizzata per ricavare l'"intenzione", il valore risultate sarà equivalente all'azione stessa. Rispetto alla Bucket Synchronization, questo sistema da un lato argina il problema del *Look Ahead Cheat* (5.1), ma dall'altro risulta essere più sensibile alle variazioni di latezna (*Jitter*), poiché ogni step, ed ogni singola fase è individuata da un periodo di tempo variabile.

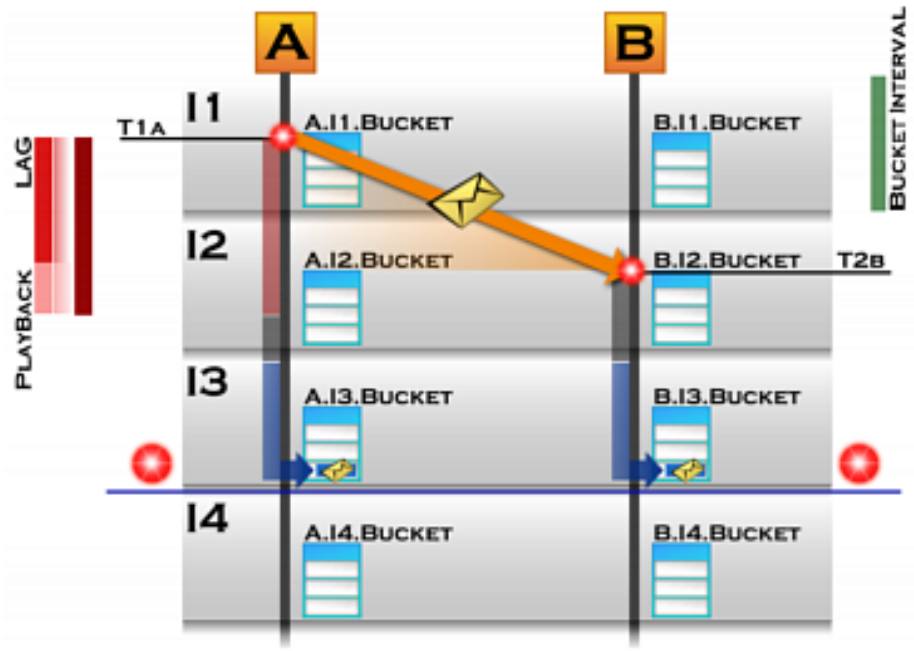

Figura 1.5: Bucket Synchronization: nella Bucket Synchronization il tempo viene idealmente suddiviso in intervalli, "Bucket Intervals" (rettangoli grigi), di N millisecondi (rettangolo verde), calcolati in base al Peer più lento. Viene inoltre definito un TimeSpan, "Delay" (rettangolo rosso), calcolato in base alla latenza più alta presente all'interno della Membership. Ogni Peer assegna ad ogni singolo intervallo un "Bucket", cioè un Buffer destinato a mantenere gli eventi in attesa di essere effettivamente eseguiti.
Nell'intervallo I1, il Peer *A* genera un evento e lo invia di conseguenza agli altri Peer appartenenti alla Membership (Peer *B*). Localmente, però, tale evento non viene eseguito come conseguenza diretta della sua generazione: Verrà invece eseguito in I3, poiché quest'ultimo rappresenta il primo intervallo di tempo successivo all'istante individuato dalla somma temporale tra il momento di generazione ed il periodo di "Delay".
Il Peer *B*, riceve l'evento in I2 ma, utilizzando il medesimo calcolo, sa che dovrà eseguirlo in I3. In entrambi i Peer, quindi, l'evento in questione verrà memorizzato nel Bucket relativo all'intervallo I3, e quindi eseguito a tempo debito.

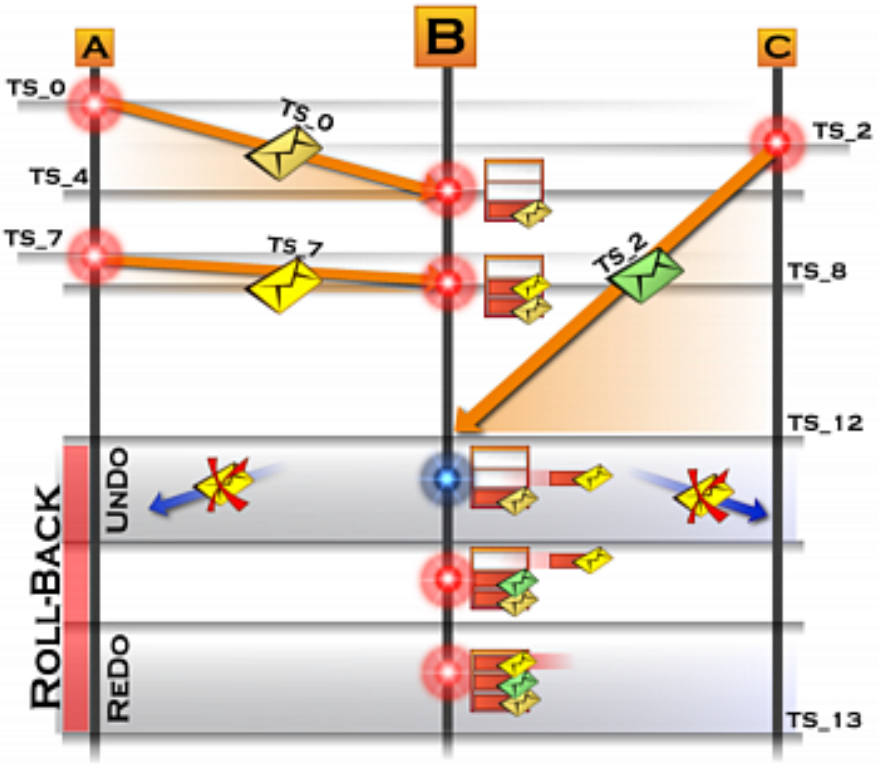

Figura 1.6: Time-Warp Synchronization: ogni evento è temporizzato tramite un TimeStamp e viene eseguito direttamente al momento della sua ricezione. Dal punto di vista del Peer B, la situazione mostrata in figura si configura nel modo seguente: Il Peer A genera un evento in TS_0, mentre il Peer C genera un evento in TS_2.
In TS_4 l'evento di A giunge a B, e viene immediatamente eseguito, mentre l'evento di C tarda ad arrivare.
In TS_7 il Peer A invia un secondo evento che giunge a B in TS_8.
Solo in TS_12 il Peer B riceve l'evento inviato da C in T_2.
Il Peer B, però ha già eseguito il secondo evento proveniente da A (TS_7), quindi è costretto ad eseguire un *Roll-back*, poiché l'evento inviato dal Peer C deve essere eseguito prima, in quanto generato ad un TimeStamp precedente. Elimina quindi gli effetti causati dal secondo messaggio di A mediante un "*Anti-Message*", notificato di conseguenza agli altri Peer. L'UnDo può essere implementato ad esempio memorizzando lo stato incrementale in una "*State queue*".

1.4 Tecnologie

Dal punto di vista dello sviluppo, il videogioco è un software complesso, e come tale si presta ad essere diviso in moduli separati, ognuno dedicato ad una precisa funzione.

Escludendo il modulo che implementa le logiche di gioco, che è fisiologicamente dipendente dalle specifiche di gameplay, si possono individuare moduli, tipicamente generalizzabili, come il *Renderer* (o Engine 3D), utilizzato per disegnare su schermo gli oggetti tridimensionali, l'*Engine fisico*, utilizzato invece per simulare il movimento, le collisioni e le relative risposte o l'*Engine di rete* (Networking Engine o "NetCode"), dedicato alla connessione e alla sincronizzazione dello stato di gioco.

Dal momento che le funzioni supportate da tali moduli sono piuttosto generiche e "gamelogic independent", diversi Team e Software House rilasciano pubblicamente le versioni dei propri Engines, opportunamente aggiornati secondo gli ultimi standard qualitativi. In questo modo è possibile sviluppare videogiochi astraendo da una vasta gamma di aspetti inerenti al game development, evitando quindi di dover reimplementare complesse componenti software già esistenti, e risolverne le problematiche annesse.

Per quanto riguarda gli Engines 3D, spesso questi sono accompagnati da motori di gioco e Tools di supporto: attualmente a livello di software Open Source, gli Engines grafici più utilizzati sono *Ogre3D*[6] che si presenta come Renderer multipiattaforma basato su uno scenegraph a nodi, e *Irrlicht*[7], simile al primo quanto a features di base ma predisposto al supporto di caratteristiche tipiche di un Game Engine quali la gestione input, dell'audio e della fisica.

[6]Object-Oriented Graphics Rendering Engine, http://www.ogre3d.org
[7]http://irrlicht.sourceforge.net

In molte realtà, però, si mira alla realizzazione di motori e tools meno "general purpose" e più orientati a quelle che sono le features particolari richieste dalle specifiche di gioco. Tra questi ve ne sono alcuni appositamente sviluppati in modo da lasciare un grado di libertà tale da poter essere riutilizzati, successivamente, per la realizzazione di nuovi titoli, eventualmente anche sviluppati da terze parti. Tra questi vi sono l'*id Tech* di *ID Software*, famoso per essere stato utilizzato come tecnologia di base per realizzare le serie di *DooM* e di *Quake* ed ormai giunto alla sua quinta versione, l'*Unreal Technology*, di *Epic Games*, utilizzato invece per la serie *Unreal* e per numerosi altri titoli per PC e console, ed il *CryEngine*, attualmente utilizzato esclusivamente per la serie *Crysis* ma tecnicamente ancora all'avanguardia.

Da qualche anno, inoltre, è aumentata la richiesta di Middleware crossplatform a basso costo che, oltre all'Engine 3D, offrano ambienti integranti Editors, Tools, Engines di gioco, Engines fisici, supporto audio e gestione degli input: *Unity3D* [8] ne è un esempio. Altrettanto validi sono *Shiva 3D*[9] e l'*UDK*[10].

Un discorso analogo lo si può fare per quanto riguarda i moduli relativi alla gestione di rete:

RakNet[11] è un Game Networking Engine crossplatform che oltre a features base a livello di comunicazione (NAT Punchthrough, ordinamento messaggi, serializzazione e generazione pacchetti) e a livello applicazione *Object replication e chiamate remote*, offre una serie di funzioni aggiuntive come la gestione utenti, gestione lobby, stanze e matchmaking. Supporta sia il modello Client-Server che quello Peer-To-Peer.

[8] http://unity3d.com
[9] http://www.stonetrip.com
[10] Unreal Development Kit, http://www.udk.com
[11] http://www.jenkinssoftware.com

SmartFoxServer[12], è un ambiente integrato con caratteristiche simili a RakNet ma orientate maggiormente verso la semplificazione di utilizzo: lato Server infatti dispone di una notevole quantità di strumenti di amministrazione per la gestione utenti, mentre lato Client mette a disposizione una serie di API per migliorarne l'integrazione.

Altre librerie invece omettono appositamente alcune features di più alto livello, in favore di una maggiore flessibilità di implementazione lato utente: *Zoidcom*[13], ad esempio, fornisce sistemi di Object replication e sincronizzazione degli stati, ma non supporta features legate alla gestione delle partite e dei giocatori. *Enet*[14], invece, fornisce solamente un leggero ma robusto Layer di comunicazione basato sul protocollo UDP.

[12]http://www.smartfoxserver.com
[13]http://www.zoidcom.com
[14]http://enet.bespin.org

Capitolo 2

Architettura Software

2.1 Problematiche

L'esigenza di un elevata reattività di risposta (logica e visiva) ai comandi del giocatore è sentita in quantità differente a seconda della tipologia di gioco: In un gioco *Arcade*[1], ad esempio, il *Timespan*[2] tra l'input "meccanico" dell'utente ed il relativo feedback visivo deve essere estremamente ridotto (si pensi al movimento del personaggio controllato dal giocatore in un *FPS*[3]), mentre in un gioco *TBS*[4] tutto questo non è necessario, dal momento che i comandi impartiti dal giocatore verranno eseguiti solo alla fine di un turno che può durare anche diversi minuti.

Queste differenze influenzano non solo il design degli algoritmi e delle strutture dati relativi alla logica specifica del *gameplay*[5], ma anche alcune scelte progettuali sul design del *Game Engine*[6]:

[1] Un videogioco caratterizzato da dinamiche rapide, immediate, e con ridotte pretese di realismo

[2] Lasso di tempo

[3] *First-person shooter*, Sparatutto in prima persona

[4] *Turn-based strategy*, Gioco strategico a turni

[5] Regole e dinamiche logiche e d'interazione che contraddistinguono il gioco

[6] Genericamente è inteso come l'insieme dei sistemi software che gestiscono il rendering, le animazioni, la logica, le collisioni, la scena, l'intelligenza artificiale, audio, input ed il

Riprendendo l'esempio precedente, e limitando il contesto ai soli giochi offline, al fine di realizzare un semplice gioco *Arcade* può non essere necessaria una forte separazione tra il Layer logico (Business Layer) ed il Layer di visualizzazione (Presentation Layer): lo spostamento di uno *Sprite*[7] che ne simula il movimento su schermo, può essere realizzato modificando direttamente le variabili relative alla posizione di disegno dello stesso, come immediata conseguenza dell'input dell'utente.
In un *TBS*, invece, i due Layer tendono ad essere disaccoppiati in modo naturale, portando il Layer di visualizzazione ad essere una mera rappresentazione dei dati logici che, in ultima analisi, sono gli unici detentori dello stato attuale di gioco.

Se però si estende l'esempio ampliandolo ai giochi online, allora anche per un gioco *Arcade* sarà opportuno seguire un pattern *Three-tier*[8] o *MVC-like*[9], dal momento che avere un Layer logico separato semplifica l'identificazione, la ricerca, l'accesso e quindi la sincronizzazione dello stato di gioco. Non solo: In questo contesto nascono problematiche relative alla latenza di rete, specialmente proprio in quelle tipologie di gioco che richiedono contemporaneamente massima reattività e massima consistenza.
Idealmente, quindi, anche quelle azioni che normalmente, in architetture *Singleplayer oriented*, producono un *feedback* diretto, come lo spostamento di un elemento grafico in conseguenza ad uno specifico input, devono qui essere processate attraverso diversi Layer prima di essere eseguite, eventualmente anche con un apposito ritardo introdotto volutamente da alcuni sistemi di sincronizzazione (si veda la sezione 4). Per alcune tipologie di gameplay, però, tale ritardo finisce con l'impattare negativamente sull'esperienza di gio-

networking

[7]Un qualsiasi elemento grafico, statico o dinamico, che compone la scena di gioco

[8]Architettura software caratterizzata da tre moduli: Interfaccia Utente, Logica Funzionale e Dati

[9]Model-View-Controller, un pattern architetturale che vede separati i concetti di Model: il sistema di accesso ai dati, View: Il sistema di visualizzazione dati, ed il Controller: Il sistema di modifica dei dati

co, rendendo necessaria l'implementazione di sistemi di interpolazione e di *Prediction* che ne attenuino gli effetti di latenza percepiti dall'utente (2.7.1). Laddove questo non bastasse, è possibile, uscendo però dal design proposto dalla pipeline illustrata precedentemente, parallelizzare tale esecuzione processandola immediatamente in un contesto locale, e, successivamente interpolandone gli effetti risultanti dall'esecuzione remota.

Astraendo dall'architettura di rete utilizzata, quello che si mira ad avere è uno stato di gioco "condiviso" da tutti gli attori partecipanti alla rete:
In un sistema *Client-Server* lo stato di gioco è mantenuto dal Server, ma una sua copia, tipicamente meno estesa e dettagliata, è presente comunque su ogni singolo Client. Questo accade specialmente in quelle tipologie di gioco che richiedono continui accessi al dato, poiché accedere direttamente ai Layer Business (o Data) del Server sarebbe deleterio in termini di performance, di sicurezza e, in ultima analisi, di design.
A maggior ragione, in un sistema *Peer-to-Peer* lo stato di gioco deve risiedere su ogni nodo, e questo introduce problematiche di sincronizzazione dello stato di gioco, rendendo necessaria l'implementazione di politiche atte a garantirne la consistenza. Su ogni nodo infatti risiede una copia dello stato di gioco sul quale vengono operate modifiche in modo indipendente a seconda degli eventi ricevuti da altri nodi in relazione alla logica di gioco implementata.

Se la politica di sincronizzazione adottata prevede la possibilità di eseguire dei *Rollback*, tale copià può anche essere mantenuta su più istanze, mirate a fornire una sorta di backup dal quale attingere in caso di *Rollback* le informazioni necessarie a ripristinare lo stato precedente (si veda Figura 1.6).

Come anticipato nell'introduzione, per l'implementazione del progetto di tesi si è scelto di limitare l'inconsistenza utilizzando un approccio che sotto alcuni aspetti si discosta da un'architettura *Peer-To-Peer* "pura".
Il sistema di sincronizzazione utilizzato predilige una politica più "State-Driven" che "Event-Driven": Infatti ogni nodo non sarà perfettamente pa-

ritetico: manterrà una sua visione del Data Layer, ma opererà solo su un limitato numero di dati di sua appartenenza.

Questo approccio coinciderà con un Data Layer nel quale ogni singola informazione atomica è di proprietà di un generico "Owner", ed è quindi vincolata da una serie di regole che ne garantiscono i permessi esclusivi di creazione, modifica e distruzione. Nella fattispecie l'Owner coinciderà con i concetti di Player e di Peer.
Quest'ultimo è concretizzato nel Layer dedicato alla gestione della Membership dei nodi che partecipano alla rete. Oltre a garantire la mutua conoscenza tra i diversi Peer, astrae il Layer NET sottostante, dedicato alla comunicazione di rete vera e propria.
Riassumendo, il software sarà composto da i seguenti Layer e Moduli:

- Layer View (Presentation Layer): Dedicato alla rappresentazione visuale dello stato logico. Oltre agli aspetti prettamente grafici, impropriamente include i moduli dedicati alla gestione della fisica, dell'audio e degli input.

- Layer Logic (Business Layer): Dedicato all'implementazione delle meccaniche di gioco. Sostanzialmente modifica il Layer Data a seconda degli Input che giungono dal Layer View facendo riferimento alle regole di gioco.

- Modulo SharedComponents (Data Layer e Consistenza): Mantiene lo stato di gioco. Espone un Manager in grado di creare, modificare e distruggere i propri componenti, e di mantenere una copia dei componenti creati dai Manager appartenenti agli altri Peer.

- Layer P2P (Membership): Dedicato alla gestione della Membership. Astrae l'interfaccia del Layer NET sottostante.

- Layer NET (Communication): Espone un'interfaccia che consente la comunicazione tra N generici endpoints.

Questa separazione di Layer e le loro caratteristiche sono state progettate per avere un sistema il più possibile generico, modulare, e, specialmente nel Layer inferiori, "gamelogic independent".

La tipologia di gioco scelta per il progetto di tesi, è identificabile all'interno della categoria degli "Shooter Games", che presenta un Gameplay dalle logiche relativamente semplici ed individuate da una serie di elementi caratteristici del genere: La triade di elementi fondamentali di uno "Shooter" è formata dal concetto di "Protagonista", che può essere sia un elemento antropomorfo (un soldato) che un elemento meccanico (una navetta spaziale) controllato direttamente dal giocatore; dal concetto di "Nemico", che può essere un elemento controllato dalla CPU o direttamente da un altro giocatore, e da un'entità in grado di apportare danno al "Nemico" conseguentemente ad una azione ad opera del "Protagonista", o viceversa.

La scelta è ricaduta su questo genere di gioco poiché, nella sua semplicità, presenta implicitamente tutta una serie di problematiche comuni alla maggior parte delle tipologie di gioco, ovvero la creazione e distruzione di oggetti grafici, la gestione del loro spostamento e delle eventuali collisioni.

Di seguito verrà analizzato ogni singolo Layer nel dettaglio, lasciando però gli aspetti prettamente implementativi al prossimo capitolo.

2.2 Layer NET

Questo modulo è necessario per fornire una astrazione di quella che è la comunicazione di rete a più basso livello. Quest'ultima infatti può differire molto da implementazione a implementazione, a partire dal protocollo di comunicazione fino alla struttura dei singoli pacchetti, mentre i Layer sovrastanti necessitano di un'interfaccia astratta e trasparente in questo senso. A questo proposito, il Layer espone, tramite un'interfaccia, i concetti di "*Node*"

e di "*Link*". Un "*Node*" è sostanzialmente un endpoint di comunicazione, mentre un "*Link*" è una rappresentazione del canale ci comunicazione tra due endpoints.
In un contesto Peer-To-Peer formato da N nodi, significa che ogni "*Node*" possiede N-1 "*Link*", uno per ogni nodo appartenente alla rete (escluso se stesso).
Da un punto di vista più vicino all'implementazione vera e propria, sia il "*Node*" che il "*Link*" sono considerati "*NetElement*", ovvero elementi identificabili tramite un "*NetID*", costruito in base all'indirizzo IP e alla porta di ascolto.

Quando si vuole connettere un nodo A ad un nodo B in ascolto, in A viene creato un Link identificato dal NetID di B. Quando B riceve la richiesta di connessione, può ricavare l'indirizzo IP di A, ma non la porta sulla quale sta ascoltando.
Inizia quindi una fase di *Handshaking*, eseguita a livello applicazione, nella quale i due nodi si scambiano i propri NetID, identificando gli attori e consolidando così la connessione.
La fase di Handshaking inizia appena un nodo (A) riceve la connessione di un secondo nodo (B).
Il primo step vede A inviare il comando NET "*HandShakeBegin*" a B.
Quando B riceve tale comando, risponderà inviando ad A il comando NET "*HandShakeResponse*" contenente il proprio NetID.
L'ultimo step vede infine A confermare a B l'avvenuto riconoscimento tramite il comando NET "*HandShakeEnd*".
Una volta terminata questa fase (Figura 2.1), entrambi i nodi possono iniziare a scambiarsi i messaggi inoltrati dai rispettivi Layer superiori.

Figura 2.1: Fase di connessione tra due nodi e relativo Handshaking, eseguito all'interno del Layer NET

2.3 Layer P2P

Una prima funzione di questo Layer è quello di astrarre quello sottostante con una serie di funzioni di alto livello che permettono la comunicazione tra i nodi secondo una logica Peer-To-Peer senza conoscerne la specifica implementazione. Ad esempio la funzione *"BroadCast(messaggio)"* si preoccuperà di inviare lo stesso messaggio, attraverso il Layer NET, a tutti i Peer conosciuti.
La seconda funzione di questo Layer è quella di garantire la conoscenza reciproca tra i diversi Peer: ogni Peer infatti contiene una lista di 'NetID' relativi ai Peer con i quali risulta già connesso.

Supponendo di avere 5 Peer, A, B ,C , D ed E, e che siano già connessi tra loro secondo lo schema seguente (Figura 2.2): (A-B) , (C-D) , E

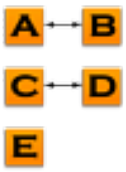

Figura 2.2: Organizzazione delle connessioni all'istante T1

- Il Peer A ha nella sua lista di Peer conosciuti il NetID di B (ed il proprio)
- Il Peer B ha nella sua lista di Peer conosciuti il NetID di A (ed il proprio)
- Il Peer C ha nella sua lista di Peer conosciuti il NetID di D (ed il proprio)
- Il Peer D ha nella sua lista di Peer conosciuti il NetID di C (ed il proprio)

- Il Peer *E* ha nella sua lista di Peer conosciuti solo il proprio NetID

Quando *E* si connette al Peer *D*, teoricamente entrerebbe a far parte della rete formata dai Peer *C*,*D* ed *E*. Ma *E* non è a corrente dell'esistenza del Peer *C*, e viceversa, *C* non è a corrente dell'esistenza di *E*.
Per ovviare a questo problema, una volta terminata la fase di *HandShaking* (a carico del Layer NET) tra i rispettivi nodi di *D* ed *E*, viene avviata una seconda fase a carico del Layer P2P, nella quale i Peer appena connessi, si scambiano tutti i NetID di loro conoscenza. In questo modo grazie a *D*, *C* conoscerà *E*, vi si connetterà, e di conseguenza *E* conoscerà *D*, chiudendo di fatto la sottorete (C-D-E) (Figura 2.3).

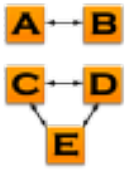

Figura 2.3: Organizzazione delle connessioni all'istante T2

Quando *C* si connette a *B*, *B* notificherà l'esistenza di *C* ad *A* che vi si connetterà a sua volta. Allo stesso modo, *C* riferirà a *D* ed *E* dell'esistenza di *B* e quindi di *A*. Infine *D* ed *E* si connetteranno entrambi ad *A* e *B* (Figura 2.4).

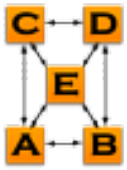

Figura 2.4: Organizzazione delle connessioni all'istante T3

In (Figura 2.5) sono mostrate le fasi di connessione tra due Peer, sia del Layer NET che del Layer P2P *A* e *B*.

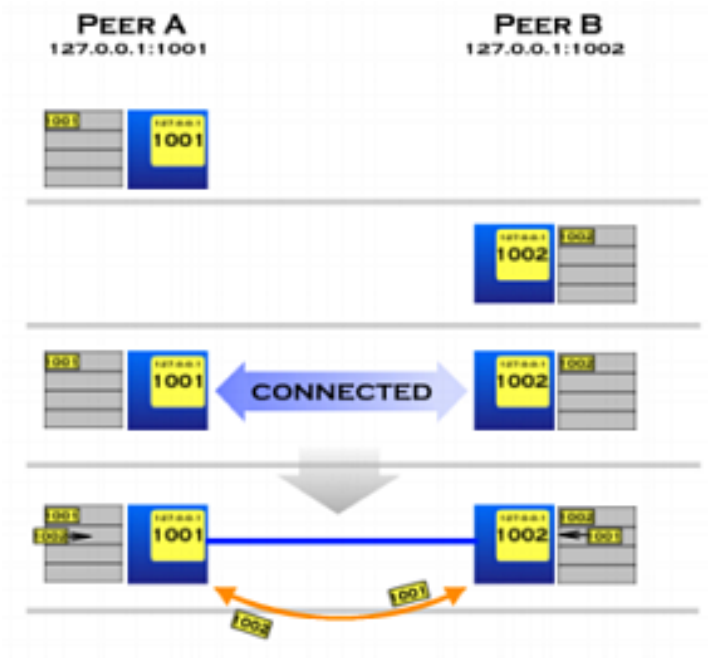

Figura 2.5: Fasi preliminari di connessione tra due Peer

Ad un istante *T0*, il Peer *B* si connette al Peer *A*.
All'istante immediatamente successivo, nel contesto dell Layer NET, avviene la fase di *HandShaking*.
Quando questa termina, nel contesto del Layer P2P, si attiva la fase di interscambio dei NetID conosciuti. In questo caso *B* conosce solo *B* (se stesso) ed *A* conosce solo *A* (se stesso).
B invierà quindi ad *A* il comando P2P "*AddPeer*" parametrizzato con il proprio NetID. *A* farà lo stesso, inviando a *B* il comando P2P "*AddPeer*" parametrizzato con il proprio NetID.
Quando *A* riceverà il comando P2P "*AddPeer*" contenente il NetID di *B*, controllerà che questo non sia già presente nella propria lista dei Peer conosciuti: nel caso non sia effettivamente presente, aprirà automaticamente una connessione verso il NetID appena ricevuto.
Non è questo il caso, dal momento che i Peer sono solo due, ed entrambi si conoscono a vicenda direttamente dalla fase di *HandShaking*.

Se però nell'esempio viene introdotta una seconda sottorete formata dai Peer *C* e *D* (Figura 2.6), questa fase assume maggior concretezza: in *T1*, il Peer *C* si connette al Peer *B*, ed in *T2* viene attivata la fase di scambio NetID, implementata tramite il comando P2P "*AddPeer*" (freccia in arancione):

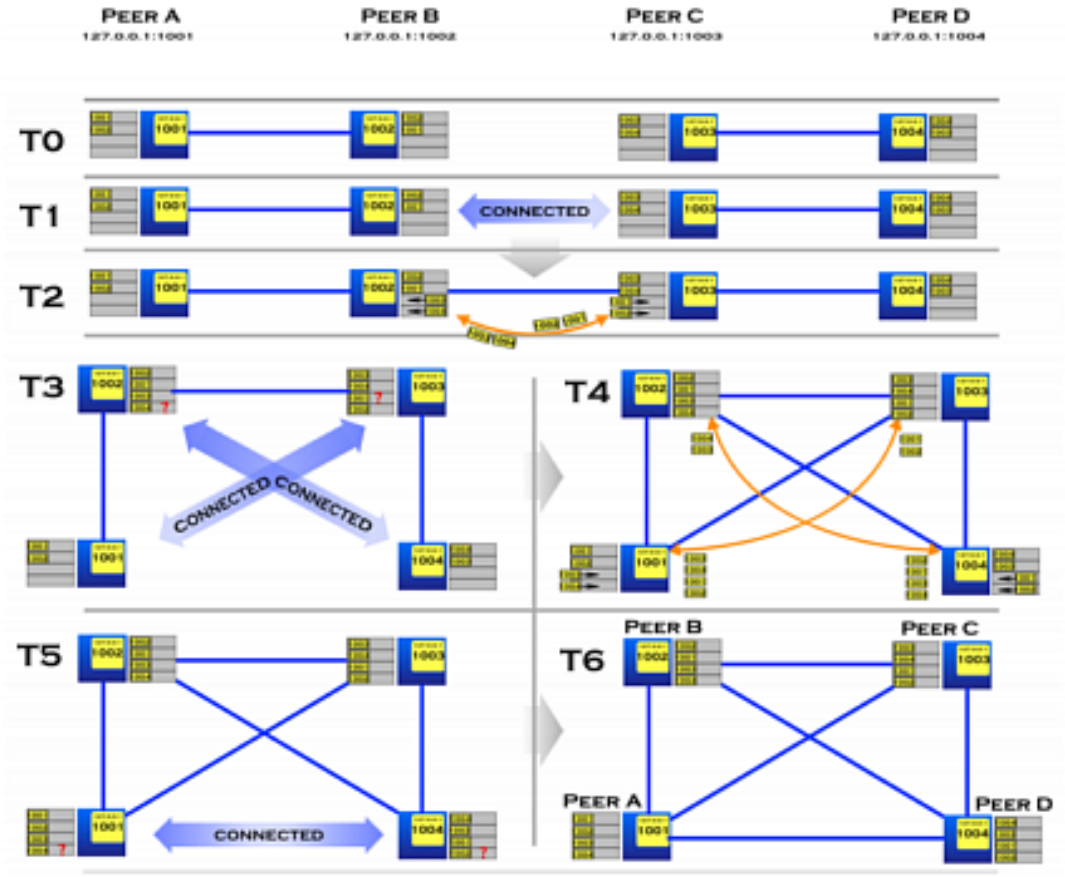

Figura 2.6: Fasi preliminari di connessione tra due sottoreti composte ciascuna da due Peer

Il Peer *C* ha, nella lista di Peer conosciuti, se stesso ed il Peer *D*. Pertanto invierà a *B* due comandi P2P "*AddPeer*"; il primo parametrizzato con il proprio NetID, ed il secondo parametrizzato con il NetID di *D*.
Quando il primo comando "*AddPeer*" giungerà al Peer *B*, quest'ultimo controllerà che il NetID contenuto come parametro del comando stesso, non sia

già presente nella propria lista di Peer conosciuti.
In questo caso B è già a conoscenza dell'esistenza di C, dal momento che la connessione è avvenuta proprio tra B e C.
Quando però arriva il secondo comando "*AddPeer*", parametrizzato con il NetID di D, il Peer B, controllando nella propria lista di Peer conosciuti, si accorge di non conoscere il Peer identificato dal NetID di D. A questo punto, quindi, Il Peer B si connetterà al NetID di D per includerlo tra i propri Peer conosciuti.

Si attiva quindi una propagazione "*a catena*" che terminerà solo una volta che tutte le liste di Peer conosciuti coincideranno tra loro.

2.3.1 Modulo Time

All'interno del Layer P2P, è presente un sotto-modulo che si occupa della gestione del *Timing*, mirato a garantire una visione temporale condivisa.

Una volta terminata la fase di consolidamento della Membership, inizia una fase parallela e prolungata nel tempo: ogni Peer notificherà a tutti gli altri Peer conosciuti, quello che è il valore del proprio *TimeStamp* di sistema attraverso il comando *TimeSync*.
Quando un Peer A riceverà un comando *TimeSync* da un Peer B, calcolerà il delta tra il *TimeStamp* in esso contenuto ed il proprio, e lo memorizzerà in un'apposita lista indicizzata per NetID. Il valore del delta deve inoltre comprendere un TimeSpan che compensi il fattore Lag: viene infatti aggiunto un lasso temporale calcolato in base al tempo di latenza tra la richiesta *TimeSync* e la relativa risposta.

Dal momento che questa fase viene reiterata (dopo un determinato periodo di tempo viene infatti eseguito nuovamente un BroadCast di *TimeSync*), ogni Peer conoscerà con una discreta precisione il fattore di conversione del *TimeSpan* di tutti gli altri Peer, rispetto al proprio.

Questo modulo, quindi, permette di convertire in maniera del tutto trasparente il *TimeStamp* del mittente, che viene appeso a tutti i comandi del Layer Logic, utilizzando come base di conversione il proprio *TimeStamp*.

Non vi è quindi un sistema di *Master Clock* assoluto, associato ad un Peer eletto tale, ad una risorsa esterna alla Membership od un sistema centralizzato Client-Server, come accade per il più accurato *NTP*[10], ma un sistema basato sulla conversione di *TimeSpan* relativi, simile a quanto avviene nella la conversione di orari provenienti da fusi orari differenti (Figura 2.7 e Figura 2.8).

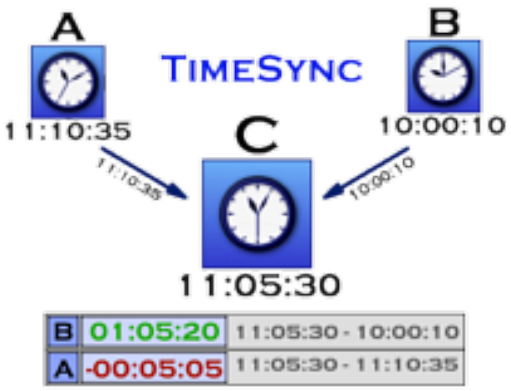

Figura 2.7: I Peer *A* e *B* inviano il proprio *TimeStamp* a *C*, che ne memorizza il *TimeSpan* rispetto al proprio

Questo sistema risulta essere soggetto a problemi derivanti dalla variazione di latenza (Jitter) tra i diversi Peer: i delta sono infatti destinati a subire oscillazioni ad ogni aggiornamento, rendendo la stima temporale piuttosto approssimativa.
L'accuratezza richiesta nelle applicazioni real-time, specialmente nelle situazioni concorrenziali, viene qui mitigata dal sistema implementato nel Modulo SharedComponents (sezione 2.4): Grazie a quest'ultimo non si possono presentare casi di inconsistenza dello stato di gioco dovuti al margine di errore

[10]Network Time Protocol

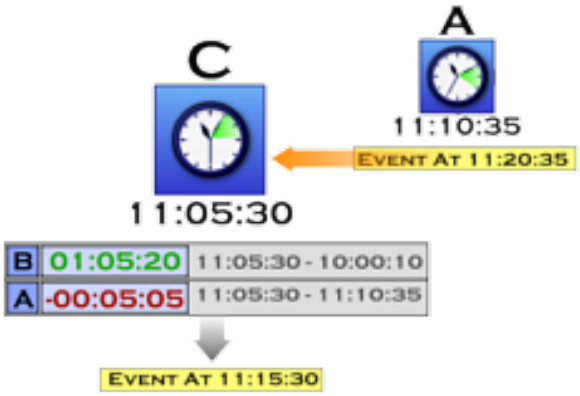

Figura 2.8: Quando arriva un comando o un evento temporizzato dal Peer *A* secondo il suo sistema di riferimento, al Peer *C*, quest'ultimo converte il *TimeStamp* secondo il delta precedentemente memorizzato

temporale relativamente elevato, poichè situazioni temporalmente sensibili (ad esempio una collisione con un proiettile) per le quali due peer modificano la stessa risorsa sono opportunamente vincolate dal modulo in questione: l'assenza di un qualche tipo di sincronizzazione temporale (4), anche se non impatta sulla consistenza dello stato di gioco, tende a penalizzare notevolmente l'esperienza di gioco dei Peer con alta latenza (si veda Figura 2.28 ed il capitolo 4).

2.4 Modulo SharedComponents

Questo modulo fornisce la base per il Data Layer, ed è completamente disaccoppiato dai Layer precedenti (NET e P2P): si presenta come un piccolo Framework dedicato alla rappresentazione di una sorta di "memoria" di alto livello sulla quale memorizzare lo stato di gioco (Figura 2.9), riprendendo in parte l'idea delle Distributed Hash Table (DHT).

Si basa sul concetto di *componente*, inteso come il minimo elemento in grado di apportare informazione sullo stato di gioco:

Il punteggio di un giocatore, ad esempio, può essere visto come un componente, così come la posizione del suo personaggio all'interno dell'ambiente di gioco.
La stessa posizione, a sua volta è composta da 3 sotto-componenti (X, Y e Z) secondo il *Composite Pattern*[11]

I componenti che riescono a rappresentare informazioni atomiche, senza doversi appoggiare a sotto-componenti, sono da considerarsi "leaf", e sono tipicamente quei componenti che emulano variabili *Value Type* (ad esempio il componenti "Int" , "Float" o "String").
Tutti gli altri componenti (come appunto la posizione del personaggio, che sarà concretizzata da un componente di tipo "Vector3") sono aggregati, diretti o indiretti, di quest'ultimi.

Ogni componente soggiace a precise regole che ne consentono una facile gestione non solo all'interno del modulo stesso, ma specialmente se quest'ultimo viene utilizzato come Data Layer all'interno di un'applicazione di rete:

- Ogni componente (*parent*) può essere composto da altri componenti (*child*)
- Ogni componente deve essere generato da un apposito *Factory*[12] (identificato da un nome), e non può essere istanziato in modo indipendente.
- Ogni componente è identificato da un ID "globale" generato dallo stesso Factory al momento della creazione.

[11] Uno dei Pattern fondamentali definito dalla *gang of four* caratterizzato dalla definizione di un oggetto mediante la composizione di altri oggetti (a formare una struttura ad albero).

[12] Secondo l'omonimo pattern creazionale, con Factory si intende un elemento che incapsula al suo interno i processi coinvolti nella creazione di oggetti.

- Ogni componente può avere un ID “locale”, se child di un parent, che ne identifica la chiave con la quale il parent accederà al child.
- Ogni componente appartiene ad un “Type” che ne identifica il tipo (“Int”, “Float” ,“String” , “Vector3” , “Object3D”, etc)
- Ogni componente ha un valore serializzabile.
- Ogni componente è caratterizzato da un TimeStamp di creazione/modifica

In questo modo è possibile modellare qualsiasi tipo di informazione, non solo dati relativi agli elementi di gioco, ma anche eventi, messaggi e sotto-stati, ed identificarla quindi univocamente.

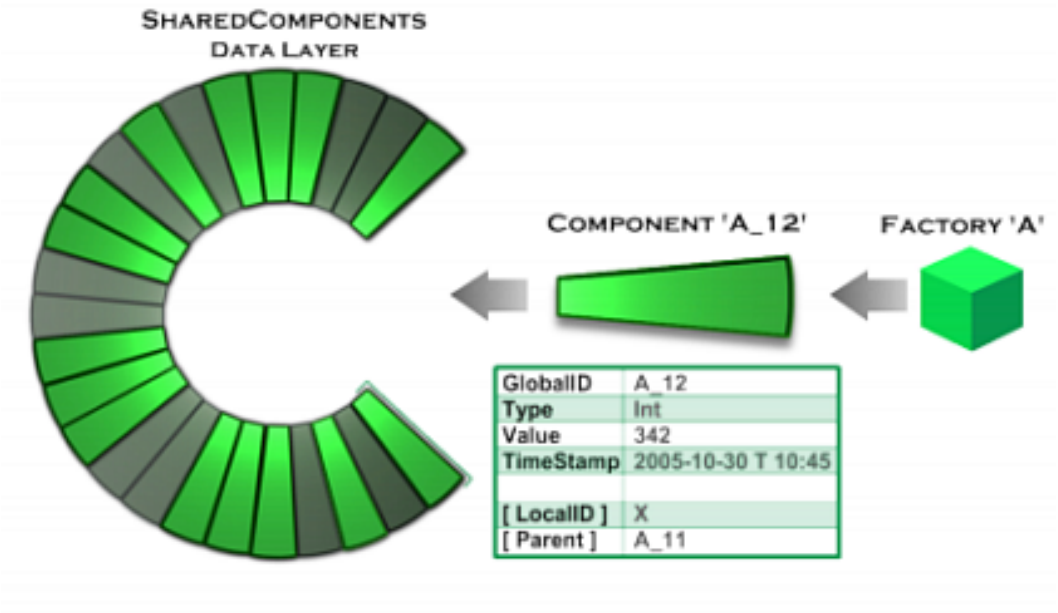

Figura 2.9: Schema dei concetti esposti dal modulo SharedCompontes. I componenti verde chiaro rappresentano i componenti generati dal Factory A, mentre i componenti verde scuro quelli generati da altri Factory.

Il Factory dispone di diversi sistemi di creazione di Componenti:
Il primo, quello diretto, istanzia un oggetto componente e lo “marchia” come

appartenente al proprio Manager. Questo sistema viene utilizzato quando da un Layer esterno si vuole creare un nuovo componente all'interno del proprio contesto, specificandone solo il tipo ed eventualmente il valore di default.
Il secondo invece viene utilizzato per istanziare "copie" di componenti appartenenti ad altri Factory, eventualmente associati ad altri Peer (associazione che verrà implementata nel prossimo Layer Logic): il componente così creato non sarà di proprietà del Factory, e non sarà "marchiato" come tale, dal momento che possiede già l'identificativo assegnatogli dal Factory originale.
Il terzo permette di aggiungere un generico componente come "sotto-componente" di un componente già esistente all'interno del contesto.

E' importante notare quindi come, all'interno del modulo SharedComponents, la lista dei componenti non sia popolata necessariamente da componenti generati dal "proprio" Factory.

Se ad esempio si vogliono modellare le informazioni che caratterizzano un oggetto tridimensionale, si possono utilizzare i seguenti componenti secondo la gerarchia proposta in Figura 2.10:

Il componente di tipo *Object3D* è caratterizzato da due sotto-componenti, rispettivamente identificati dal LocalID "*Position*" e "*Rotation*" (non è stato inserito un componente relativo al valore di *Scaling* dell'oggetto perché è un parametro legato maggiormente al Layer VIEW ed eventualmente può essere comunque incluso estendendo questo componente). Entrambi sono componenti di tipo *Vector3* che, come illustrato precedentemente, è composto a sua volta dai componenti di tipo *Float*, "*X*","*Y*" e"*Z*".

Un componente non "leaf" viene considerato "*ready*" solo quando tutti i suoi componenti sono stati istanziati e risultano "*ready*" a loro volta: un Object3D, quindi, non sarà considerato "*ready*" fintanto che non lo saranno i suoi due Vector3 "*Position*" e "*Rotation*";

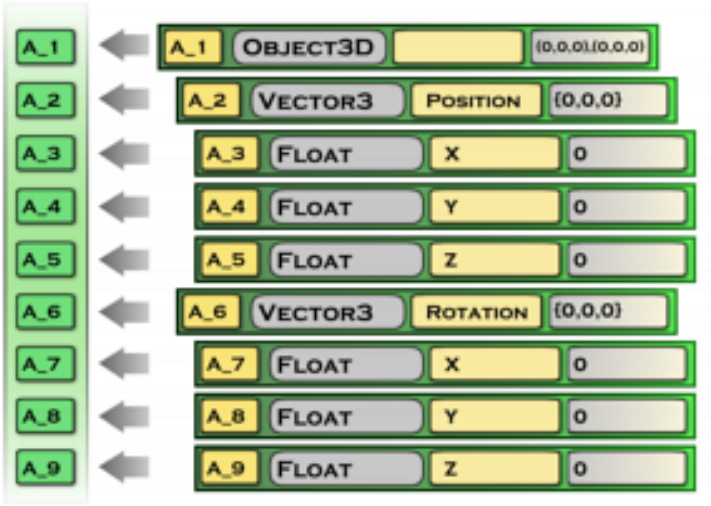

Figura 2.10: tassonomia dei sotto-componenti che, aggregati, realizzano un componente di tipo Object3D

Questa "frammentazione" dell'informazione risulta molto comoda specialmente se la si pensa contestualizzata all'interno di un'applicazione di rete, come emergerà nel prossimo capitolo.
I comandi principali a carico di questo Layer sono:

- CreateComponent: Crea un componente tramite il Factory utilizzando il sistema di "copia" di componenti appartenenti ad altri Factory, in quanto si presume che, dal momento che tale comando è giunto da un'altro contesto, il componente da creare appartenga a quest'ultimo.

- UpdateComponent: Aggiorna un componente secondo il valore serializzato contenuto come parametro

- DestroyComponent: Distrugge un componente.

- ActivateComponent: Attiva, cioè modifica l'attributo "*ready*" di un componente.

È presente, inoltre, un sistema di *Messaging* tra i diversi componenti: ogni componente infatti può implementare il metodo *ExecMessage()*, e, a seconda di quanto riportato come parametro, comportarsi di conseguenza.

2.5 Layer Logic

In questo Layer prende forma il *contesto* logico: legando il concetto di Peer esposto dal Layer P2P con il Factory del DataLayer esposto dal modulo SharedComponents nasce l'idea di un contesto di dati appartenenti al singolo Peer.
Ogni componente creato direttamente dal Factory risulterà "legato" al relativo Peer, e di conseguenza potrà essere modificato o distrutto solo da quest'ultimo.

A questo Layer, inoltre, è delegata un'altra funzione fondamentale, senza la quale l'utilità del modulo P2P e del modulo SharedComponents sarebbe fine a se stessa: rimane in ascolto delle notifiche (creazione/modifica/distruzione componenti) in uscita dal modulo SharedComponents, preoccupandosi poi di inoltrarle al Layer P2P che, mediante il Layer NET, le segnalerà a sua volta a tutti i Peer conosciuti.

Questo porta sostanzialmente ad avere lo stesso DataLayer (cioè popolato dagli gli stessi componenti) su ogni Peer, e le regole che definiscono i permessi di creazione/modifica/distruzione garantiscono la consistenza, dal momento che ogni componente può cambiare di stato solo tramite il suo Peer 'Owner'.
Quando un Peer A si connette ad un Peer B, dopo le fasi di HandShaking (Layer NET) e di scambio NetID (Layer P2P), A invierà preventivamente a B i componenti serializzati di sua proprietà presenti nel proprio SharedComponents, e B farà lo stesso.
Quando i componenti serializzati di A arrivano a B, il Factory di quest'ultimo creerà di fatto una "copia" del Data Layer di A (*Object Replication*).

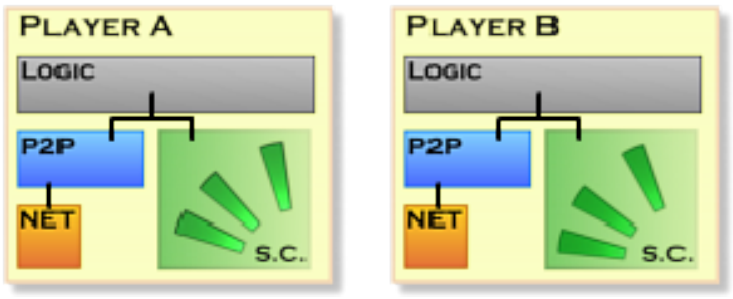

Figura 2.11: Due Player che non si sono ancora connessi tra loro. Notare come il Layer Logic utilizzi il Layer P2P ed il modulo SharedComponents per concretizzare il "contesto" logico. Notare inoltre come il contenuto del Data Layer sia differente tra i due Player.

Figura 2.12: Layer NET: I due Player si connettono e avviano la fase di HandShaking.

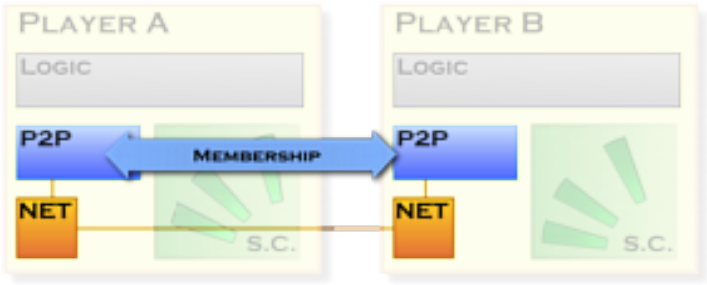

Figura 2.13: Layer P2P: I due Player si scambiano i NetID conosciuti.

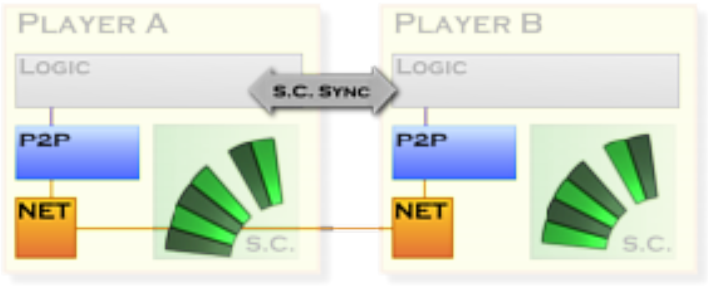

Figura 2.14: Layer Logic: I due Player si scambiano il contenuto corrente dei propri Data Layer. Notare come, conclusa questa fase, il contenuto dei due Data Layer risulti identico.

Una volta che i due Player si sono scambiati il contenuto dei rispettivi Data Layer, sono pronti per inviarsi messaggi relativi alla logica di gioco.

Ad esempio, quando il personaggio controllato dal Player *A*, connesso ad un Player *B*, spara un proiettile, crea di fatto un componente logico di tipo "*Bullet*". Il componente "*Bullet*" estende *MovingObject3D*, che a sua volta estende il componente *Object3D*.
MovingObject3D ha gli stessi sotto-componenti di *Object3D* (*Vector3 Position* e *Vector3 Rotation*), con l'aggiunta di altri due nuovi sotto-componenti: Un *Vector3 NextPosition* ed un *String NextTimeStamp*. Il primo (*NextPosition*) indica la posizione dell'oggetto 3D attesa per l'istante di tempo espresso dal secondo (*NextTimeStamp*).
La nuova istanza del componente di tipo *Bullet* viene creata all'interno del contesto logico di *A*, attraverso il suo Factory e depositato nel proprio Data Layer.
In questo momento i Data Layer di *A* e *B* sono desincronizzati, dal momento che l'istanza del componente *Bullet* è presente solo nel Data Layer di *A*.
Quando avviene una creazione diretta di questo tipo, il modulo SharedComponents notifica ai Layer in ascolto (il Layer Logic, ad esempio) l'avvenuta creazione del nuovo componente.

Il Layer Logic quindi esegue un BroadCast esteso a tutti i Peer conosciuti inviando il comando *CreateComponent* appositamente parametrizzato per la descrizione del componente Bullet.

Ogni comando di questo Layer viene marcato con l'attuale *TimeStamp* esposto dal sotto-modulo *Time* del Layer *P2P*.
In questo modo si possono implementare controlli basati sul tempo: quando un comando gestito dal Layer Logic viene ricevuto da un Nodo, questo viene inserito in una lista di esecuzione, ed eseguito solo all'iterazione successiva, o, nel caso di comandi esplicitamente temporizzati, solo quando il *TimeStamp* del Peer destinatario sarà maggiore o uguale a quello presente all'interno del comando stesso (opportunamente convertito in modo automatico dal sotto-modulo *Time* del Layer P2P).

Questo tipo di controlli viene effettuato in un sotto-modulo composto da una serie di *Filtri* che intervengono tra la ricezione del comando e la sua esecuzione. Uno di questi filtri controlla l'*Obsolescence* dei comandi UpdateComponent:
Può capitare infatti di ricevere due messaggi UpdateComponent di uno stesso componente riportanti valori differenti. In questo caso viene confrontato il *TimeStamp* che accompagna entrambi i messaggi, ed eliminato l'UpdateComponent temporalmente più vecchio.
Escludendo situazioni di estrema congestione, si tratta comunque di un'evenienza abbastanza rara, in quanto due UpdateComponent di uno stesso componente possono essere inviati solo dal medesimo Peer, e quest'ultimo tipicamente effettua aggiornamenti di componente con una frequenza più bassa rispetto a quella di esecuzione.

Questo Layer inoltre mette a disposizione un'interfaccia che permette di inviare messaggi sia ai propri componenti che a quelli appartenenti agli altri Peer, utilizzando il sistema di *Messaging* del modulo *SharedComponents*.

2.6 Layer View

A causa di alcune scelte implementative, esposte nel prossimo capitolo, il Layer View si presenta come un insieme di moduli che gestiscono i diversi aspetti di gioco non ancora affrontati nei Layer sottostanti.
Qui risiede il sotto-modulo *Renderer*, fondamentale per la rappresentazione e visualizzazione dello stato di gioco; ma vi si possono anche trovare sotto-moduli non prettamente dedicati all'aspetto visivo: il modulo *Audio*, il modulo che gestisce l'*Input* utente, il modulo che gestisce i movimenti, le animazioni, la fisica e le collisioni.
Si tratta di uno strato piuttosto eterogeneo, quindi, che deve il suo nome più alle similitudini con il Layer View del pattern *MVC*, che alla sua funzione vera e propria.
Così come il Layer Logic introduceva il concetto di "Contesto logico", questo Layer introduce il concetto di "Contesto visuale" concretizzato in quello che, nell'architettura di un Game Engine, viene comunemente chiamato *SceneGraph*.
Con *SceneGraph* si intende quella struttura dati che contiene i riferimenti a tutti gli elementi grafici presenti nell'ambiente di gioco (*Scene*).
Può anche essere implementata come Array o Plain List, ma tipicamente viene realizzata mediante una struttura ad albero, dal momento che, quest'ultima, torna particolarmente comoda per la propagazione delle trasformazioni geometriche per tutti quegli elementi grafici che in qualche modo sono legati tra loro da una precisa gerarchia.

Questo tipo di struttura, inoltre, è perfetta per automatizzare il processo di sincronizzazione tra il DataLayer, esposto dal Layer Logic, ed il Layer View.
Tale processo "legge" il contenuto del Data Layer e ci costruisce sopra la scena 3D a seconda dei componenti presenti in esso.

Chiaramente non tutti i componenti si possono convertire automaticamente in oggetti grafici. Un componente di tipo *Int*, ad esempio, non può

essere convertito direttamente, dal momento che si tratta di un tipo di variabile troppo generico, e non è possibile quindi accostarlo sempre ad un unico tipo di elemento grafico: in un *FPS*, sia il valore dell'*HP*[13] che il numero di munizioni disponibili sono rappresentati nel Data Layer attraverso componenti di tipo *Int*, ma a livello grafico possono essere visualizzati in maniera del tutto differente tra loro.
Quindi, per questo tipo di componenti, è necessario creare manualmente gli elementi visuali ed implementare una logica di sincronizzazione con il rispettivo componente del Data Layer (di solito mediante sistemi che utilizzano l'*Observer Pattern*).

Esistono però alcuni componenti che possono essere inseriti in un logica di automazione di tale processo. È il caso dei componenti di tipo *Object3D* e dei suoi derivati.
Un componente di tipo *Object3D* verrà trasformato in un oggetto grafico di tipo *Object3D*. Allo stesso modo un componente di tipo *Bullet* (si veda esempi precedenti) verrà trasformato in un oggetto grafico di tipo *Bullet*.
In generale ogni componente taggato, a livello di definizione di classe, con il tag "*View*", quando esiste una corrispondenza 1-a-1 tra componente Data Layer e classe View, soggiace a questo tipo di automatismo: basterà infatti creare un componente logico per vederlo apparire su schermo all'iterazione successiva (Figura 2.15).

Il Layer View non si limita solo a rappresentare il contenuto del Layer Logic e del Data Layer, operando quindi in sola lettura, ma, anzi, è da qui che tipicamente vengono lanciate le operazioni di scrittura e aggiornamento dei componenti logici:
Ad ogni elemento grafico generato automaticamente dal contesto visuale, viene "applicata" un'istanza di una classe, estesa da "*ComponentScript*" che ne definisce il comportamento.

[13]Health Point, punti vita / punti ferita. Unità di misura che rappresenta lo stato di salute o il danno subito dal protagonista in molte tipologie di gioco

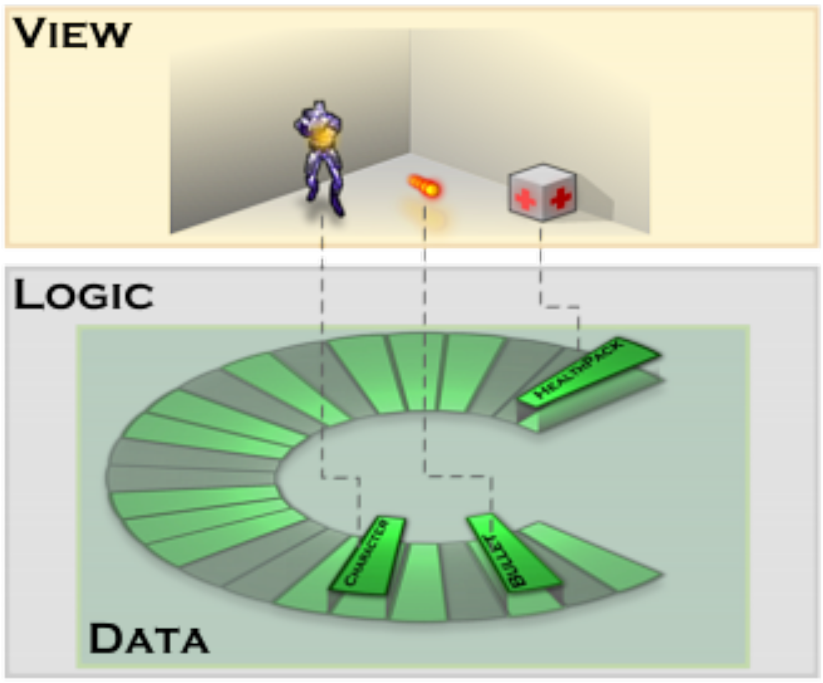

Figura 2.15: Il contesto visuale istanzia automaticamente elementi grafici in relazione ai componenti presenti nel Data Layer

"*ComponentScript*" è una classe base per l'implementazione della logica e delle dinamiche di aggiornamento dei componenti; a sua volta eredita dalla classe Behavior, e permette di definire le classiche fasi di "*Inizializzazione*", "*Aggiornamento*" e "*Distruzione*". La sua peculiarità è quella di consentire una separazione formale di quelle quelle che sono le azioni e gli aggiornamenti da eseguire in ogni singola fase a seconda che si tratti di un componente appartenente al proprio Peer o meno.
Ad ogni "*ComponentScript*" viene associato il GlobalID del componente di cui si vuole definirne la logica di aggiornamento.

Ad esempio, Il componente logico di tipo "*Character*" (esteso da "*Object3D*") identifica lo stato del personaggio comandato dal giocatore. Questo viene tradotto automaticamente nel rispettivo elemento grafico, ma le politiche di aggiornamento e sincronizzazione vengono definite dalla classe "*CharacterScript*", estesa appunto da "*ComponentScript*".
Qui, a seconda della proprietà del componente, CharacterScript esegue due metodi distinti. Per la fase di Update, ad esempio, il Peer "Owner" eseguirà il metodo "*Update_Writer*", mentre tutti gli altri eseguiranno il metodo "*Update_Reader*".
Siano Peer "*A*" e Peer "*B*" due Peer connessi e sincronizzati a livello di Data Layer, e sia il componente "*A_12*" di tipo "*Character*" il componente che identifica lo stato relativo al personaggio controllato dal Player "*A*", il "*ComponentScript*" applicato alla relativa istanza grafica si dovrà comportare in modo differente nel contesto View dei due Peer.
Su Peer "*A*", infatti, dovrà aggiornare le coordinate dell'oggetto 3D a seconda degli input del giocatore (ad esempio, su pressione dei tasti direzionali, o W-A-S-D, verrà mosso l'oggetto), e, a intervalli di tempo prefissati, *scrivere* le sue attuali coordinate all'interno del rispettivo componente logico.
Su Peer "*B*", invece, dovrà *leggere* il valore delle coordinate valorizzate nel rispettivo componente logico, ed interpolarle con le coordinate dell'oggetto3D del contesto View (Figura 2.16).

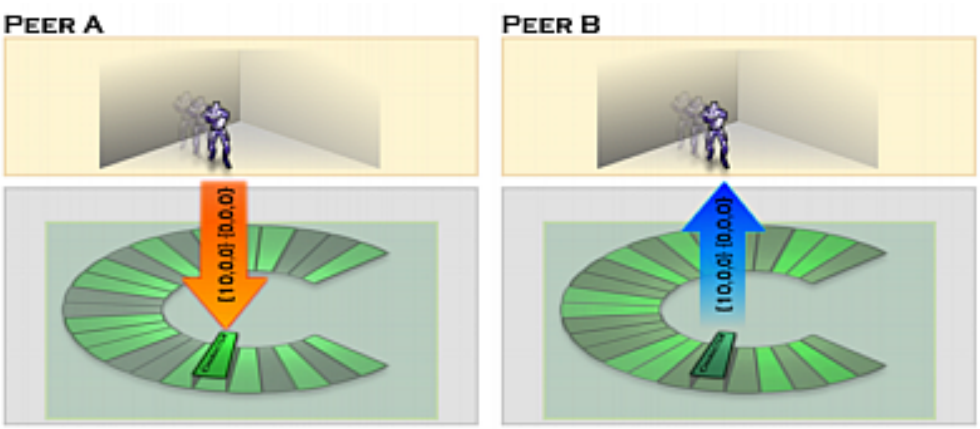

Figura 2.16: Il contesto visuale istanzia automaticamente elementi grafici in relazione ai componenti presenti nel Data Layer

L'aggiornamento delle coordinate del componente di tipo "*Character*" può essere effettuato in diversi modi. Come evidenziato nel capitolo relativo al modulo SharedComponents, è possibile modificare l'intero componente composto, alcuni suoi sotto-componenti, o addirittura ogni singolo componente "leaf": ad esempio, se il Character grafico si muove lungo la sola asse X, invece di aggiornare l'intero componente, conviene aggiornare il singolo componente "X" appartenente al suo sotto-componente "Position".
Allo stesso modo, se il Character grafico si muove senza ruotare, conviene aggiornare solo "Position" piuttosto che l'intero componente relativo al Character.

2.7 Esempi pratici

Al fine di completare la dissertazione sull'architettura software implementata, di seguito verranno proposti una serie di casi che affrontano i problemi più comuni che ne derivano, e la relativa la soluzione adottata prendendo in esame ogni singolo Layer.

2.7.1 Movimento di oggetti

Il movimento di elementi grafici in funzione di un input utente è uno degli aspetti che contraddistingue le applicazioni multimediali interattive e, nella fattispecie, i videogiochi.
Per questo motivo è nata l'esigenza di estendere il componente "*Object3D*", che di fatto è sufficiente per modellare le informazioni che caratterizzano oggetti 3D "statici", in un apposito componente "*MovingObject3D*" che consentisse la descrizione di un oggetto in movimento.

Questo componente, come precedentemente illustrato nella sezione 2.5, è composto dagli stessi sotto-componenti di "*Object3D*", ma introduce due nuovi sotto-componenti: un componente di tipo Vector3 con LocalID "*NextPosition*" ed un componente di tipo String con LocalID "*NextTimeStamp*". Il primo indica la posizione che assumerà l'oggetto all'istante di tempo indicato dal secondo.
E' importante sottolineare come, nel Layer View e nel Layer Logic (e Data Layer), le coordinate della posizione di un oggetto in movimento vengano campionate utilizzando intervalli di tempo differenti: nel Layer View si deve garantire l'illusione di movimento "*continuo*", e questo avviene aggiornando ad ogni iterazione le coordinate secondo un vettore di "*offset*" calcolato in funzione della velocità e del tempo trascorso tra il "*frame*" attuale e quello precedente. In quei casi in cui la velocità di movimento è talmente alta da non consentire l'illusione di movimento, spesso si interviene con effetti "*Post Processing*" quali l'"*Onion Skinning*" ed il "*Motion Blur*".

Nel Layer Logic invece la frequenza di aggiornamento delle coordinate può essere notevolmente più bassa, poiché le informazioni presenti nel Data Layer devono essere le minime indispensabili a garantire la corretta interpretazione dello stato di gioco su ogni Peer, astraendo, da una parte, da quelle informazioni grafiche e decorative che non apportano cambiamenti sostanziali allo stato di gioco, cercando, laddove possibile, di mantenere una bassa

frequenza di aggiornamento (cioè intervalli di campionamento relativamente lunghi) dal momento che questi devono essere ogni volta propagati su tutta la rete Peer-To-Peer, e dall'altra, da quelle informazioni che in qualche modo possono essere ricavate mediante inferenza sulle informazioni presenti nel proprio Data Layer.
Qui entrano in gioco sistemi di "*prediction*" e di interpolazione della posizione: il primo viene comunemente realizzato mediante "*Dead Reckoning*", cioè un algoritmo grazie al quale, utilizzando informazioni quali velocità e accelerazione, è possibile stimare la posizione che assumerà l'oggetto preso in esame al prossimo istante di tempo, conoscendo la posizione attuale o, ancora, stimare la posizione attuale conoscendo quella relativa all'istante di tempo precedente. Inversamente, è possibile ricavare la velocità di spostamento (e quindi ogni singolo step da applicare lato View) se si è a conoscenza della posizione attuale e della posizione prevista ad un preciso prossimo istante di tempo.

Se si hanno a disposizione questi dati è possibile applicarvi un'interpolazione in modo da ricavare gli *step* intermedi tra la posizione all'istante di tempo "*Tn*" e la posizione all'istante di tempo "*Tn+1*". Questi *step* saranno più o meno ampi a seconda della velocità di spostamento e dei millisecondi trascorsi tra l'iterazione attuale e quella precedente.
Le interpolazioni che vengono comunemente applicate sono l'*interpolazione lineare* e l'*interpolazione spline*. Per la prima sono sufficienti le informazioni che abbiamo deciso di includere nel componente *MovingObject3D*, anche se il risultato non sempre sarà esteticamente apprezzabile , dal momento che all'interno dello stesso intervallo il movimento risulterà fluido, ma tra un intervallo e l'altro possono palesarsi repentini cambi di direzione.
Il secondo invece riduce questo problema, ma necessita di maggiori informazioni (più punti interpolanti) e, in particolari tipi di movimento, può presentare forti oscillazioni (fenomeno di Gibbs), discostandosi quindi da quello che era il movimento originale. Nel progetto di tesi si è scelto di utilizzare

l'interpolazione lineare.
Per ridurre ulteriormente artefatti di questo tipo, tale processo va esteso anche alla rotazione, utilizzando algoritmi interpolanti studiati appositamente per le rotazioni, come l'*interpolazione sferica*.

Grazie al "*Dead Reckoning*", quindi, si riduce la richiesta di banda per la notifica dei movimenti, ma contemporaneamente, dal momento l'interpolazione viene effettuata dal Layer View di ogni singolo Peer in modo indipendente, non viene garantita la consistenza dello stato di gioco durante l'intera azione di movimento, salvo la sincronizzazione apportata implicitamente dalla notifica dei singoli intervalli di tempo.
Sorgono però i seguenti problemi:
Quando un Peer "A" effettua il movimento di un oggetto di sua appartenenza, direttamente a carico del Layer View, ed invia, a precisi intervalli di tempo, l'aggiornamento del relativo componente logico al Peer "B" a lui connesso, subentra un problema causato dall'eventuale "*Lag*" tra "A" e "B".
L'aggiornamento dello stato del componente logico relativo all'oggetto in movimento, arriverà a "B" dopo un certo numero di millisecondi "N": significa che "B" inizierà il processo di movimento, e quindi di interpolazione e "*Dead Reckoning*", quando l'oggetto in "A" si è già spostato di un vettore pari alla velocità al millisecondo, moltiplicata per "N". Essendo comune per entrambi il punto d'arrivo al medesimo istante di tempo, l'oggetto in "B" sarà costretto a muoversi più velocemente rispetto ad "A" per recuperare il ritardo accumulato con il Lag (Figura 2.17).

Un altro problema frequente è caratterizzato da un fastidioso effetto visivo che vede l'improvviso riposizionamento dell'oggetto tra un istante di aggiornamento e quello successivo, quando la posizione di destinazione prevista differisce da quella reale. Errori di arrotondamento, fattori non previsti (collisioni, attrito, etc) e desincronizzazioni temporali portano a questo tipo di "*Jumping*" o di "*Warping*" (Figura 2.18 e Figura 2.19).

Quando la differenza tra il punto di destinazione previsto e quello reale

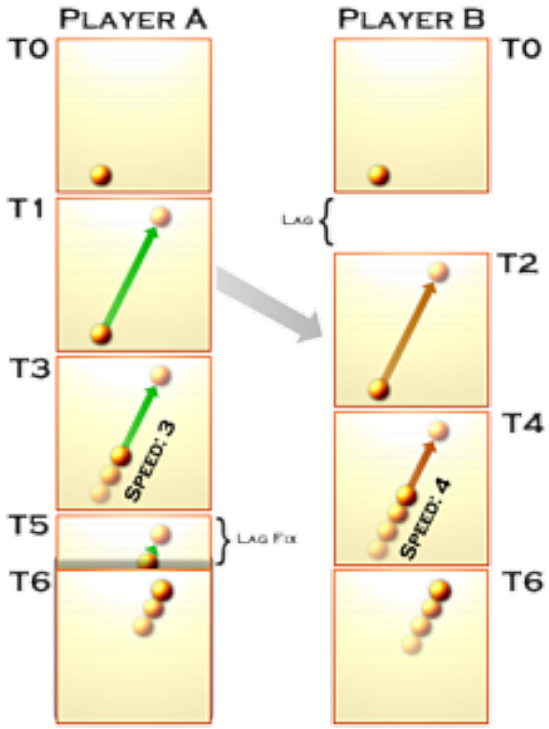

Figura 2.17: A T0 entrambi i Peer detengono uno stato sincronizzato, e l'oggetto (sfera arancione) si trova alla medesima posizione.
A T1 nel Peer A l'oggetto iniza a muoversi, ed invia la relativa notifica al Peer B.
A T2 arriva la notifica al Peer B, che inizierà il proprio processo di movimento.
A T3 l'oggetto in A si trova più avanti rispetto alla controparte in B.
A T4 l'oggetto in B deve muoversi più velocemente rispetto alla controparte in A, per compensare il ritardo.
A T5 l'oggetto in A ha quasi raggiunto il punto di destinazione.
A T6, che coincide con l'istante di tempo previsto nella notifica di movimento, entrambi gli oggetti sono giunti a destinazione.

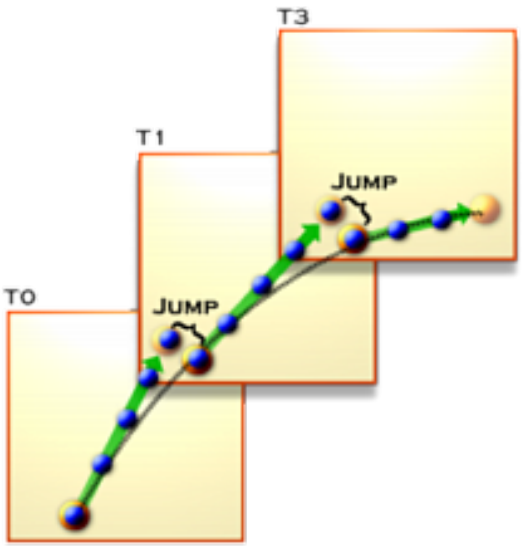

Figura 2.18: Ad ogni "istantanea" dello stato dell'oggetto si presenta una differenza di posizione (Jump) tra quella prevista come punto di destinazione nella istantanea precedente e quella riportata come punto di partenza nell'istantanea successiva. Le frecce verdi indicano la direzione, le sfere arancioni solide indicano il punto di partenza mentre quelle semitrasparenti indicano il punto di destinazione. I punti blu rappresentano gli step effettuati dall'oggetto grafico vero e proprio.

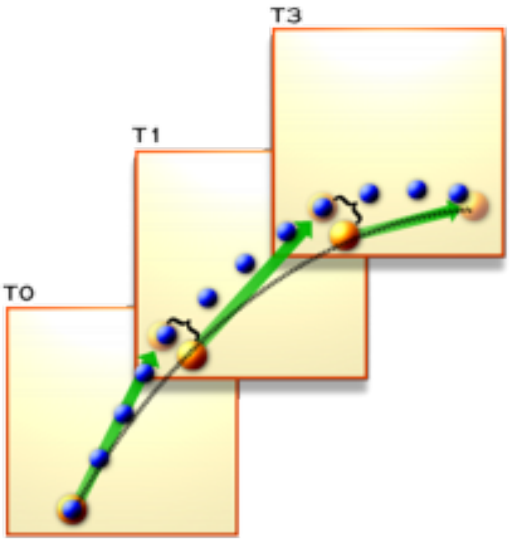

Figura 2.19: Una soluzione potrebbe essere quella di ignorare il punto di partenza dell'istantanea successiva quando la distanza tra questo ed il punto di destinazione dell'istantanea precedente è al di sotto di una certa soglia. Dal momento che l'oggetto è in movimento e lo sarà anche nella prossima istantanea, si può evitare di riposizionarlo sul punto di partenza, che causerebbe antiestetici artefatti di Warping, lasciando continuare la sua corsa verso il prossimo punto di destinazione, calcolata rispetto la sua attuale posizione.

è troppo elevata, risulta difficile compensare l'offset di spostamento, e contemporaneamente garantire un movimento naturale e continuo.
Una soluzione è quella di intervenire sulla frequenza di sincronizzazione, eseguendo aggiornamenti sul DataLayer (quindi propagati automaticamente su tutti i Peer connessi) "eccezionali", fuori cioè dalla logica di aggiornamento periodico ad intervalli di tempo prefissati.

Questo genere di aggiornamenti "forzati" viene attuato in quelle situazioni critiche nelle quali la probabilità che il punto di destinazione previsto nell'istantanea precedente sia diverso da quello che si va a prefigurare è molto elevata: cambi repentini di direzione, cambiamenti di velocità, previsioni di eventuali collisioni imminenti, o lo stop improvviso dell'oggetto in questione, ricadono nella casistica delle situazioni critiche più comuni (Figura 2.20 e Figura 2.21).

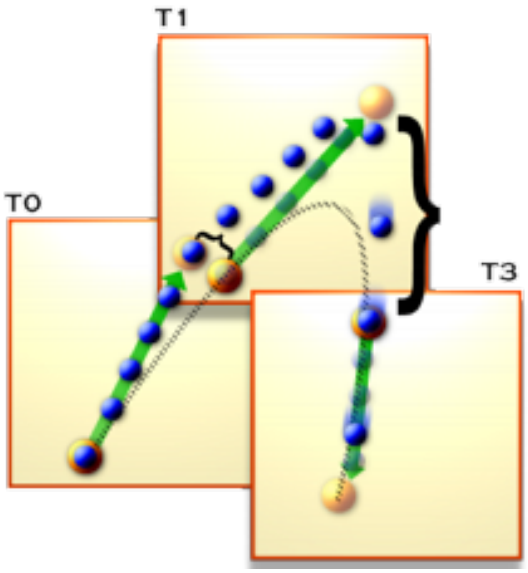

Figura 2.20: Anche se si evita di riposizionare l'oggetto in movimento nella posizione di partenza proposta dall'istantanea di turno, quando la differenza è troppo elevata tra questa e quella di destinazione prevista nell'istantanea precedente, risulta difficile mantenere il movimento costante e continuo.

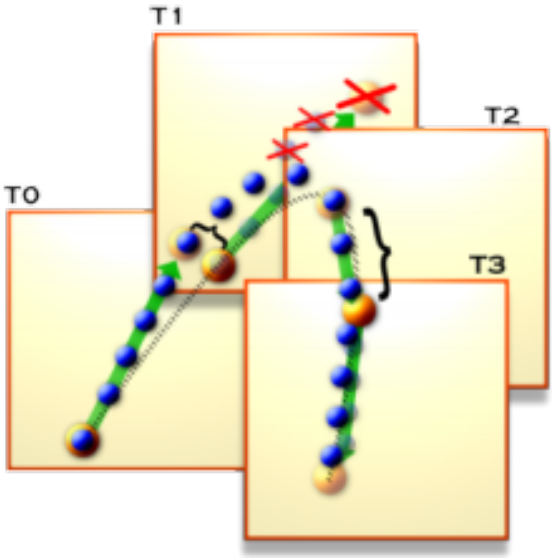

Figura 2.21: Una soluzione può essere quella di broadcastare una o più istantanee ausiliarie in quelle situazioni critiche nelle quali è molto probabile che la destinazione prevista non sia più valida.

Questo tipo di soluzioni si basano però sull'assunzione che il l'oggetto in questione sia in movimento e rimanga tale: quando l'oggetto termina il proprio movimento, passando dallo stato "*Moving*" a quello di "*Stopped*", il palesarsi di artefatti di minore entità è inevitabile, dal momento che la posizione riportata in un'istantanea di un oggetto ormai fermo non può essere ignorata, poiché rappresenta l'attuale stato dell'oggetto e questo può anche non subire più modifiche fino a fine partita: se non si riposizionasse l'oggetto alle coordinate specificate come punto di partenza, questo rimarrebbe fermo in una posizione che, visti i problemi sopra elencati, probabilmente differirebbe da quanto riportato nel DataLayer del Peer "Owner".

Riassumendo, dato un oggetto 3D non statico, questo viene mosso nel contesto View dal suo Peer "Owner"; il relativo "*ComponentScript*" aggiornerà periodicamente il componente logico del Data Layer a cui è associato. Di conseguenza i Peer "non Owner" appartenenti alla Membership riceveran-

no un aggiornamento periodico che modificherà automaticamente lo stato dei componenti sui loro Data Layer. Questo si ripercuoterà sui rispettivi Layer View, che, attraverso i "*ComponentScript*" associati ai rispettivi oggetti grafici, ne modificheranno le coordinate secondo una politica di *Dead Reckoning* (Figura 2.22).

2.7.2 Collisione tra oggetti

La collisione tra elementi grafici è un altro aspetto caratteristico delle dinamiche presenti nei videogiochi. Non solo, è anche uno degli aspetti più sensibili e influenti sul flusso di gioco, poiché questi possono rappresentare una discriminante tra la vittoria e la sconfitta di una partita.
Ogni collisione è tipicamente composta da due fasi:"*Detection*" e "*Response*".
La prima fase si occupa di testare la presenza di eventuali collisioni tra i diversi oggetti che compongono la scena 3D, mentre la seconda si occupa di calcolare la risposta, in termini di movimento e rotazione, che tali oggetti devono compiere a seguito di una collisione.
La "*Collision Detection*" solitamente opera su un'approssimazione di quelli che sono i volumi occupati dalla geometria degli oggetti 3D, e solo raramente direttamente sulla geometria poligonale dei singoli modelli.
Tra le approssimazioni più comuni vi sono le collisioni "*Sphere-Sphere*","*Sphere-Box*" e "*Box-Box*".
La "*Sphere-Sphere*" è tendenzialmente la meno precisa ma anche la più semplice da realizzare e computazionalmente meno costosa. È implementata mediante un test sulla distanza tridimensionale tra i due oggetti (Figura 2.23).
"*Sphere-Box*", invece, prevede un test di intersezione tra una sfera ed un "*BoundingBox*", cioè un'approssimazione del volume del modello tridimensionale in un parallelepipedo, calcolato in base ai vertici estremi del modello stesso per ogni asse. Se il test viene eseguito su un "*Axis-Aligned Bounding Box*" (AABB, ovvero un BoundingBox non orientato) continua ad essere poco costoso e di semplice implementazione (Figura 2.24).

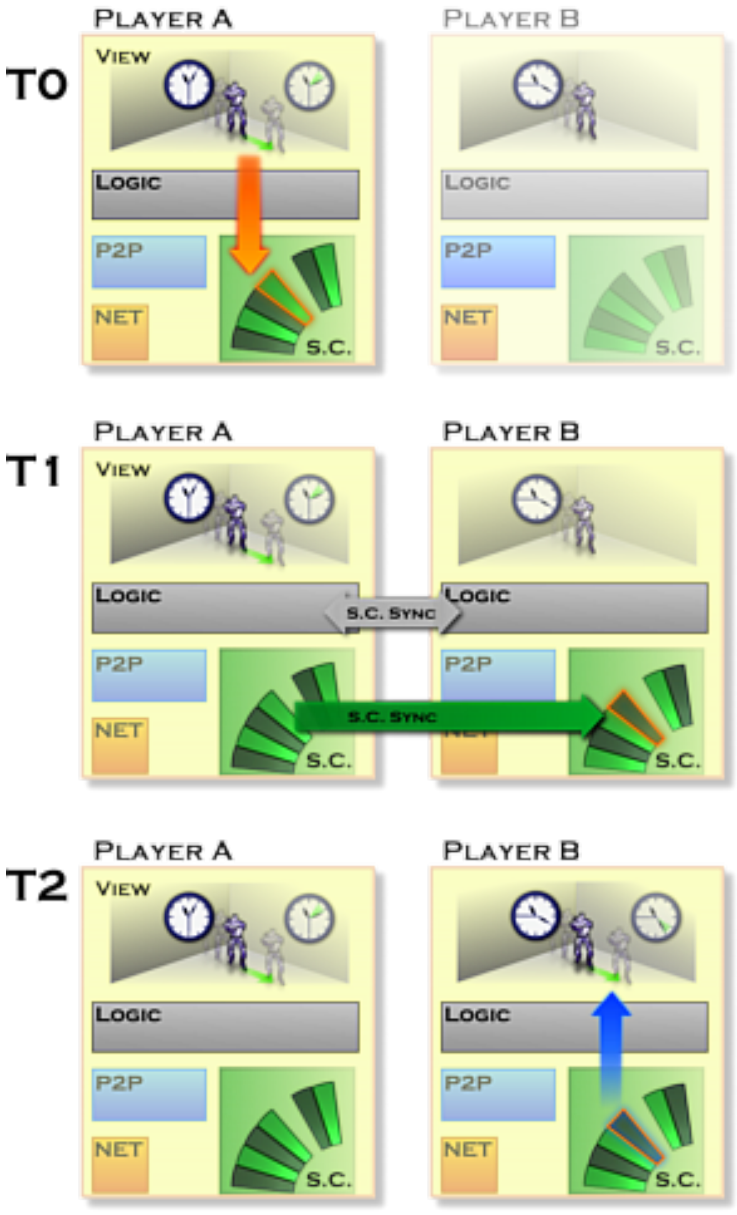

Figura 2.22: T0: Il Player *A*, l'"Owner" sposta l'oggetto direttamente a livello View, e tramite il relativo ComponentScript stima la prossima posizione. Di conseguenza aggiorna il rispettivo componente di tipo "*MovingObject3D*"
T1: Automaticamente viene propagato l'aggiornamento al Player *B*.
T2: Il ComponentScript in B applica il Dead Reckoning sul valore riportato nel componente appena aggiornato.

La "*Box-Box*" invece testa l'intersezione tra due "*BoundingBox*" (Figura 2.25).

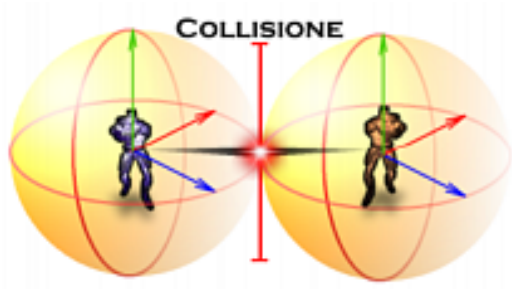

Figura 2.23: Sphere-Sphere

Figura 2.24: Sphere-Box

Esistono inoltre altri tipi di approssimazione che introducono nuovi volumi geometrici come il "*Cylinder*" (anch'esso poco costoso), o una composizione di tali (Con due "*Sphere*" ed un "*Cylinder*" sia ha una "*Capsule*", particolarmente indicata per la rappresentazione di oggetti antropomorfi).

La "*Collision Response*" viene eseguita utilizzando i dati ricavati nel processo di "*Collision Detection*": oltre ad informazioni geometriche di base, come le coordinate del punto di intersezione, la surface od il poligono colpito ed il suo vettore normale, possono essere utilizzate informazioni aggiuntive

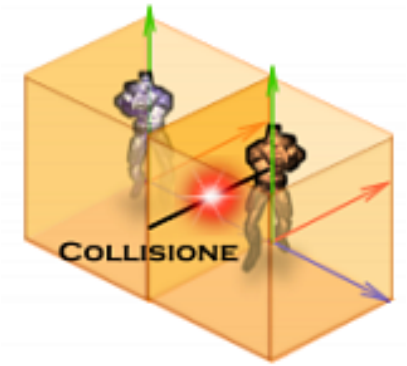

Figura 2.25: Box-Box

utilizzate dall'Engine fisico, come il tipo di materiale colpito, la "*Bounciness*", la sua "*Densità*' e la massa degli oggetti coinvolti nella collisione.

Ci sono poi una serie di ottimizzazioni che operano sui dati presenti nello "*SceneGraph*", atte ad evitare che vengano eseguiti test di "*Collision Detection*" inutilmente e tra tutti gli oggetti, anche quando la distanza tra questi è notevole: spesso infatti viene eseguito un test molto approssimato mediante un "*Sphere-Sphere*" massimizzato, che "*filtra*" e raggruppa quegli oggetti che si trovano in prossimità reciproca, applicando quindi "*Collision Detections*" più precise solo a quest'ultimi.
Nel caso di grandi spazi, inoltre, si tende a razionalizzare preventivamente la scena di gioco in strutture apposite come il "*QuadTree*", che partizionano lo spazio in quadranti ricorsivi.
Dal punto di vista implementativo, entrambe le fasi verranno gestite dall'"*Engine fisico*" presente nel Layer View. Ciò significa che, così come avviene per il movimento, chi controlla ed eventualmente modifica la posizione dei singoli oggetti 3D risidede nel Layer View dei loro rispettivi Peer "*Owner*".

Siano due oggetti tridimensionali, "*Oggetto A*" ed "*Oggetto B*" in collisione, appartenenti rispettivamente al Peer "*A*" e al Peer "*B*", la "*Collision Detection*" verrà eseguita allo stesso modo in entrambi i Peer, ma il "*Colli-*

sion Response" su "*Oggetto A*" verrà gestito esclusivamente dal Peer "*A*", e, viceversa il "*Collision Response*" su "*Oggetto B*" verrà gestito esclusivamente dal Peer "*B*". Gli effetti delle "*Collision Responses*" sui componenti logici coinvolti, verranno poi immediatamente inoltrati agli altri Peer, che sposteranno infine i rispettivi oggetti View di conseguenza.

Esistono però contesti nei quali la "*Collision Response*" non individua una risposta "*fisica*", cioè trasformazioni di rototraslazioneil il cui aggiornamento è a carico di *MovingObjectScript*, bensì individua variazioni logiche che impattano sulle dinamiche di gioco e sul valore di specifici componenti del Data Layer. Si pensi ad esempio alla collisione tra un *Bullet* ed un *Character*, o ancora tra un *Character* ed un "*HealthPack*". Nel primo caso il *Bullet* non deve subire una risposta fisica, rimbalzare o deviare traiettoria, ma distruggersi e decrementare i *Punti vita* del Character colpito; viceversa, nel secondo caso invece l'"*HealthPack*" deve distruggersi e incrementare i *Punti vita* del Character colpito.

In entrambi i casi viene utilizzato il sistema di *Messaging* per la notifica della collisione: Poiché il componente che rappresenta i *Punti vita* di un *Character* può essere modificato esclusivamente dal proprio Player "Owner", mentre il *Bullet* (o l'*HealthPack*) tipicamente appartengono ad un Player differente, e dal momento che si è scelto di gestire il "*Collision Detection*" tra oggetti di tipo *Bullet* e *Characater* solo all'interno *ComponentScript* associato agli oggetti del primo tipo, quest'ultimo deve "informare" dell'avvenuta collisione il *Characater* coinvolto, che, di conseguenza decrementerà il proprio sotto-componente dedicato ai *Punti vita* (Figura 2.27).

Analogamente al movimento, anche per quanto riguarda le collisioni esistono dei problemi dovuti al Lag, arrotondamenti dei valori interpolanti e alla frequenza di aggiornamento dell'Engine fisico (Tick). Anzi, in qualche modo, quest'ultimi vengono proprio amplificati dall'instabilità degli algoritmi di interpolazione utilizzati nel processo di movimento, e alla loro, seppur minima, differenza tra Peer e Peer.

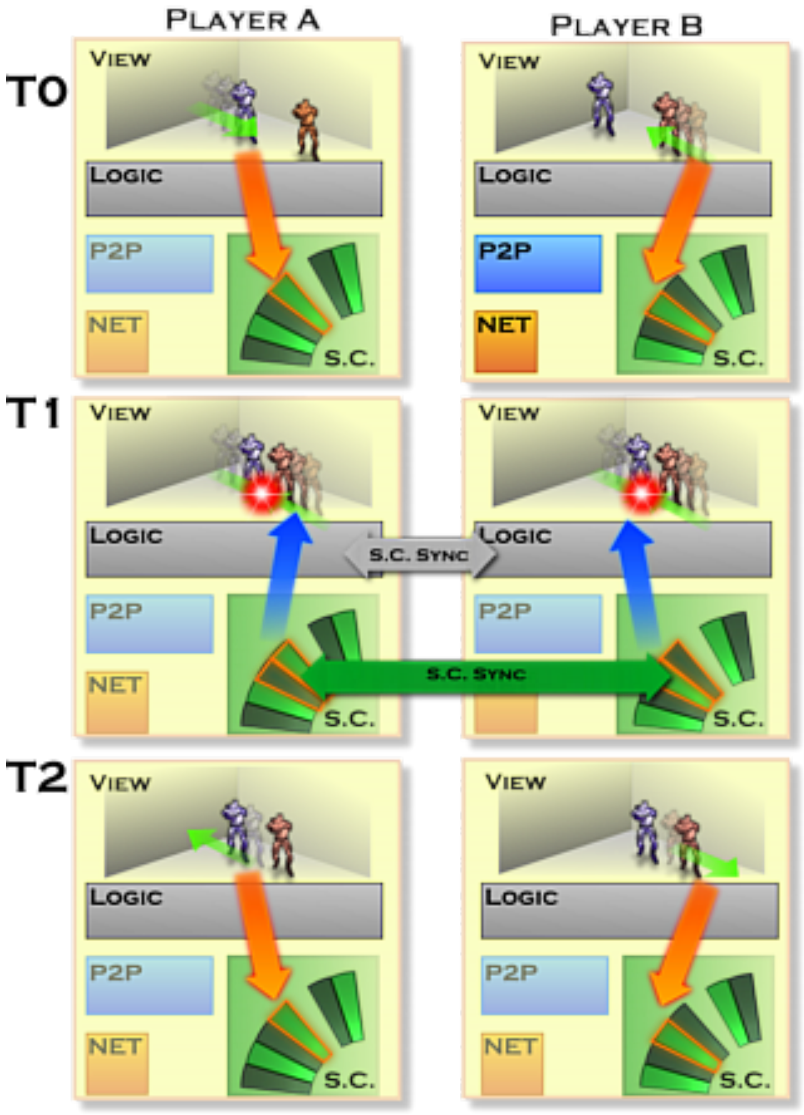

Figura 2.26: In T0 il Player A sposta il suo personaggio in direzione del personaggio appartenente al Player B, e, viceversa, il Player B muove il suo personaggio verso il personaggio del Player A.
In T1 Viene attutata la sincronizzazione del DataLayer, e di conseguenza ognuno dei due Player, eseguirà, nel proprio LayerView, mediante Dead Reckoning, il movimento del personaggio appartenente all'altro Peer. Entrambi gli Engine fisici rileveranno una collisione e, in T2, come risposta muoveranno i rispettivi personaggi nel verso opposto.

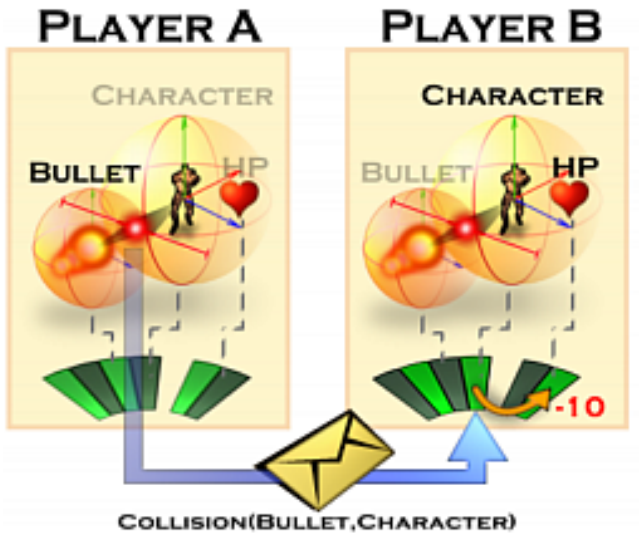

Figura 2.27: Il Layer View del Player A rileva una collisione tra il proprio Bullet ed il Character appartenente al Player B. Come "*Collision Response*" logico, viene inviato un messaggio al componente del DataLayer che rappresenta il Character del Player B. Una volta ricevuto, all'interno del suo metodo ExecMessage(), verrà decrementato il valore del sotto-componente HealthPoint di N punti, a seconda del tipo di Bullet.
Tale modifica verrà notificata automaticamente al Player A.
Notare come Il Player A avrebbe comunque potuto inviare il messaggio anche alla propria copia del componente relativo al Character di B: il risultato sarebbe stato immediato, ed il successivo aggiornamento proveniente da B ne avrebbe semplicemente sovrascritto il valore senza modificarne lo stato di gioco. (Entrambi infatti avrebbero riportato lo stesso valore di HP decrementato). Una differenza di valore, in questo senso, potrebbe essere un campanello d'allarme per un eventuale sistema Anti-Cheating.

Può capitare infatti che un Bullet di un Player non rilevi la collisione con un Character in movimento appartenente ad un altro Player perchè uno dei due (o entrambi) sono soggetti ad un'elevata latenza a tal punto che le notifiche dei rispettivi movimenti vengano ricevute con un sincronizzazione temporale tale da bypassare la *Collision Detection* tra il Bullet ed il Character. O viceversa, che venga rilevata una collisione con un Character che in realtà si trova già fuori portata di tiro.

Facendo riferimento alla Figura 2.28, si possono individuare alcuni problemi comuni: Il Player A ha un Lag relativamente basso (*60ms*), contrariamente a B e C che hanno un Lag relativamente alto (rispettivamente *120ms* e *240ms*).
In T1 il Player A spara un Bullet. In questo istante il proiettile è presente solamente nel Data Layer e Layer View del Peer A.
In T2 il Player B riceve la notifica di creazione Bullet, e conseguentemente crea la propria copia del componente logico, e quindi il relativo oggetto grafico nel proprio Layer View. Inoltre inizia a muovere il proprio Character nella direzione del Bullet di A. Nel frattempo in A il proiettile si è mosso di qualche step. Il Player C, invece, è ancora fermo allo stato di gioco di T0.
In T3 il Bullet di A ed il Character di B continuano la propria corsa, mentre in C è finalmente giunta la notifica di creazione Bullet.
In T4, nel Layer View del Player B, si palesa una collisione tra il Bullet di A ed il Character di B. Cosa che invece non avviene nel Layer View del Player A, poiché il movimento di B è in ritardo di qualche step. Dal momento che, per una scelta implementativa, è il *ComponentScript* dell'Owner del *Bullet* ad effettuare la *Collision Detection*, cioè A, tale collisione viene ignorata e non ha ripercussioni quindi sullo stato di gioco.
Il Player C inizia a muovere il proprio Character, ma tale informazione per il momento rimane confinata al suo Peer.
In T5 è C a rilevare una collisione tra il Bullet di A ed il Character di B, ma per gli stessi motivi elencati nel precedente punto, ciò non ha ripercussioni

sullo stato di gioco. Intanto C continua ad allontanarsi dalla rotta di collisione del Bullet, ma tale movimento continua ad essere notificato in ritardo agli altri Player.
In T6, A rileva finalmente una collisione, tra il proprio Bullet ed il Character di C. In realtà il Character di C si è già allontanato da tempo dalla rotta del Bullet, ma A, basandosi sul proprio DataLayer, crede ancora che il Player C si trovi alla posizione di partenza. Il Player A quindi invia il messaggio di collisione al componente relativo al Character di C.
In T8, il Character di C riceve il messaggio di A e si decrementa i propri *Punti vita*.
In T9 ed in T10, la propagazione di tale modifica raggiunge infine sia il Player A che il Player$B$

L'esempio, volutamente amplificato, mostra come, in caso di prestazioni di rete non ottimali, la desincronizzazione "percepita" possa deteriorare notevolmente l'esperienza di gioco (il Player C si vede sottrarre dei *Punti vita*, nonostante, dal suo punto di vista, abbia schivato abbontantemente il proiettile di A). In un contesto più realistico, il rapporto tra gli spazi di movimento, la frequenza di aggiornamento ed il Lag è minore, rendendo l'impatto di questo genere di desincronizzazioni piuttosto ridimensionato.
È importante notare però come, nell'ultimo istante temporale presentato nell'esempio, lo stato di gioco sia perfettamente sincronizzato in tutti i tre Peer.

In un First Person Shooter solitamente si possono distinguere due tipi di Bullet: quello "*fisico*", tipicamente più lento e quindi schivabile, e quello detto "*HitScan*", che non implica la creazione di un apposito oggetto 3D e relativa "*Collision Detection*": viene infatti testata direttamente l'intersezione tra il vettore "*Ray*", le cui coordinate coincidono con le coordinate del mouse (con la dovuta conversione da ScreenSpace (2D) a WorldSpace (3D) ed il cui verso coincide con la profondità individuata dalla rotazione della telecamera, e gli oggetti di tipo Character (od una approssimazione del loro volume geometrico).

Figura 2.28: Problematiche relative alla collisione tra oggetti in un contesto caratterizzato da una un'elevata latenza

Quando si verifica un'intersezione tra un "*Ray*" ed un Character, e quest'ultimo si trova in testa alla lista degli oggetti intersecati (un "*Ray*", può intersecare, infatti, più oggetti contemporaneamente, ordinati in una lista dal più vicino al più lontano), il Peer che ha eseguito l'"*HitScan*", invierà un messaggio al componente relativo al Character colpito, che di conseguenza decrementerà i propri "*Punti vita*".

2.7.3 Creazione e distruzione di oggetti

Come illustrato nelle sezioni 2.4 e 2.6, la creazione di un elemento grafico che apporta informazione allo stato di gioco, come può essere un oggetto 3D, viene concretizzata, istanziando il componente visuale automaticamente dal *View Context* del Layer View, basandosi sui componenti presenti nel Data Layer.
Per quegli oggetti grafici che devono essere "condivisi", cioè presenti in tutti i Peer appartenenti alla Membership, è imperativo, quindi, passare per il Data Layer, istanziando il componente logico, piuttosto che creare direttamente l'oggetto 3D.
Per esempio, se si vuole creare un Bullet tridimensionale, sarà sufficiente creare un componente di tipo Bullet opportunamente inizializzato (coordinate e rotazione devono coincidere con quella del Character che "*spara*" il Bullet).
Analogamente, per distruggere un oggetto 3D, invece di eliminare direttamente l'istanza grafica, è necessario rimuovere il relativo componente dal DataLayer.
Eliminando solamente l'istanza grafica, infatti, ne verrebbe istanziata una nuova copia, automaticamente all'iterazione successiva, dal momento che il componente logico risulta ancora presente nel Data Layer.

Ogni creazione e distruzione di un componente, come illustrato nella sezione 2.4, viene propagata all'intera Membership attraverso i comandi "*Create-*

Component" e "*DestroyComponent*". Se il tipo di componente è un aggregato di sotto-componenti, allora saranno inoltrati tanti comandi "*CreateComponent*" e "*DestroyComponent*" quanti sono i sotto-componenti.
Gli elementi presenti nell'ambiente tridimensionale condiviso, a seconda delle loro caratteristiche e della loro funzione, si possono raggruppare nei seguenti insiemi:

- Elementi di scena: sono quegli elementi "statici" che non apportano informazione a livello di stato di gioco. Tipicamente *Arene (o mappe)*[14], stanze, corridoi, o altri oggetti prettamente decorativi rientrano in questa tipologia di elementi, che non vengono istanziati conseguentemente ad un evento contestualizzato a livello logico o di rete, ma son presenti direttamente nel Layer View, indipendentemente dallo stato del Peer sottostante.

- Elementi attivi: sono quegli elementi "dinamici" che, con la loro presenza alterano lo stato di gioco: characters, Bullets, HealthPacks, ad esempio, rientrano in questa tipologia.

Tra gli elementi attivi si può operare un'ulteriore distinzione. Alcuni di questi elementi, infatti, "appartengono" in modo naturale ad un Peer: il Character del Player *A* è chiaramente di proprietà del rispettivo Peer. Analogamente un Bullet sparato da un Character del Player *B* apparterrà al Peer *B*. Altri elementi però sfuggono a questa regola. Si pensi agli *HealthPack*: a quale Peer appartengono? O, in altre parole, a quale Factory è delegata la loro creazione?
Per questo tipo di elementi si è pensato di agire nel seguente modo: sulla mappa "statica" (appartenente quindi alla categoria degli elementi di scena) vengono inseriti N *PlaceHolder*, cioè punti astratti che al momento della realizzazione della mappa non contengono nessun oggetto in particoare, ma identificano comunque una posizione eventualmente occupabile in runtime.

[14]Gli ambienti nei quali si svolgono le partite di un FPS

Questi *PlaceHolder* sono anch'essi elementi di scena dal momento che caratterizzano la mappa di gioco (come colonne, punti luce, ostacoli, etc), e sono quindi caricati contestualmente al loading della mappa (che viene eseguita in modo del tutto indipendente da un discorso di rete).

Quando inizia una partita, ogni Peer "occupa", preventivamente, N/(Numero di Peer) *PlaceHolder* con un componente di tipo *HealthPackGenerator* (ereditato da *Object3D*). Il criterio (necessariamente condiviso) utilizzato per la distribuzione e l'assegnazione dei *PlaceHolder* ai diversi Peer può essere realizzato mediante politiche basate sul NetID, o qualunque altra informazione che contraddistingue ogni singolo Peer, evitando quindi funzioni aleatorie.
Ogni *HealthPackGenerator* si occuperà di istanziare periodicamente, laddove non già presente, un componente di tipo *HealthPack* alle stesse coordinate del relativo *PlaceHolder*.

Questo significa che ogni Player "possiede" un determinato numero di *HealthPack* e ne gestisce quindi anche le collisioni con i Character degli altri Player (Figura 2.29).

2.7.4 Connessione e Disconnessione dei Player

Nelle sezioni precedenti è stato evidenziato come ogni informazione atomica appartenga ad un determinato Peer.
Questo risolve, da una parte, il problema della consistenza dello stato di gioco, ma dall'altra introduce alcune problematiche dovute all'importanza di ogni singolo Peer, al punto da poter essere considerati degli *SPOF*.

Quando un Peer si disconnette od esce dalla Membership, gli effetti della logica di aggiornamento dei componenti di sua appartenenza non viene più propagata agli altri Peer. Per alcuni tipi di componenti, inoltre, può essere necessario distruggerne le istanze quando il Peer owner non è più raggiungibile.
Non esiste una regola generale, ma è possibile definirne una per ogni tipo

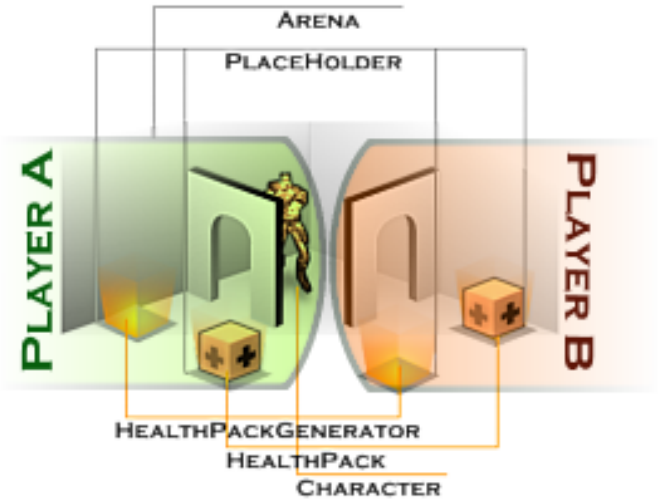

Figura 2.29: Arena e PlaceHolder sono elementi di scena, mentre HealthPack-Generator, HealthPack e Character sono elementi attivi. Alcuni di questi appartengono al Player A ed altri al Player B.

di componente: gli HealthPack e i Bullet, ad esempio, non possono essere distrutti, e devono continuare ad essere aggiornati. Il Character, invece, generalmente viene distrutto.

Sorge quindi un problema legato all'aggiornamento di quei componenti il cui Peer non è più presente all'interno della Membership. Una soluzione potrebbe essere quella di forzare il cambio di proprietà dal Peer disconnesso ad un Peer ancora connesso.
Il Modulo SharedComponents, in casi eccezionali, permette di delegare la proprietà di un componente come se fosse stato generato da un'altro Factory. Il Peer che ne acquisisce la proprietà diventa il suo nuovo "*Owner*" e deve notificare tale passaggio a tutti gli altri Peer ancora connessi.
Per decidere a quale Peer cedere la proprietà, analagomente a quanto succede per la creazione degli HealthPack, l'utilizzo di una funzione aleatoria è sconsigliabile. Così come è sconsigliabile eseguire un Polling N-a-N: ogni Peer esegue un Polling di check sul Peer temporalmente connesso dopo di

lui, e nel caso quest'ultimo non sia più raggiungibile si appropria dei suoi componenti. Se l'ordine di connessione è: Player A, Player B, Player C e Player D, quindi, Player A esegue un Polling sul Player B, il Player B sul Player C, il Player C sul Player D ed infine il Player D sul Player A. Ogni Peer comunque, a livello di implementazione del modulo NET, riceve una notifica alla disconnessione di nodo: se sia il Player B che il Player C si disconnettono contemporaneamente, prima che Player B sia riuscito quindi a notificare agli altri Peer l'avvenuta acquisizione dei componenti di Player C, Player A, sapendo che sia B che C non sono più presenti nella Membership, acquisisce i componenti appartenenti al Player B e quindi quelli appartenenti al Player C.

Capitolo 3

Implementazione

Di seguito, dopo un'introduzione generale, verranno illustrate le problematiche e le soluzioni implementative che sono state individuate e concretizzate all'interno di ogni singolo Layer.

3.1 Introduzione Generale

L'implementazione software rispecchia la suddivisione dell'architettura presentata nel capitolo 2 (Figura 3.1): Tutti i Layer ed i Moduli, ad esclusione del Layer View, sono disponibili come *DLL*[1] e realizzati in modo da minimizzare le dipendenze.
Quest'ultime sono, infatti, riscontrabili esclusivamente nei legami tra un Layer superiore e quello inferiore, e non viceversa.
Per la realizzazione di tali legami, inoltre, si é scelto da una parte di esporre *Interfacce* piuttosto che *Classi* specifiche, e dall'altra di seguire, laddove possibile ed effettivamente conveniente, l'*Observer Pattern*, poiché induce un maggiore disaccoppiamento tra i diversi Layer.
Si é inoltre cercato di evitare l'utilizzo di metodi e membri statici, rendendo

[1]Dynamic-Link Library, libreria software che viene caricata dinamicamente in fase di esecuzione

possibile l'esecuzione di più Peer all'interno dello stesso programma. Ciò ha facilitato i processo di debugging dei Layer logici e di rete.
I diversi Layer condividono alcuni elementi di natura strutturale: ad esempio il Layer Net, il Layer P2P ed modulo SharedComponents concretizzano il concetto di "*comando*" nel medesimo modo (utilizzando lo stesso tipo di interfacce e presentando implementazioni simili).

Per quanto concerne le tecnologie ed i linguaggi di programmazione utilizzati, uno dei fattori che ne hanno vincolato la scelta, risiede nel Layer View. Il Layer View infatti é composto da moduli piuttosto complessi, le cui funzioni esulano da quelle che sono le tematiche del progetto di tesi.

Era necessario individuare un Framework od un Engine che offrisse il maggior numero di funzionalità richieste dalle specifiche del Layer View e, contemporaneamente, permettesse in breve tempo l'integrazione con i Layer sottostanti.

La scelta é ricaduta su *Unity3D*[2].
Unity3D si presenta come un ambiente integrato che mette a disposizione un *Renderer 3D*, uno *SceneGraph* particolarmente modulare, un *Engine fisico* per la gestione dei movimenti e delle collisioni nonché sistemi per la gestione degli input ed il supporto audio.

Tra le caratteristiche peculiari di *Unity3D* vi sono il costo (gratuito per applicazioni non commerciali), l'Editor WYSIWYG, che consente una rapida definizione della scena tridimensionale, e la possibilità di convertire il progetto su diverse piattaforme.

Per quanto riguarda la programmazione, Unity3D supporta l'utilizzo del linguaggio C#, e mette a disposizione un sistema semplificato di "scripting"

[2]www.unity3d.com

basato sul concetto di *Behaviour* (che definisce parte della logica di ogni singolo oggetto grafico), consentendo comunque l'utilizzo di paradigmi ed approcci più classici.

Per ridurre al minimo problematiche d'integrazione si é scelto, quindi, di procedere con l'utilizzo del linguaggiio C# anche per l'implementazione dei Layer sottostanti.

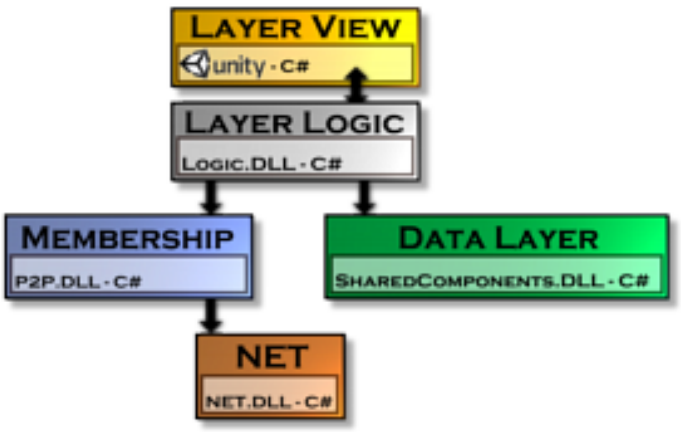

Figura 3.1: Il Layer NET (dedicato alla comunicazione) è implementato nel file *NET.DLL*;
Il Layer P2P (dedicato alla gestione della Membership) è implementato nel file *P2P.DLL*;
Il Modulo SharedComponents (utilizzato come Data Layer) è implementato nel file *SharedComponents.DLL*;
Il Layer Logic è implementato nel file *Logic.DLL*, mentre il Layer View fa da EntryPoint e coincide con l'eseguibile dell'applicazione.

3.2 Layer NET

Il Layer NET presenta, ad un primo livello, alcune classi di servizio, che per lo più espongono proprietà e metodi statici:

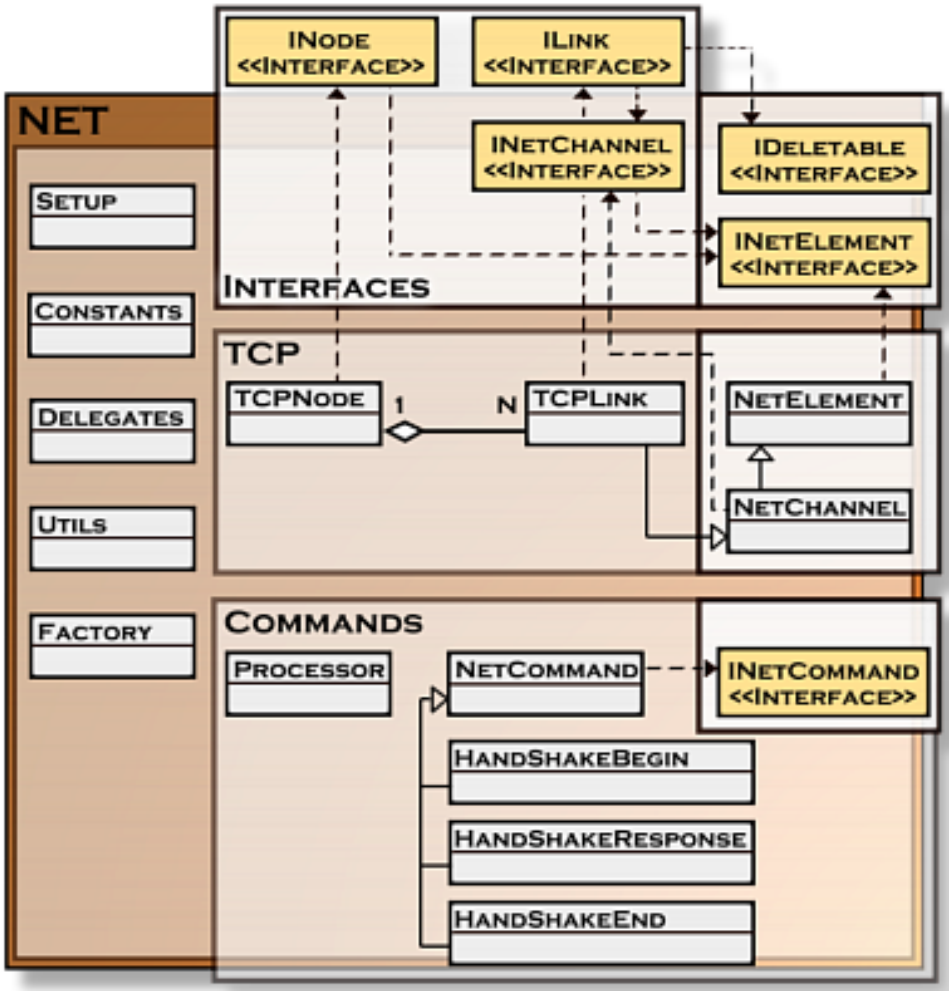

Figura 3.2: Diagramma delle Classi - Layer NET

- Le classi *Setup* e *Constants* fanno da contenitore a tutti i parametri di configurazione del Layer. Ad esempio sono qui definiti i nomi delle classi che implementeranno le interfacce *INet* e *ILink*, i millisecondi di attesa dei Thread di lettura e scrittura di un *Link*, e la dimensione della finestra TCP (nel caso sia utilizzato questo protocollo). Come prima implementazione, tali valori sono inizializzati direttamente all'interno del codice, ma come sviluppo futuro è previsto l'utilizzo di file di configurazione esterni.

- La classe *Delegates* raccoglie la definizione di tutti i *Delegates*, ovvero tipi di dato necessari a formalizzare la natura dei metodi anonimi utilizzati per l'implementazione degli Eventi e dell'Observer Pattern in generale, sui quali si basa la comunicazione da questo Layer verso i Layer superiori.

- La classe *Utils* raccoglie una serie di metodi ausiliari e di utilità generale. Ad esempio qui risiedono i metodi di conversione tra IP Address, Port e NetID.

- La classe *Factory* permette, attraverso il principio *IoC*[3] di istanziare oggetti concreti di INode e ILink astraendo dalla loro implementazione.

Sono poi definite le *interfacce* esposte ai Layer superiori:

- INode: Identifica un nodo di rete. Al suo interno è presente un lista di ILink, che vengono istanziati sulla chiamata del metodo *ConnectTo(string ipAddress, int port)*.
  Il metodo *SetNewIdentifierOfLink(ILink link, String newIdentifier)* consente di cambiare l'identificativo di un link.
  Sono inoltre esposti una serie di metodi che permettono ai Layer superiori di registrarsi ad alcuni eventi dispacciati dal singolo nodo: ad esempio quando un ILink presente nella lista riceve un messaggio, viene lanciato il metodo delegato che si era registrato all'evento OnRead.

[3]Inversion of control, un pattern grazie al quale è possibile delegare ad una terza entità la creazione di un preciso oggetto indicandone, ad esempio, solamente l'interfaccia

- ILink: identifica un canale di rete. Attraverso i metodi *Read()* e *Write()* è possibile ascoltare ed inviare messaggi all'endpoint associato all'ILink in questione.

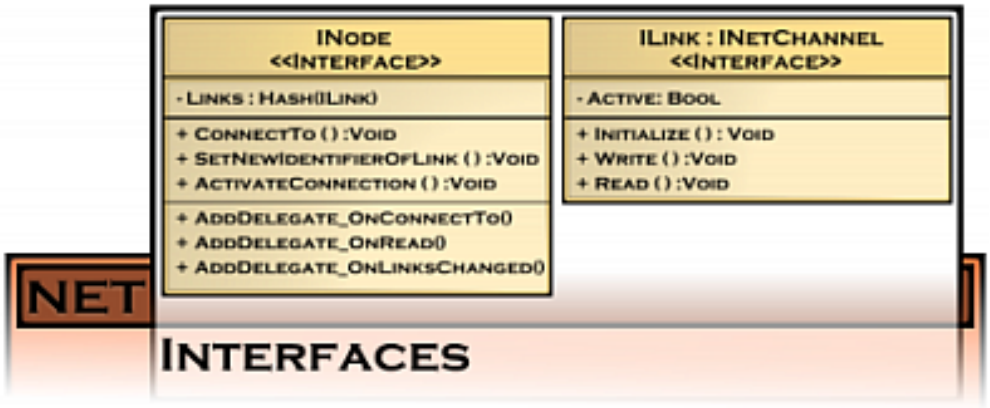

Figura 3.3: Interfacce NET

Tali interfacce sono state concretizzate attraverso due classi: TCPNode e TPCLink. Il protocollo di rete utilizzato come prima implementazione è infatti il TCP/IP. Applicazioni interattive come i videogiochi, spesso richiedono un frequente e rapido scambio di informazioni. Il protocollo UDP potrebbe essere consigliabile in questo senso, poiché è *Message-Oriented* invece che *Stream-Oriented*, e per ogni pacchetto inviato non deve aspettare un messaggio di ritorno; d'altro canto, però, per evitare problemi di consistenza, risultano necessari sistemi che garantiscono maggiore sicurezza e affidabilità in termini controllo di congestione, *Ack* di ricezione ed il mantenimento dell'ordine di invio dei pacchetti. Inoltre per l'individuazione di una Membership può risultare più comodo un protocollo *Connection-oriented* poiché la connessione e la disconnessione di nodi all'interno della rete vengono gestiti direttamente dal sistema, rendendo superflua un'emulazione logica lato applicazione: esistono infatti implementazioni UDP "*Reliable*", anche appositamente mirate ai videogiochi (si veda *Raknet*[4]), che, lato applicazione,

[4]http://www.jenkinssoftware.com

fanno fronte a questo tipo di problematiche. Dal momento che il Layer NET comunica con il Layer dedicato alla gestione della Membership attraverso interfacce agnostiche rispetto al protocollo utilizzato, si è scelto, in prima istanza, di appoggiarsi al protocollo TCP/IP. Questa scelta non risulta comunque vincolante, e sarà possibile, in futuro, supportare un'implementazione esclusivamente UDP, od eventualmente ibrida, senza dover modificare le interfacce dei Layer sovrastanti. E' importante sottolineare però come l'introduzione del supporto al protocollo UDP renda necessario un sistema di ordinamento dei pacchetti ricevuti, non richiesto invece nell'attuale implementazione.

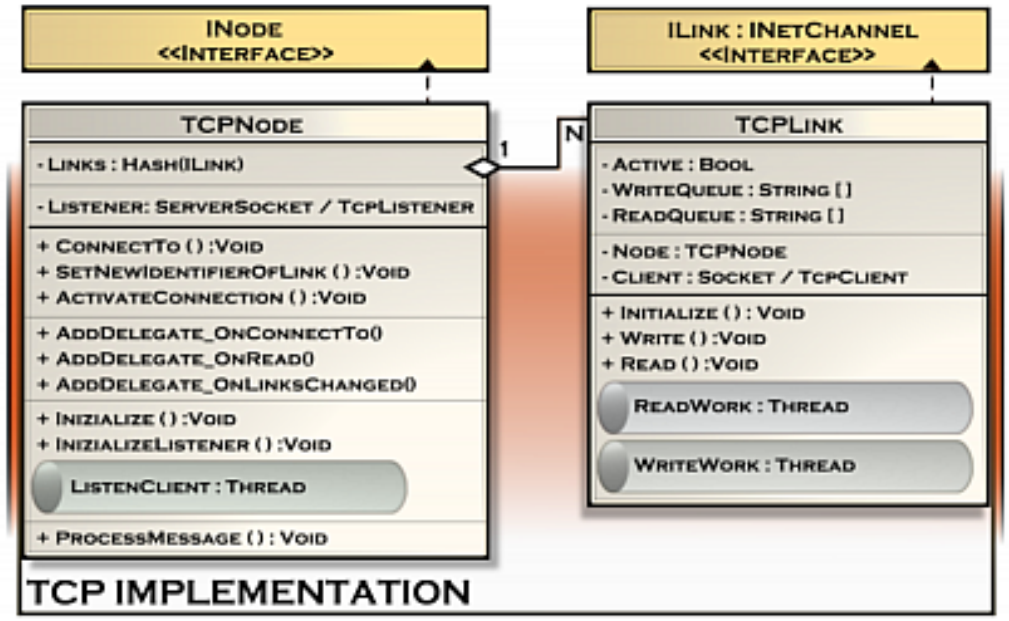

Figura 3.4: Classi principali - Layer TCP

- TCPNode: Concretizza l'interfaccia INode basandosi sul protocollo TCP. Per rimanere in ascolto di connessioni esterne, viene utilizzato un oggetto della classe *TcpListener* del FrameWork .NET.
  Quando si vuole connettere un TCPNode *A* ad un TCPNode *B*, sul primo viene chiamato il metodo *ConnectTo('B')* che creerà automaticamente un TCPLink verso *B*, mentre sul secondo, quando sarà ri-

levata la connessione da parte di A, verrà creato un TCPLink verso quest'ultimo.

- TCPLink: Concretizza l'interfaccia ILink basandosi sul protocollo TCP. Attraverso due Thread sarà possibile rispettivamente leggere e scrivere sullo stream dal proprio TCPNode all'EndPoint specificato, e viceversa.

  Quando sullo stream viene individuato un messaggio, cioè quando si presentano i caratteri terminatori "$\backslash r \backslash n$", questo viene memorizzato in un'apposita coda, e successivamente, passato al metodo *ReadLine(string message)*. In *ReadLine()* viene chiamato a sua volta il metodo *ProcessMessage(ILink link, string message)* del proprio TCPNode, che si occuperà di processare il messaggio.

3.2.1 Commands

La maggior parte dei messaggi scambiati tra i nodi appartenenti alla Membership è realizzata tramite i *Commands*.
L'idea che sta alla base dei *Commands* ricalca quella enunciata dal *Command Pattern*.
Sostanzialmente ogni comando è implementato in un'apposita classe che concretizza l'interfaccia *ICommand*, ed ha una duplice funzione:
Da una parte permette di generare in modo automatico e uniforme la stringa da inviare come messaggio, mentre dall'altra, attraverso il metodo "*Exec()*", esegue le operazioni che caratterizzano il comando stesso.

A livello di Layer NET, questi sono implementati dalla classe base *NetCommand*, estesa poi per ogni singolo comando. Un *NetCommand* è individuato dai *Parameters*, dal *INode* destinatario e dal *ILink* mittente.
Tra questi vi sono i 3 comandi dedicati all'*HandShaking* (si veda Figura 2.1):

- HandShakeBegin: Nel suo metodo *Exec()* viene chiamato *Node.SetNewIdentifierOfLink(ILink Link, string Parameters)*, che confermerà o aggiornerà l'identità del Link conosciuto dal Node sul quale è stato ese-

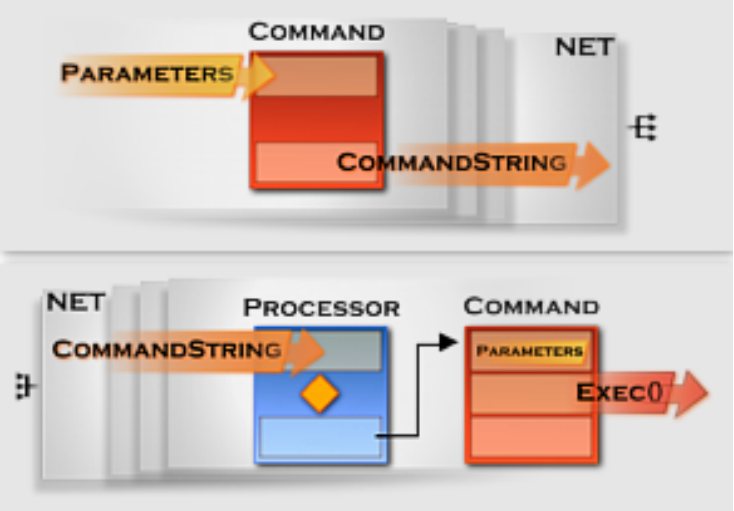

Figura 3.5: Per inviare un comando da gestire a livello NET, da un Nodo *A* ad un Nodo *B*, viene prima creata un'istanza dell'apposita classe ereditata da NetCommand, ed inizializzata secondo i parametri desiderati; quindi serializzata attraverso l'ovveride del metodo *ToString()* e scritta sullo stream del Link *A*-*B*.

Quando il messaggio giunge a *B*, viene prima invocato il *Processor* appartenente al Layer NET. Se il messaggio deve essere eseguito sul Layer NET, allora tale *Processor* istanzierà automaticamente, grazie alla *Reflection*, un oggetto della classe che definisce il comando, e, a seconda dei casi, lo eseguirà direttamente, chiamando il suo metodo *Exec()* o lo inserirà in una coda di esecuzione. Se invece non si tratta di un messaggio da eseguire sul Layer NET, allora l'INode destinatario chiamerà gli eventuali metodi delegati registrati al proprio evento OnRead, cosicché i Layer superiori possano processare il messaggio allo stesso modo.

guito il comando, e quindi *Link.Write(new HandShakeResponse(Node.GetNetID()).ToString())* , che di fatto corrisponderà alla risposta che il Node invierà al Link per notificare la propria identità.

- HandShakeResponse: analogamente a HandShakeBegin, viene prima chiamato *Node.SetNew- IdentifierOfLink(ILink Link, string Parameters)* (questa volta sull'altro Nodo), e quindi *Link.Write(new HandShakeEnd(Node.GetNetID()))* per notificare a Link l'avvenuta ricezione del comando *HandShakeResponse*. Inoltre viene flaggato a "*true*" l'attributo "*Active*" relativo alla connessione tra il Node e l'EndPoint di Link.

- HandShakeEnd: analogamente ad HandShakeResponse, sull'*Exec()* viene attivata la connessione tra il Node e l'EndPoint di Link.

Tra gli sviluppi futuri è stata considerata la possibilità di implementare un sotto-strato ulteriore dedicato alla compressione / decompressione dei dati, da inserire o a livello del *ToString()* del *NetCommand* base, o a livello dei threads Read / Write della classe *NetChannel*, oppure direttamente sotto il Layer NET.

In Figura 3.6 è illustrata la struttura della stringa che viene generata di default dall'*override* del *ToString()*.

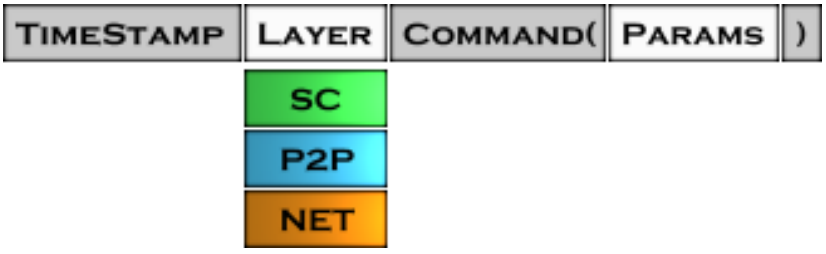

Figura 3.6: Struttura della stringa dei messaggi implementati dai Commands

3.3 Layer P2P

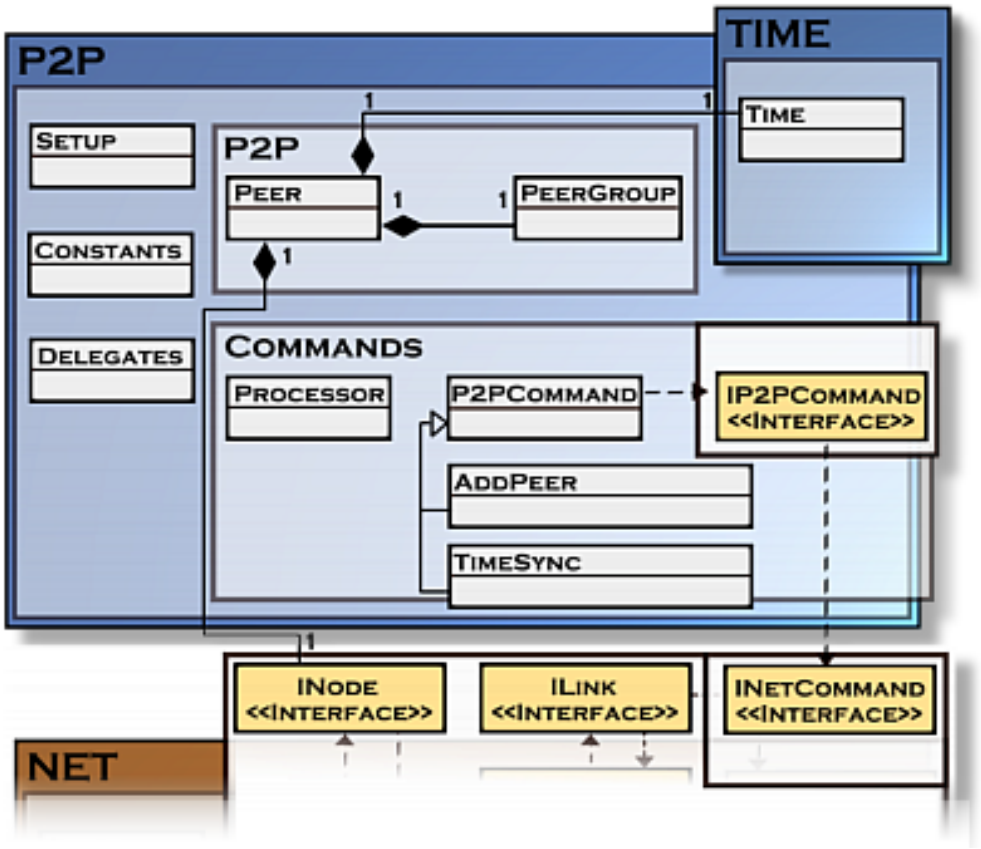

Figura 3.7: Diagramma delle Classi - Layer P2P

Il Layer P2P organizza gli INode esposti dal Layer NET sottostante in una Membership di Peer completamente connessi tra loro.
Qui risiedono alcune classi che per nome e funzionalità sono del tutto simili alle controparti presenti nel Layer NET, ma risultano finalizzate al contesto del Layer P2P: oltre alle classi *Setup*, *Constants* e *Delegates*, vi sono i comandi *AddPeer* e *TimeSync*, entrambi ereditati da *P2PCommand*.

É presente inoltre un sotto modulo, che al momento contiene una singola classe, dedicato alla gestione del tempo (si veda la sezione 2.3.1).

Infine sono qui implementate le due classi principali che permettono di gestire la Membership: *Peer* e *PeerGroup*

La classe *Peer* è composta da un'istanza di *INode* che viene invocata per inviare e ricevere messaggi di rete. Contiene inoltre un'istanza della classe *PeerGroup*, che tiene traccia di tutti i Peer conosciuti appartenenti alla Membership, ed un'istanza della classe *Time*, che viene utilizzata per la gestione del tempo. A livello di funzionalità astrae il Layer NET esponendo i metodi *ConnectTo ()*, che fa da Wrapper all'omonimo metodo di INode, *SendTo ()*, che permette di inviare messaggi ad altri Peer specificandone il NetID, e *BroadCast ()* che invia automaticamente un messaggio a tutti i Peer conosciuti.

Alla creazione di un oggetto di tipo Peer, viene istanziato, attraverso il Factory del Layer NET, un oggetto di una classe che concretizza l'interfaccia INode (attualmente quest'ultima è individuata dalla classe TCPNode). Viene poi creata un'istanza di PeerGroup, alla quale viene aggiunto automaticamente il riferimento all'INode appena istanziato, ed un oggetto di classe Time.
Infine vengono registrati i delegati in ascolto sul Layer NET relativi ai metodi *OnConnect()*, *OnRead()* e *OnLinksChanged()*.

Quando viene invocato *OnConnect*, attraverso il metodo *AddToKnownPeers()* viene aggiunto nel proprio PeerGroup il Peer che vi si è appena connesso.
Inoltre viene notificata l'esistenza di quest'ultimo a tutti i gli altri Peer conosciuti (quelli già inclusi nella Membership), tramite il comando *AddPeer()* parametrizzato con il suo NetID.

Sull'*Exec()* di *AddPeer()* viene chiamato a sua volta un *ConnectTo()* verso il NetID passato come parametro: in questo modo, quando subentra un nuovo Peer all'interno della Membership, tutti i Peer presenti vi si connetteranno automaticamente (si veda la sezione 2.2).

La classe PeerGroup è quella che, associata alla classe Peer, concretizza il concetto di Membership.

Si presenta come un piccolo Manager che gestisce una lista di identificatori (NetID) sotto forma di stringhe: non è quindi vincolato dall'implementazione della classe Peer. Fornisce inoltre due metodi di serializzazione e deserializzazione che consentono di trasformare in stringa (e viceversa) *CSV*[5] la lista di identificatori, per poter essere facilmente scambiata, attraverso il Layer NET, tra due Peer.

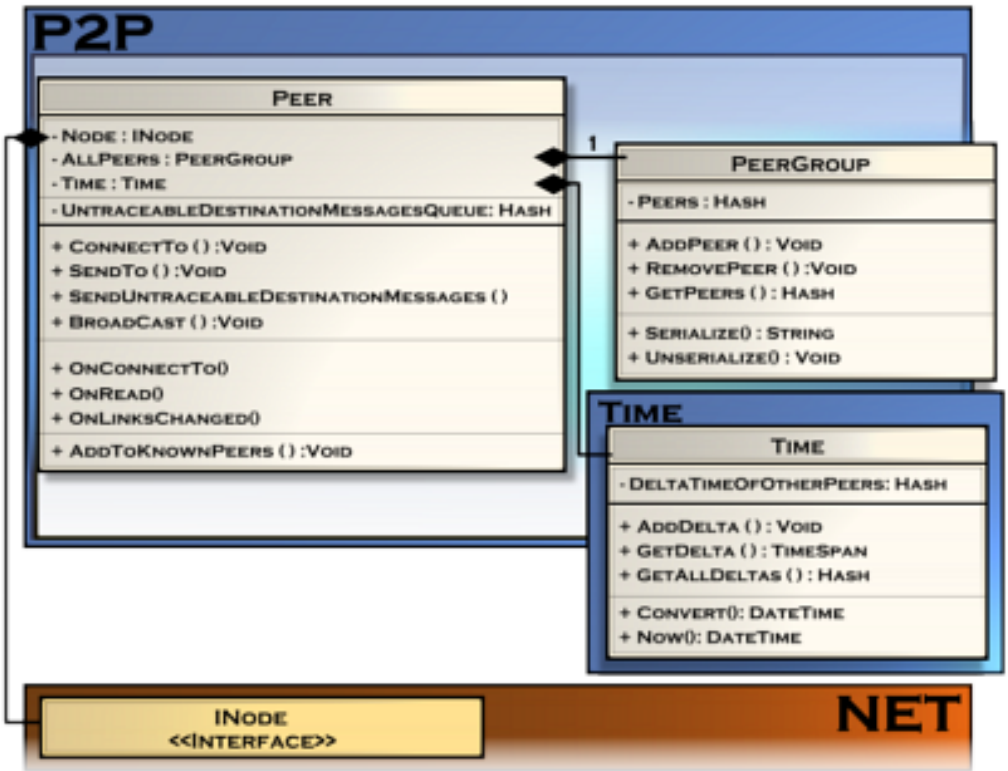

Figura 3.8: Classi principali- Layer P2P

[5]Comma-Separated Values

3.3.1 Modulo Time

Il modulo Time è attualmente composto da un'unica classe: *Time*. Questa contiene una lista di *TimeSpan* (.NET Framework) indicizzati per *NetID*, ed espone alcuni metodi accessori per la sua gestione: *AddDelta ()*, *GetDelta () e GetAllDeltas ()*.
Implementa inoltre un metodo polimorfo *Convert ()* che permette di convertire, laddove possibile, *TimeStamp* calcolati sulla base temporale di altri Peer rispetto al proprio (si veda 2.3.1).

Infine, grazie al metodo *Now()* è possibile ricavare il proprio *TimeStamp*. Tale metodo utilizza la proprietà statica del .NET framework *DateTime.Now*. In fase di debug e di analisi delle prestazioni, è risultato molto utile aggiungere 2 fattori aleatori: il primo emula una differente timezone (od un differente settaggio dell'orario di sistema), mentre il secondo introduce un fattore di Lag tra 0 e 1000 millisecondi.

La lista contenuta in Time viene popolata periodicamente grazie al comando *TimeSync*: ogni Peer invia agli altri Peer conosciuti un comando *TimeSync* contenente il proprio *TimeStamp*. Quando questo viene ricevuto, verrà eseguito quanto implementato nel suo metodo *Exec()*, ovvero verrà calcolato il delta tra il TimeStamp ricevuto e quello ritornato dal metodo *Now()* relativo all'istanza Time appartenente al Peer destinatario. Quindi inserito nella lista, indicizzato utilizzando il NetID del mittente.

3.4 Modulo SharedComponents

Il modulo si basa sul concetto di *SharedComponent*, implementato nell'omonima classe. Le istanze di tale classe sono gestite e memorizzate in un'apposito Manager (*ISharedComponentsManager*), e create da un'apposito Factory (*ISharedComponentsFactory*).

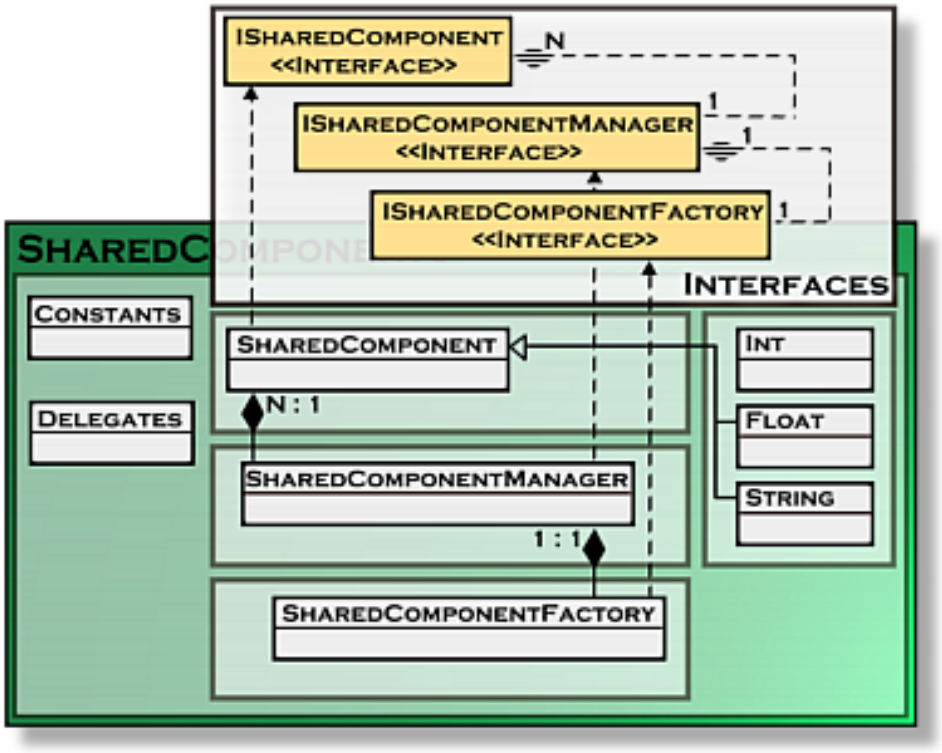

Figura 3.9: Diagramma delle Classi - Modulo SharedComponents

Come illustrato nel capitolo 2.4, ogni *SharedComponent* è identificato da un *GlobalID*, qui implementato attraverso l'attributo *ID* ed un *Value*, qui rappresentato sotto forma di stringa nell'attributo *SerializedValue*: tipicamente, comunque, ogni classe che eredita da SharedComponents contiene al suo interno una variabile privata appositamente tipizzata, adibita alla memorizzazione del valore vero e proprio.
Facoltativamente possono essere valorizzati anche gli attributi *ParentID* e *LocalID*, se l'istanza di *SharedComponent* si tratta di un sotto-componente.
Inoltre ogni istanza di *SharedComponent* può essere "*taggata*" per essere raggruppata, individuata o filtrata in maniera efficace.

Dal punto di vista funzionale, sono presenti quattro metodi virtuali di inizializzazione: il primo, *Initialize()*, chiama in sequenza gli altri tre: *InitializeClass()*, *InitializeSubComponents()* e *InitializeUpdate()*. Possono subire quindi un'*override* nel caso le operazioni di default non implementino le funzionalità richieste, o lo facciano solo in parte.
Ad esempio il componente *Object3D* esegue sia un *override* di *InitializeClass()*, nel quale viene aggiunto un Tag "*View*" dal momento che si tratta di un componente automaticamente legato ad un elemento grafico, sia un *override* di *InitializeSubComponents()* per istanziare i due sottocomponenti *Vector3 Position* e *Vector3 Rotation*. Ogni componente, di default, è taggato con il tag *Constants.ComponentTag_NotifyChange*, che ne attiva la notifica automatica sulle modifiche apportate al valore del componente stesso.

La funzione di *InitializeUpdate()* è strettamente legata al metodo virtuale *Update()*: di default infatti, se il componente che eredita da SharedComponent esegue un *override* di Update, in *InitializeUpdate* questo verrà registrato nel proprio *Manager* di appartenenza come componente da aggiornare ad ogni iterazione.
SharedComponent espone inoltre una serie di metodi accessori per la gestione automatizzata dei sotto-componenti (*Add()*, *Remove()*, *FindByLocalID()*,

FindByGlobalID() e *FindByType()*) e per la gestione dei Tags (*Tag_Add()*, *Tag_Remove()* e *Tag_Has()*).

Infine è presente un metodo virtuale *OnChange()*, automaticamente invocato ogniqualvolta viene modificato il *Value* del componente. In questo metodo, se si tratta di un proprio componente (non appartenente ad un contesto esterno, quindi) e se questo è taggato con *Constants.ComponentTag_NotifyChange*, allora viene chiamata la funzione *NotifyChange()* del *SharedComponentsManager*.

La classe *SharedComponentsManager* ha la funzione principale di mantenere e gestire la lista di tutte le istanze di *SharedComponent*, sia quelle create direttamente dal proprio Factory (un oggetto di una classe che concretizza *ISharedComponentsFactory*), che quelle provenienti da un contesto esterno. É identificata da un *Name*. Quest'ultimo sarà estremamente importante per la gestione del contesto logico introdotta dal Layer Logic.

Sono inoltre implementati i seguenti metodi ausiliari:

- *FindByGlobalID()*: per cercare un componente all'interno della lista conoscendo il solo GlobalID.
- *GetOnlyMine()*: per ricavare solo i propri componenti, cioè quelli direttamente generati dal proprio Factory.
- *IsMine()*: per sapere se il componente passato come parametro, appartiene o meno a questo Manager.
- *AddToUpdater()*: aggiunge il componente passato come parametro alla lista dei componenti da aggiornare ad ogni iterazione.
- *RemoveFromUpdater()*: rimuove il componente passato come parametro dalla lista dei componenti da aggiornare ad ogni iterazione.

- *Update()*: il metodo da chiamare ad ogni iterazione per aggiornare tutti i componenti registrati.

Infine vi sono i quattro metodi utilizzati per invocare gli eventi *OnCreate*, *OnChange*, *OnActive* e *OnDestroy*.
OnCreate viene invocato, attraverso il Factory, sulla creazione di un proprio componente, mentre *OnDestroy* viene invocato, sempre attraverso il Factory, sulla sua distruzione. *OnChange* viene invocato quando cambia il valore di un proprio componente, e *OnActive* quando quest'ultimo cambia stato e diventa attivo (cioè quando ogni suo eventuale sotto-componente è stato creato ed inizializzato). Tali eventi verranno poi ascoltati dal Layer Logic, che si occuperà di propagarne la loro notifica attraverso la rete.

La classe *SharedComponentFactory* viene utilizzata per creare e distruggere istanze di *SharedComponents*.
Quando il componente da creare appartiene al Manager legato all'istanza del Factory in questione, allora verrà creato attraverso il motodo *Create()*.
Quando invece il componente da creare appartiene ad un altro contesto (tipicamente sono copie di componenti appartenenti ad altri Peer), allora viene invocato il metodo *CreateFromOtherContext()*.

3.5 Layer Logic

Il Layer Logic è sostanzialmente diviso in due regioni, differenziate dal livello di generalizzazione implicito nelle classi che ne fanno parte.
La regione più *Generic Oriented* introduce il contesto logico, individuato dalla classe *PeerContext*: questa unisce le funzionalità esposte dal *Layer P2P* e quelle caratteristiche del Data Layer rappresentato dal modulo *SharedComponents*.

La classe *PeerContext*, infatti, é composta da un'istanza di *Peer* e una di *SharedcomponentsManager*; queste risultano legate tra loro tramite il mede-

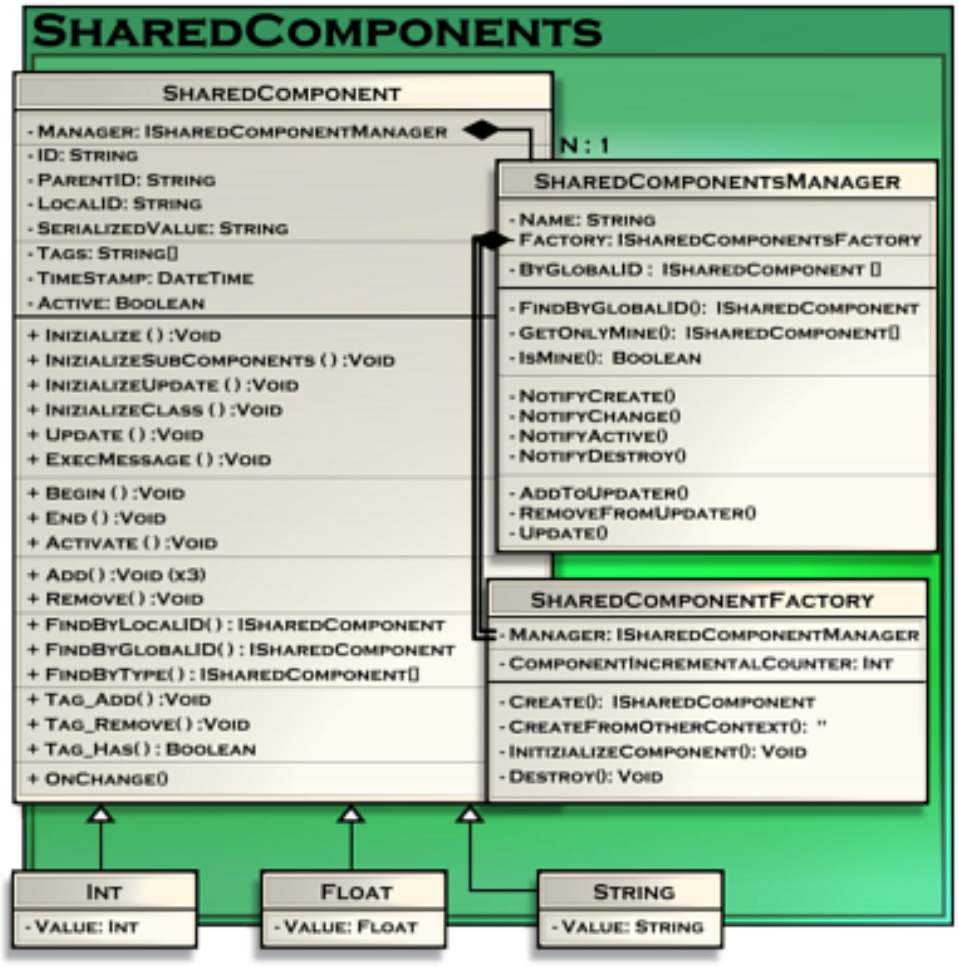

Figura 3.10: Classi Principali - Modulo SharedComponents

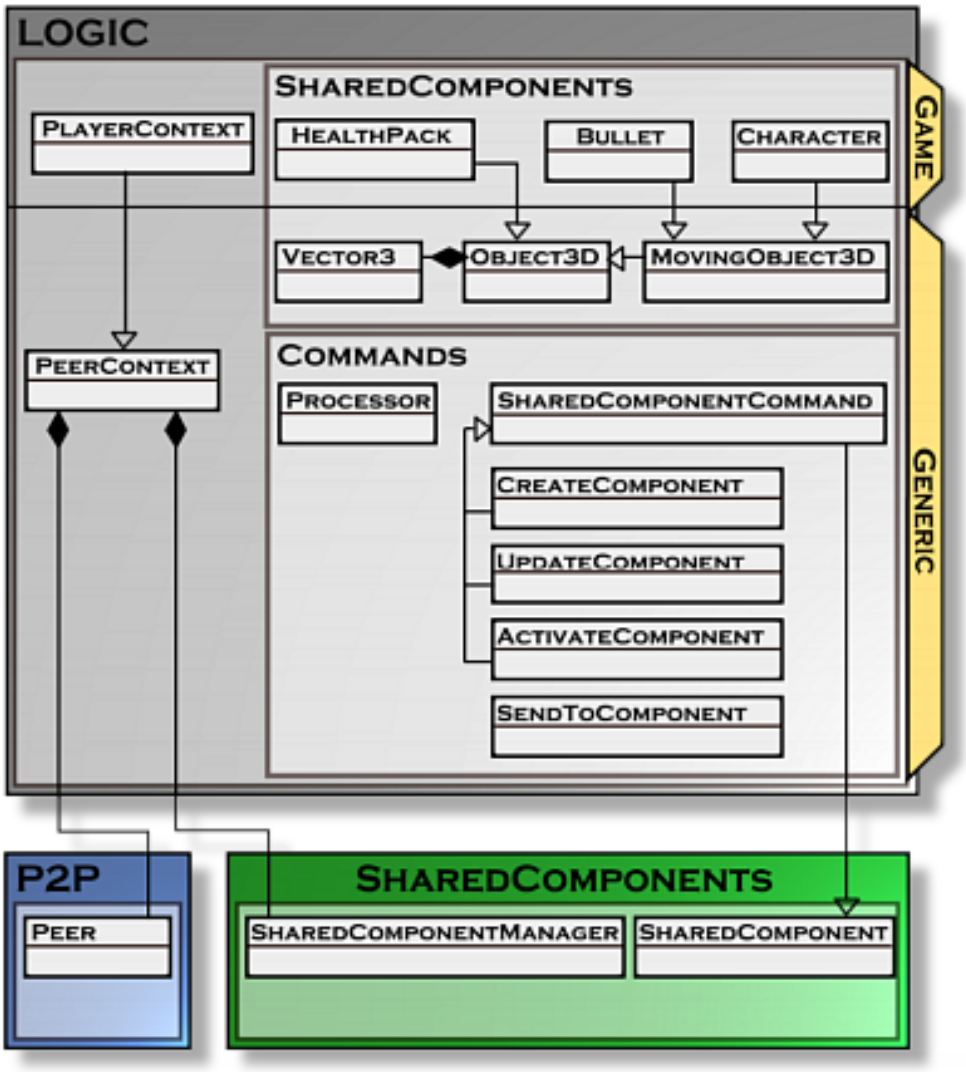

Figura 3.11: Diagramma delle Classi - Layer Logic

simo identificativo: l'attributo *NetID* dell'istanza di *Peer* e l'attributo *Name* dell'istanza di *SharedComponentsManager* hanno lo stesso valore.

Tale vincolo ha l'effetto implicito di estendere la proprietà di uno SharedComponent, non più al solo Manager ma anche all'intero Peer.

Sono inoltre implementati i *Commands* che permettono la sincronizzazione automatica del DataLayer tra i diversi Peer (*CreateComponent*, *UpdateComponent*, *ActivateComponent* e *SendToComponent*) ed i componenti che consentono di rappresentare le informazioni più comuni per l'identificazione si un elemento grafico all'interno di un generico videogioco tridimensionale (*Object3D* e *MovingObject3D*).

La seconda regione è più *Game Oriented*, nel senso che da qui in poi iniziano a presentarsi classi che implementano logiche strettamente legate alle specifiche di gioco e alle relative dinamiche di GamePlay.
Qui si trova la classe *PlayerContext*, ereditata da *PeerContext*, che da una parte astrae il contesto legato al concetto di "*Peer*", limitandolo a quello più specifico di *Player*, e dall'altra implementa una serie di logiche che caratterizzano quest'ultimo: ad esempio viene eseguito un *override* sul metodo *Join*, nel quale viene istanziato un componente di tipo *Character*; in questo modo, quando inizia la partita, ogni Player creerà automaticamente nel proprio contesto logico un'istanza *Character*, la quale sarà quindi condivisa tra tutti i Peer.

3.6 Layer View

Come evidenziato nel capitolo 2.6, il Layer View è molto vasto, e fa fronte ad un notevole numero di problematiche e funzionalità, molte delle quali non sono strettamente connesse all'argomento di tesi.
Aspetti come il Rendering, la gestione degli Input, la gestione della riprodu-

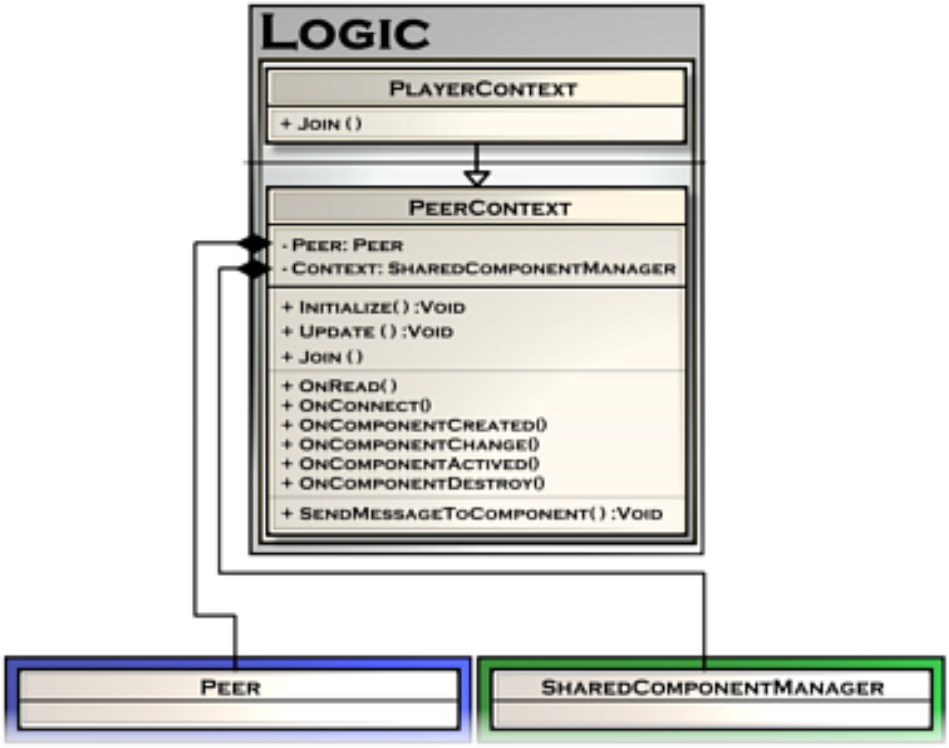

Figura 3.12: Classi Principali - Layer Logic

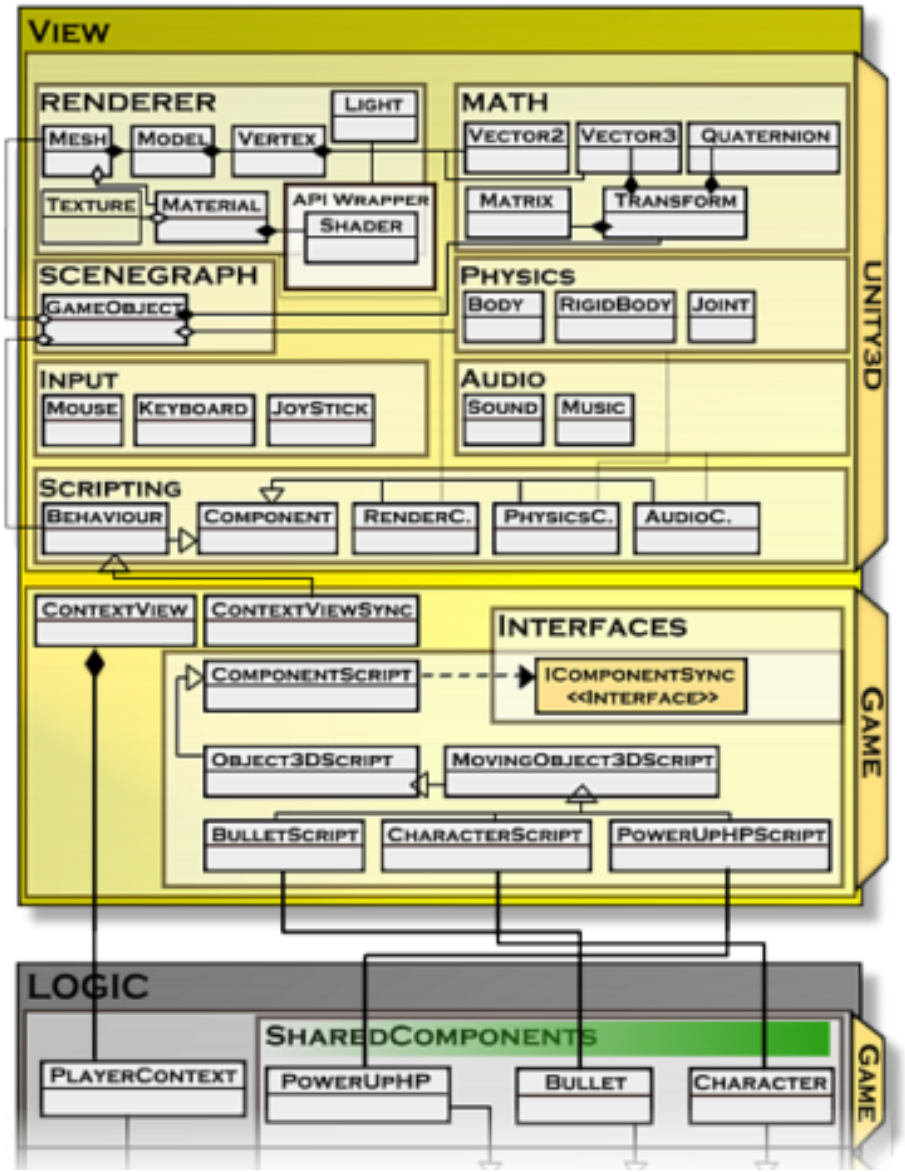

Figura 3.13: Diagramma delle Classi - Layer View

zione Audio e la simulazione della fisica, sono infatti delegati agli strumenti messi a disposizione da *Unity3D*.
Quest'ultimo permette di definire i comportamenti che caratterizzano gli elementi presenti nella scena, mediante un sistema di "*Scripting*" basato sulla classe "*Behaviour*".

La classe principale, *ContexView*, concretizza il contesto "*View*" e fa da "*Bridge*" tra questo ed il contesto "*Logic*" via *composition*. Risulta essere quindi il punto d'accesso a tutti i Layer sottostanti (Figura 3.14).

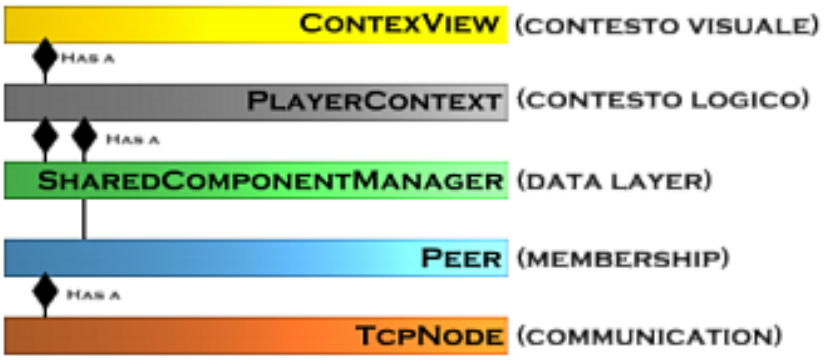

Figura 3.14: Catena dei Layer, legati via composition

Parallelamente la classe *ViewContextSync*, estesa dalla classe *Behaviour*, si occupa della sincronizzazione tra il contesto logico e quello visuale.

Ad ogni iterazione, infatti, esegue un polling sul Data Layer, controllando che non vi siano componenti logici taggati con il tag "*View*" senza una rispettiva controparte visuale all'interno della scena 3D. Quando ciò accade, viene istanziato automaticamente un elemento grafico ed applicata ad esso una classe (che segue la naming convention "[nome componente]Script"), estesa da *ComponentScript* (*Behaviour*), che ne definisce il comportamento (si veda 2.6).

La classe *ComponentScript* è definita secondo il seguente schema:

- String LogicComponentID: attributo al quale viene assegnato il valore del GlobalID appartenente allo *SharedComponent* che rappresenta lo stato logico dell'elemento grafico.

- Update(): il metodo che viene invocato ad ogni iterazione. Il suo comportamento di default, eventualmente modificabile via *override*, prevede la chiamata di *Update_Writer()* se lo *SharedComponent* assegnato appartiene al contesto logico del Player. In caso contrario viene chiamato *Update_Reader()*. Infine, attraverso *CheckDestroy()*, viene verificata l'esistenza del componente logico all'interno del Data Layer: se questo non è più presente, allora verrà distrutto l'elemento grafico corrispondente (e di conseguenza anche il relativo *ComponentScript*)

Ad esempio, il *ComponentScript* relativo al componente di tipo *Character* (cioè la classe *CharacterScript*), all'interno del metodo *Update_Writer()* eseguirà l'aggiornamento periodico delle coordinate e della rotazione del personaggio controllato dal Player, mentre nel metodo *Update_Reader()* vi sarà implementata una logica di *Dead Reckoning* (Figura 3.15).

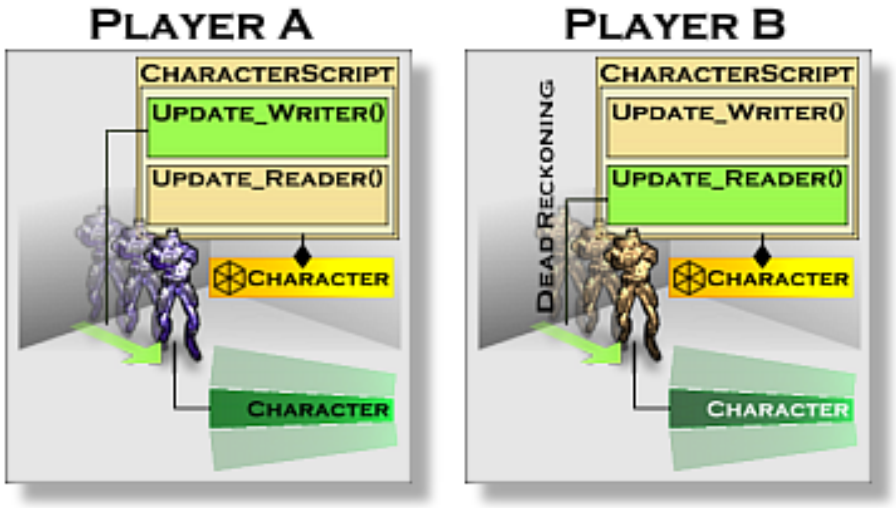

Figura 3.15: CharacterScript

Capitolo 4

Analisi delle prestazioni

Il contesto nel quale è stato sviluppato il progetto rende difficoltosa l'individuazione di parametri e delle relative misurazioni che si possono effettuare, invece, in scenari più ampi e strutturati. Misure derivanti da fattori come la latenza, la sua variazione ed il numero di giocatori non sono facilmente simulabili in modo realistico in un contesto locale.
In questo senso è stato implementato un sistema che consente di simulare la presenza di Lag tra i diversi Peer che compongono la Membership, ritardando l'esecuzione dei comandi inviati e ricevuti.

Introducendo infatti un valore di latenza relativamente alto (200 / 500 millisecondi) in uno dei Peer connessi alla Membership di prova è stato possibile ricavare alcune considerazioni qualitative:
I processi di interazione che coinvolgono elementi (SharedComponent) gestiti dal Peer soggetto a Lag, risultano rallentati su tutti i Peer.
Mentre, dal punto di vista del Peer soggetto a Lag, tale ritardo si ripercuote su qualsiasi tipo di aggiornamento proveniente dagli altri Peer.
Come era prevedibile, l'esperienza di gioco risulta quindi essere abbastanza compromessa, specialmente per i Peer con la latenza più elevata.

Un primo scenario analizzabile quantitativamente in un contesto locale può essere individuato nella misurazione del tempo trascorso tra l'invio di un

messaggio da parte di un Peer e la sua esecuzione da parte degli altri Peer appartenenti alla Membership. Questo prova è stata eseguita istanziando 10 oggetti di classe *PeerContext* (si veda la sezione 3.5): Il "*Peer 1*" invia 7 aggiornamenti consecutivi di un componente presente nel proprio Data Layer, incrementandone ogni volta il valore all'interno di un ciclo *For*.
In Figura 4.1 sono riportati i risultati sottoforma di grafico a dispersione: i diversi stati di gioco (in questo caso limitati al valore del singolo componente modificato) sono rappresentati da linee curve che interpolano i punti nei quali i diversi Peer (linee verticali) eseguono l'aggiornamento del componente in questione.

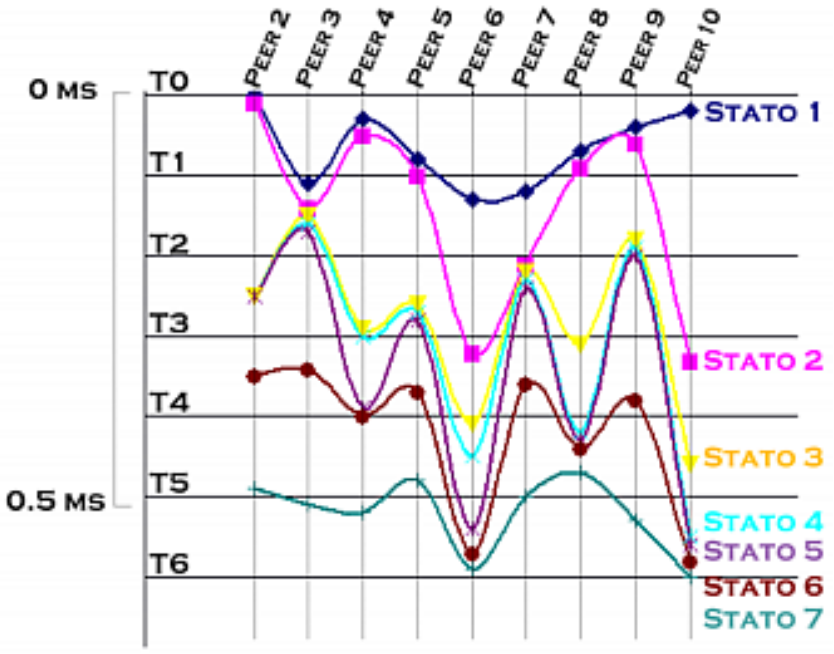

Figura 4.1: Tempo di ricezione ed esecuzione di un aggiornamento di un componente logico

L'esecuzione degli aggiornamenti, nonostante si presenti nel medesimo ordine sui diversi Peer (grazie all'utilizzo del TCP), non avviene in modo sincronizzato: lo "*Stato 5*", ad esempio, viene eseguito dal "*Peer 2*" in T2,

e dal "*Peer 10*" in T6.

Il TimeSpan preso in esame è comunque estremamente ridotto (minore di un millisecondo), e le variazioni riscontrate sono da ricondurre alla politica con la quale vengono schedulati i Thread di lettura e scrittura utilizzati da ogni Peer per comunicare con gli altri Peer appartenenti alla Membership .

Utilizzando un sistema di sincronizzazione "*Stop-And-Wait*" (si veda la sezione), tali variazioni non si sarebbero riscontrate; o meglio, l'istante di tempo di esecuzione sarebbe stato comunque differente sui diversi Peer, ma all'interno di uno stesso periodo di tempo lo stato di gioco sarebbe risultato il medesimo (Figura 4.2).

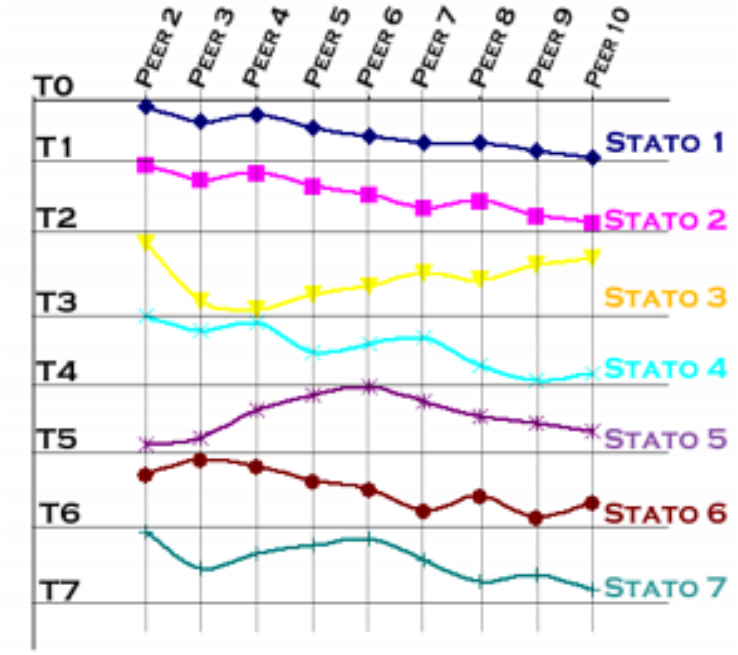

Figura 4.2: Tempo simulato di ricezione ed esecuzione di un aggiornamento di un componente logico con sistema di sincronizzazione Stop-And-Wait

Il passo successivo è stato quello di introdurre nello scenario precedente la simulazione di latenza di rete: ogni Peer è stato impostato per simulare una latenza variabile da 5 a 30 millisecondi, ad esclusione del "*Peer 3*",

la cui latenza è stata settata sui 100 millisecondi. La frequenza di invio degli aggiornamenti è stata abbassata appositamente a 10 millisecondi per consentire una migliore leggibilità del grafico (Figura 4.3).

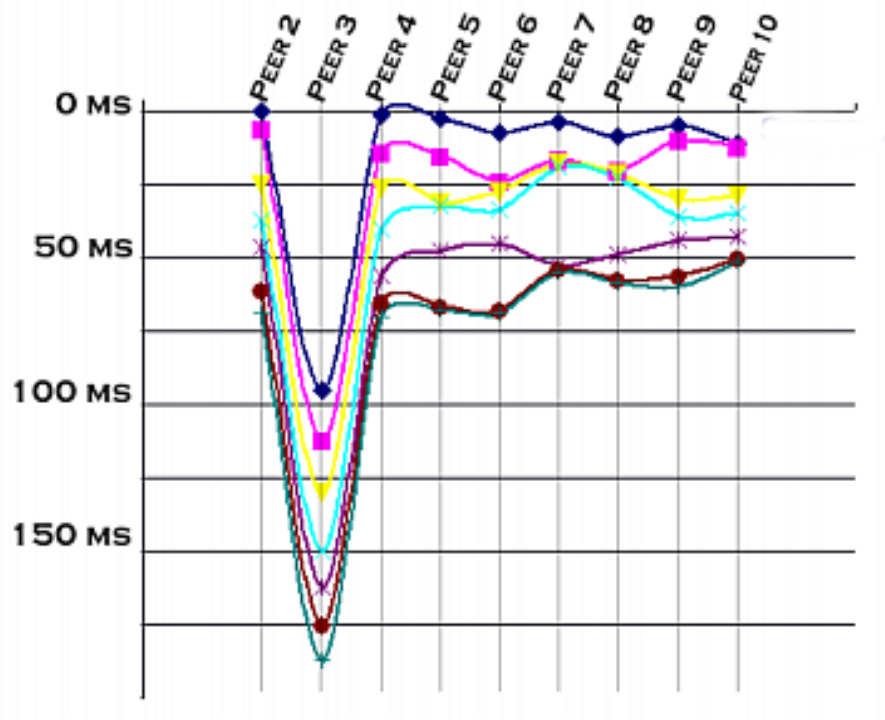

Figura 4.3: Tempo di ricezione ed esecuzione introducendo latenza di rete

Come era facile prevedere, il "*Peer 3*" riceverà ed eseguirà gli aggiornamenti in ritardo rispetto agli altri Peer. L'ordine di esecuzione rimane comunque coerente: graficamente questo è visibile dal fatto che non sono presenti linee che si intersecano.

Queste misurazioni sono state effettuate disabilitando il Layer View poichè la richiesta di risorse da parte del Renderer non consentiva l'esecuzione contemporanea di più di 4 Peer.
Utilizzando Unity3D si è vincolati all'utilizzo di un unico Thread, sul quale processare alternativamente la logica prima ed il rendering poi; quest'ultimo

è in assoluto il processo che richiede più risorse, sia a livello di utilizzo CPU che di memoria (Figura 4.4).

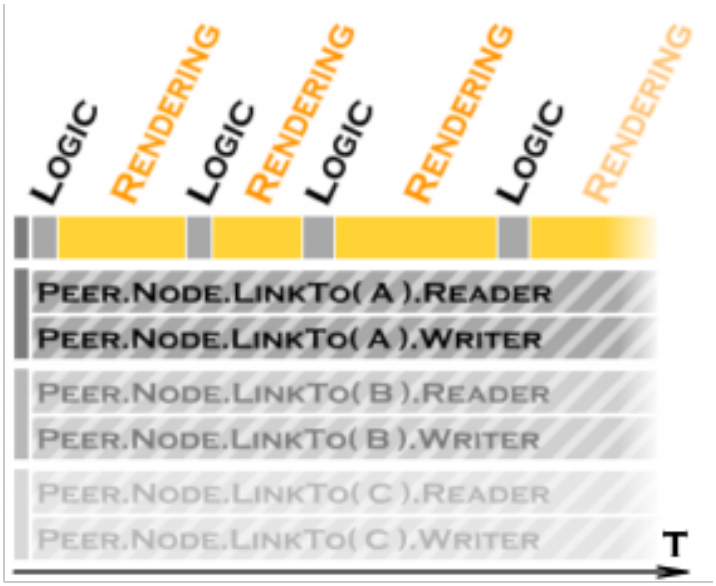

Figura 4.4: Threads di esecuzione: il Thread principale è gestito da Unity3D e processa alternativamente le dinamiche logiche e la pipeline di rendering. Per ogni Peer appartenente alla Membership vengono inoltre istanziati due Thread, uno di lettura e uno di scrittura.

Successivamente è stato analizzato uno scenario comprensivo del Layer View: sono stati misurati il numero di messaggi scambiati tra 4 Peer durante una sessione di gioco. Il *Update_Write()* di ogni *CharacterScript* è stato modificato in modo da produrre aggiornamenti sufficientemente realistici per lo scenario di analisi, in maniera automatica, senza cioè che sia richiesta la presenza di un giocatore reale, simulando il movimento del Character e la generazione di Bullets.

Da i risultati ottenuti è emersa l'esistenza di alcune fasi critiche: facendo riferimento alla Figura 4.5, in T2, T4 e T6 sono individuabili periodi di picco, esponenzialmente crescenti, sia per quanto rigurada il numero di messaggi

inviati, che per la durata dei periodi stessi, in funzione dell'incremento del numero dei Peer connessi alla Membership. Tali periodi sono contestualizzati all'interno della fase di sincronizzazione del Data Layer che avviene conseguentemente al *join* di un nuovo Peer all'interno della Membership, nella quale ogni Peer è tenuto ad inviare la lista dei propri componenti presenti nel modulo *SharedComponents* (capitolo 2). In T1, T3, T5 e T7, invece il numero totale di messaggi scambiati ritorna nel range di 0-5 messaggi ogni 0.25 secondi. Chiaramente anche quest'ultimo sarà destinato a crescere al crescere del numero di Peer connessi.

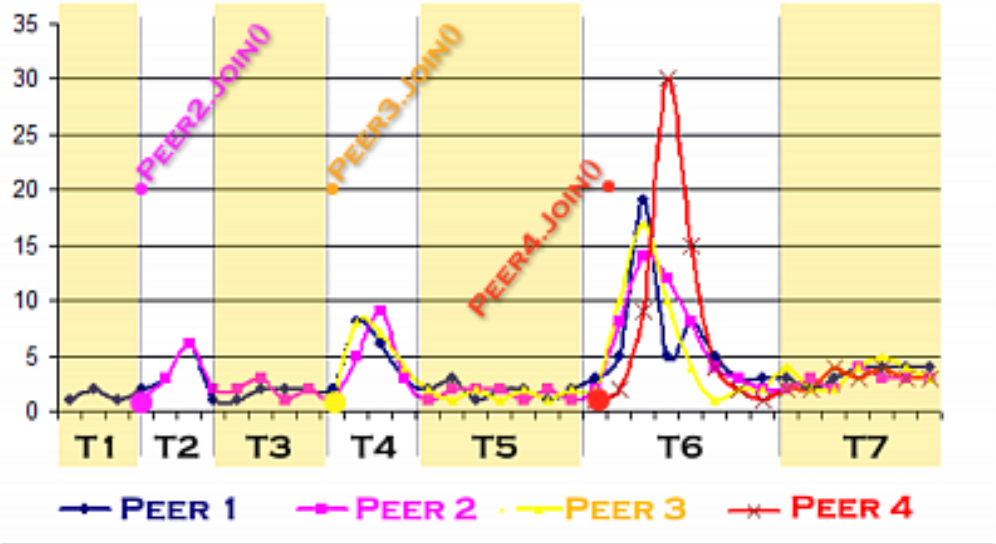

Figura 4.5: Messaggi scambiati in una sessione di gioco:

Capitolo 5

Sviluppi futuri

5.1 Anti-Cheating

Per "*Cheating*" si intende quella attività messa in atto da alcuni giocatori ("*Cheaters*") in grado di modificare l'esperienza di gioco e trarne vantaggio. Per alterare la dinamica di gioco è possibile operare in diversi contesti e non necessariamente a livello software: si pensi ad esempio ad un gioco le cui regole prevedono un equo scontro tra N giocatori. Se alcuni di questi si coalizzano tra loro, guadagnano un vantaggio sui giocatori che invece non si sono coalizzati. La dinamica di gioco viene così alterata, e le soluzioni software per far fronte a questo problema sono, in questo caso ("*Cheating by collusion*"), inefficaci.

Tendenzialmente per far fronte a queste tipologie di Cheating è necessario lavorare direttamente sul Gameplay, cercando di evitare, cioè, di far emergere meccaniche "sociali" di questo tipo, ed eventualmente disincentivarle, rendendole notevolmente svantaggiose per il giocatore che le sfrutta. Non sempre è possibile: il solo atto di disconnessione (o logout) durante una partita, non solo turba pesantemente il flusso e l'assetto di gioco, ma è anche impossibile evitare.

Un altro scenario particolarmente interessante che non coinvolge problematiche e soluzioni software è quello del cosiddetto "*Gold farming*", ovvero la compravendita, esterna al contesto di gioco, di items e abilità virtuali in cambio di denaro reale.

I sistemi di Cheating basati su exploit software operano su diversi livelli. E' possibile infatti modificare il Client in modo da automatizzare alcuni processi, rimpiazzando di fatto il giocatore, che in questo modo può dedicarsi ad altri aspetti di gioco: in un RTS, un Cheat Software può occuparsi della gestione economica del regno virtuale, lasciando le sezioni di combattimento in mano al giocatore, o viceversa. O ancora, si possono visualizzare dati e informazioni presenti nel Data Layer ma volutamente nascoste nel Presentation Layer, ad esempio eliminando la cosiddetta "*Fog Of War*" e quindi, di conseguenza, conoscere l'intera area di gioco, anche quella inesplorata.

In ambito FPS, invece, i sistemi di Cheating più utilizzati sono da una parte quelli che permettono di migliorare le prestazioni del giocatore, attraverso lo *Speed Hacking*, che aumenta la velocità del Character, e l'*Auto Aiming*, che ruota la visuale automaticamente sul target più vicino, e dall'altra quei sistemi che si basano sulla riduzione e sull'alterazione della qualità grafica, in modo tale da evidenziare o visualizzare elementi grafici altrimenti poco visibili o addirittura nascosti.

Tipicamente questa tipologia di Cheat opera lato Client e pertanto, attraverso sistemi quali *Punk Buster*[1] e *AntiTCC*, è possibile in qualche modo limitare la loro azione. Tali sistemi infatti eseguono una scansione locale, sia a livello di FileSystem, verificando l'integrità dei file utilizzati dal videogioco, che a livello dei processi e relativa memoria. Se la scansione di *Integrity Check*, solitamente avviata ad inizio e fine partita, o in concomitanza di particolari situazioni di gioco, ad esempio subito dopo un *Frag*, fallisce, allora il

[1] http://www.evenbalance.com

giocatore viene esplulso dalla partita.

Sistemi di Cheating come lo *Speed Hacking* e *Auto Aiming*, però, possono essere implementati anche ad un livello inferiore rispetto a quello applicativo individuato dal Client. Utilizzando appositi programmi esterni è infatti possibile monitorare e modificare ogni singolo pacchetto inviato al Server al fine di trarne un vantaggio nel contesto delle dinamiche di gioco.

In un'architettura Client-Server si tende a delegare la logica di gioco ed i relativi controlli sul Server poichè si presume sia un'entità "*trusted*", limitando il più possibile i Clients a quelle che sono le operazioni meno sensibili dal punto di vista dell'avanzamento dello stato di gioco.
In una logica Peer-To-Peer, la mancanza di un assetto centralizzato porta ad avere come unici elementi potenzialmente "*trusted*" gli stessi Peer.
In una rete autogestita, questo implica da una parte un sistema di *Cheat-detection* su ogni singolo Peer, e dall'altra un'organizzazione interna alla Memebership in grado di individuare un'autorità adibita al *Cheat-management*.

Inoltre sistemi di sincronizizzazione temporale, come la *Bucket Synchronization*, possono essere aggirati sfruttando il ritardo tra la ricezione di eventi provenienti dagli altri Peer e la loro effettiva esecuzione: un Peer può ad esempio inviare un evento opportunamente creato bastandosi sulle "*intenzioni*" ricevute ma non ancora eseguite (*Look Ahead Cheat*). O più semplicemente modificare il TimeStamp di un evento così da risultare essere stato inviato in un istante di tempo precedente (*TimeStamp Cheat*).

Di seguito viene proposta l'implementazione di un Modulo dedicato al *Cheat-detection*: come anticipato nel capitolo 1, tipicamente i MOGs che si basano su un modello Peer-To-Peer vengono supportati comunque da un'architettura centralizzata, utilizzata come punto d'accesso alle partite, ed eventualmente come autorità per la gestione dell'Anti-Cheating. Il contesto individuato del progetto di tesi, invece, non contempla questo tipo di supporto.

Tale mancanza sposta il problema del *Cheat-detection* da un livello superiore al livello della Game Logic.

Infatti è possibile estendere la classe ComponentScript, aggiungendo la chiamata di una funzione astratta *CheatDetection()* all'interno del metodo *Update_Reader()*. Eseguendo un *Override* di questo metodo, a seconda delle caratteristiche del componente associato al ComponentScript, si possono eseguire determinati controlli specifici atti a controllare la coerenza delle informazioni giunte da altri Peer.

Per quanto riguarda il *Cheat-management* è necessario implementare un sistema "*democratico*" in grado di prendere decisioni quali, ad esempio, l'espulsione di un determinato Peer dalla Membership.

Un sistema potrebbe essere quello di attribuire ad ogni Player un punteggio di "*Fairness*". Quando, in una specifica implementazione di CheatDetection(), viene rilevata un'incoerenza sospetta tale punteggio viene decrementato. Ogni Player attribuisce agli altri Player una propria visione di Fairness: non è detto, cioè, che il punteggio di "*Fairness*" appartenente al Player *B*, secondo il Player *A*, sia lo stesso attribuitogli dal Player *C*.

Il punteggio di *Fairness* attributo da ogni Player ad ogni Player, dal momento che deve rappresentare un'informazione costantemente condivisa, può prendere forma nel Data Layer, sotto forma di SharedComponent, e gestito da un'apposito modulo dedicato all'Anti-Cheating.

In questo modo ogni Player può conoscere il punteggio di *Fairness* globale (cioè la somma di tutti i punteggi di *Fairness* che ogni singolo Peer attribuisce ad un determinato Player sospettato di Cheating), e se questo è al di sotto di una certa soglia di tolleranza, il Player incriminato sarà rimosso dalla visione della Membership di ogni Peer (Figura 5.1).

Questo tipo di meccaniche di *Cheat-management* decentralizzate e autogestite, non può però funzionare in un contesto Peer-To-Peer nel quale la maggioranza dei Player fa uso di Cheats.

Se si considera, inoltre, che le valutazioni di *Cheat-Detection* vengono eseguite su specifiche informazioni provenienti da generici messaggi in entata,

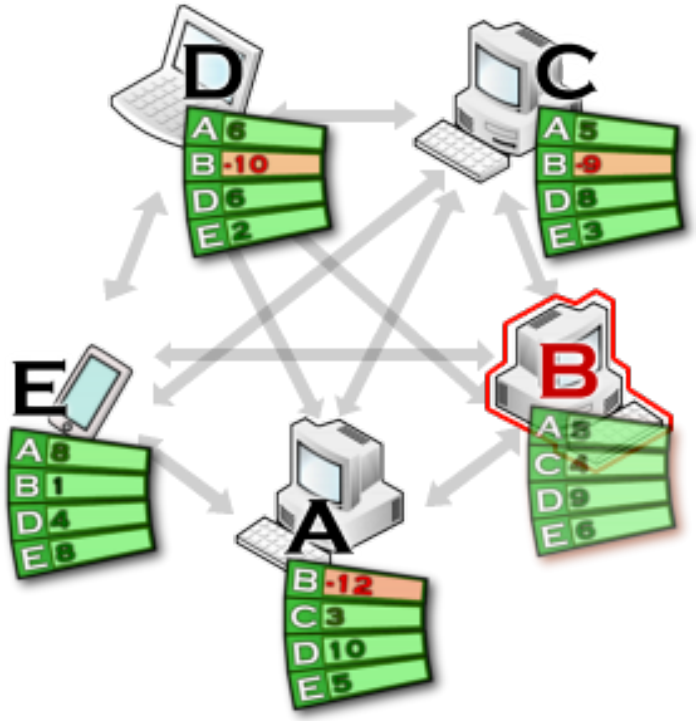

Figura 5.1: Il valore di Fairness attributo al Player B dai Player A, C, D ed E è rispettivamente di -12, -9, -10 e 1 punti. La somma di tali valori è minore di zero, e per tanto il Player B viene escluso dalla Membership

senza un supporto lato Client (come l'*Integrity Check* eseguito da *AntiTCC*) ed un supporto lato server (come gestione e tracciamento degli utenti malevoli), e dal momento che alcuni comportamenti sospetti potrebbero essere causati dalla latenza, dalla sua variazione, e dalla congestione di rete, tali sistemi non possono mirare all'eliminazione del problema di *Cheating*, ma rappresentano solo un tentativo atto ad arginarne gli effetti.

5.1.1 Time Based Cheating

Come illustrato nel capitolo 2.3.1, periodicamente ogni Peer, tramite il comando TimeSync, richiede agli altri Peer appartenenti alla Membership di notificare il valore del proprio TimeStamp, cosicché vi si possa possa calcolare, e memorizzare, il delta relativo.

Quando viene memorizzato il nuovo valore di tale delta, questo teoricamente non dovrebbe discostarsi troppo dal valore memorizzato precedentemente. Se questa differenza è maggiore di una certa soglia (che tiene conto di eventuali oscillazioni dovute al *Lag* e *Jitter*), allora viene diminuto il punteggio di Fairness relativo al Peer "sospettato" di cheating.

La soglia di tolleranza, sotto questo aspetto, può anche essere relativamente alta, poiché a questo livello, il sistema di Timing deve semplicemente garantire una velocità di esecuzione "simile" su ogni Peer. Non deve essere possibile, quindi, avere un Peer A il cui tempo avanza a velocità N ed un'altro Peer B il cui tempo invece avanza a velocità 2N. Dal momento che ogni singolo comando in grado di apportare modifiche allo stato di gioco individuato nel modulo SharedComponents è accompagnato dal Timestamp, sarà possibile valutare, dal punto di vista temporale, la sua coerenza.

Se si prende come riferimento il Player A in Figura 5.2 e si analizza il suo delta time rispetto agli altri Player ogni 5 secondi, emergono i seguenti comportamenti:
Il delta rispetto a B è costante per i primi due istanti di tempo, mentre al terzo questo subisce un decremento di 5 secondi, per poi ristabilizzarsi al delta originale all'istante T4.

Il Player C ha un timing due volte più veloce rispetto ad A: il delta infatti decrementa costantemente fino a invertire il segno.
Il Player D Invece ha un delta più o meno costante, ma nell'istante T3 questo invia un comando riportante un TimeStamp valorizzato come se fosse stato inviato in T4.
E' difficile valutare con sicurezza se l'alterazione del delta del Player B sia dovuta a problemi di latenza o sia stata causata dall'utilizzo di un Cheat.

Anche nel caso del Player D la valutazione risulta delicata, poichè può capitare che in un momento di congestione il TimeSync inviato dal Player D al Player Player A, venga ricevuto da quest'ultimo con un ritardo tale da giustificare il TimeStamp riportato sul comando inviato successivamente.

Nel caso del Player C, invece, il timing è palesemente accelerato (2X), e per tanto deve essere escluso dalla Membership in ogni caso.

5.1.2 Space Based Cheating

Un primo sistema in grado di scremare quelle situazioni palesemente sospette, come la velocità del Character raddoppiata, è relativamente semplice da implementare: in *CheatDetection()* di un MovingObjectScript è infatti possibile controllare, da una parte che i valori dei componenti *NextPosition* e *NextTimeStamp* siano coerenti rispetto alle velocità di movimento specificata all'interno della classe stessa (ovvero il valore sul quale, in *Update_Writer()* verrà calcolato il vettore di spostamento), e dall'altra verificare che, salvo casi particolari (ed opportunamente gestiti) come il *Respawn*[2], la distanza tra il valore del componente *Position* appena aggiornato ed il valore assunto precedentemente stia entro i limiti imposti dalla velocità di spostamento (Figura 5.3).
Da questo punto di vista Cheats come lo *Speed-Hack* sono facilmente individuabili.

[2]Azione di riposizionamento, all'interno della mappa di gioco, del Character quando quest'ultimo muore o viene distrutto

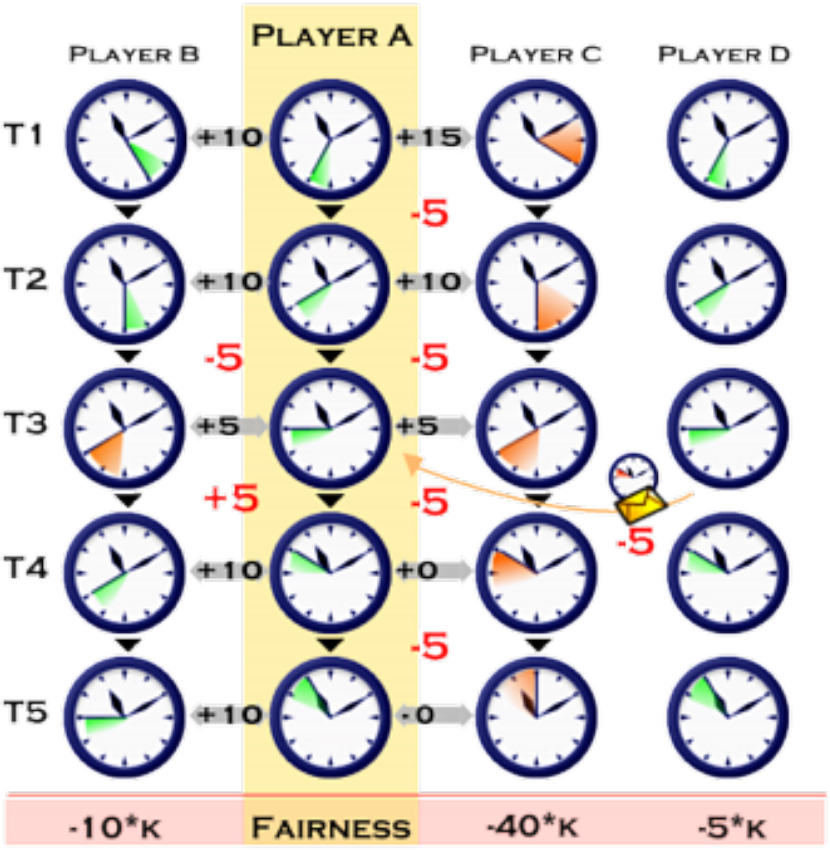

Figura 5.2: Time Based Cheating

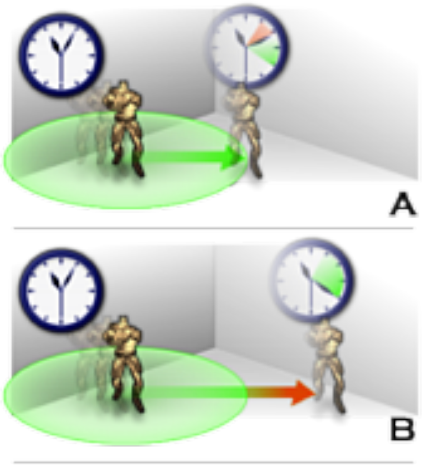

Figura 5.3: In *A* viene eseguito un movimento spazialmente coerente, ma temporalmente incoerente. In *B*, invece, il movimento eseguito risulta incoerente poichè la posizione attuale (o prevista) non ricade entro l'area di spostamento consentito nel periodo di tempo specificato (cerchio verde).

Maggiori difficoltà sorgono quando vi è la necessità di controllare il comportamento di ogni singolo movimento, non solo dal punto di vista della rapidità di spostamento, quindi, ma anche dal punto di vista della rotazione e dei relativi cambi di direzione.

Risulta impossibile valutare, infatti, se l'agilità di movimento di un Character sia dovuta all'abilità del giocatore che lo controlla, o all'utilizzo di un apposito Cheat in grado di spingere tali capacità appena al di sotto del limite imposto dalle regole di gioco.

Quello che però è sicuramente possibile (e necessario) controllare, è la coerenza delle rotazioni in relazione al TimeSpan tra un aggiornamento e l'altro: se ad esempio il gioco prevede un Character in grado di compiere una rotazione di 20° ogni secondo, nel metodo *Update_Reader()* del relativo ComponentScript sarà possibile valutare se gli eventuali cambi di direzione segnalati rientrino effettivamente nei limiti imposti dalle specifiche di gioco

(Figura 5.4).

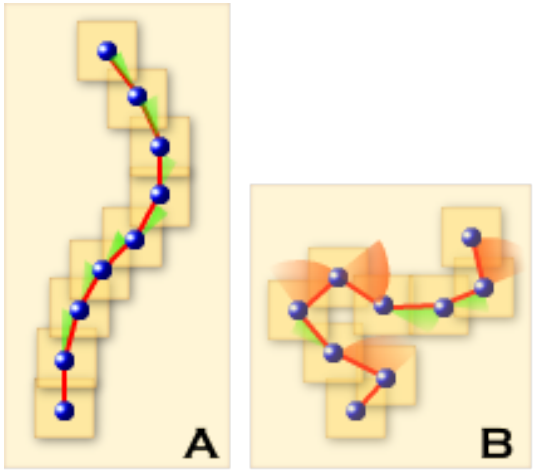

Figura 5.4: Le rotazioni effettuate dal Character (punti blu) in *A* rientrano nei limiti imposti dalle regole di gioco, mentre quelle in *B* risultano, in relazione al TimeSpan intrercorso tra un'aggiornamento e l'altro, essere troppo ampie.

Vi sono inoltre alcune situazioni particolari nelle quali è richiesto un controllo sulle coordinate spaziali che un oggetto 3D assume in fase di inizializzazione: la creazione di un *Bullet*, ad esempio deve avvenire in prossimità del Character che l'ha generato. Non è tollerabile, viceversa, che un *Bullet* venga istanziato lontano dal proprio Character, o addirittura posizionato appositamente in prossimità di un Character nemico (Figura 5.5).

Un'altro problema fondamentale legato al Cheating è rappresentato da quelle collisioni che implicano modifiche allo stato di gioco: grazie ai controlli effettuati sullo spazio e sul tempo, è possibile garantire una certo grado di coerenza in quelle che sono le dinamiche di gioco che coinvolgono movimenti e interazioni tra gli elementi di gioco. Come anticipato nella sezione 2.7.2, quando una collisione coinvolge un *Bullet* ed un *Character*, il Player proprietario del *Bullet* segnala l'avvenuta collisione al Player proprietario del *Character* che decrementerà i propri punti HP di conseguenza.

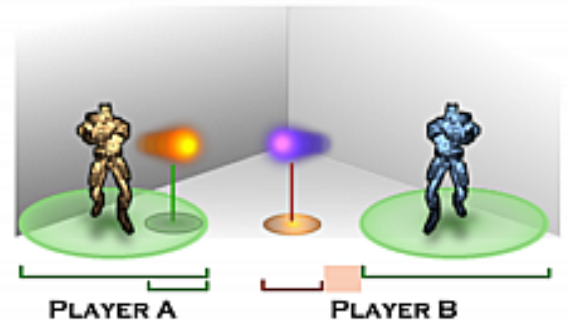

Figura 5.5: Player *A* crea un proprio *Bullet* all'interno dell'area consentita (Cerchio verde, calcolato tenendo conto degli effetti causati dal Lag); Player *B* invece crea il proprio *Bullet* esternamente, e questo viene interpretato come possibile segno di Cheating.

Evidentemente un assetto di questo tipo, se non controllato, si presta ad essere soggetto a problematiche di Cheating: infatti da una parte il Player che ha generato il *Bullet* può notificare collisioni in realtà non avvenute, mentre dall'altra il Player possessore di un *Character* realmente colpito può non operare le dovute modifiche al proprio HP (Figura 5.5).

Una soluzione al primo problema potrebbe essere quella di controllare, sulla ricezione del messaggio Collision, che il *Bullet* coinvolto nella presunta collisione, esista nel proprio Data Layer e che abbia una posizione ed una direzione tale da giustificare una collisione con il *Character* (Figura 5.5).

Per quanto riguarda il secondo problema si può procedere nel seguente modo: quando un *Bullet* appartenente al Player *A* colpisce il *Character* appartenente al *B*, ci si aspetta che *B* decrementi il proprio HP e lo notifichi, automaticamente, a tutti gli altri Player. Parallelamente però, il Player *A* può prevedere tale decremento e modificando di conseguenza il valore del componente HP relativo al Character colpito, all'interno del proprio Data Layer. Se il valore di HP inviato da *B* risulterà maggiore, allora si può sospettare l'utilizzo di un Cheat. E lo si può eventualmente accertare, dal momento che ad ogni incremento del valore di HP deve corrispondere la distruzione di un HealthPack.

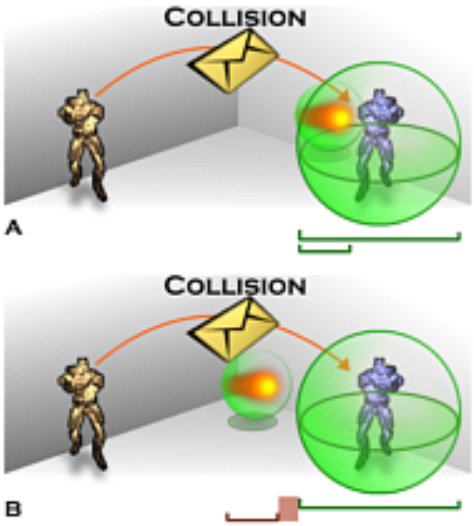

Figura 5.6: Nella figura *B* viene inviato un messaggio Collision quando il *Bullet* in realtà non è avvenuta nessuna collisione

5.2 Ottimizzazione NET Layer

Attualmente le uniche classi che concretizzano le interfacce esposte dal Layer NET sono basate sul protocollo TCP. In quest'ottica si può vagliare la possibilità di una seconda implementazione, basata invece sul protocollo UDP. Un'alternativa da valutare è l'utilizzo di librerie esterne di Game Networking (si veda il capitolo 1.4).
Nel primo caso gli interventi da effettuare a livello strutturale sono minimi, ma è comunque richiesta l'implementazione di una serie di sistemi di controllo, che nel TCP sono gestiti direttamente dal sistema operativo, come il controllo di flusso ed il controllo di congestione, mirati a rendere tale protocollo *Reliable*.
Il secondo caso, invece, implica la sostituzione di uno o più Layer a seconda della libreria utilizzata, e richiede un intervento di *Refactoring* relativamente esteso.

Sotto quest'aspetto può risultare imprescindibile l'introduzione di un si-

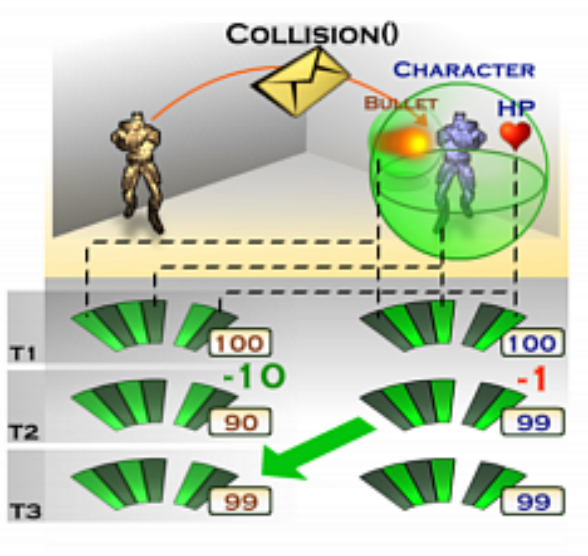

Figura 5.7: In T1 avviene la collsione tra il *Bullet* del Player *A* ed il *Character* del Player *B*; in T2, il Player *A* decrementa il valore di HP appartenente al Player *B* di 10 punti come da specfiche, mentre il Player *B*, cheatando, decrementa il valore del proprio HP di solo un punto. In T3 il Player *A* riceve la notifica di aggiornamento e nota la differenza tra il valore previsto (90) e quello notificato (99).

stema di sincronizzazione temporale, perché se è vero che il modulo SharedComponents limita l'impatto di problemi legati alla latenza sulla consistenza dello stato di gioco, utilizzando il protocollo UDP la sicurezza di ricevere i messaggi nello stesso ordine con il quale sono stati inviati viene meno.

Un'ulteriore possibilità di ottimizzazione da vagliare è rappresentata dall'utilizzo di un sistema di compressione dati: attualmente i *Commands* vengono scritti sullo *Stream* sottoforma di stringhe UTF-8 (ogni carattere occupa da 1 a 4 byte), e l'utilizzo di una libreria di compressione come *Zlib*[3] o *GZip*[4], entrambe supportate nativamente da *.NET*, può ridurre la richiesta di banda. L'efficacia di questo tipo di compressione può però risultare relativamente bassa, dal momento che le stringhe scambiate tra i diversi Peer sono piuttosto corte.

5.3 Gestione utenti e MatchMaking

Nell'attuale implementazione non sono presenti macrostrutture adibite alla gestione del concetto di "*partita*" ed ai sotto-concetti ad esso connessi, come l'inizio e la fine di un match, la vittoria e la sconfitta.
Attualmente infatti non c'è nessun tipo di limitazione alla formazione autonoma di una Membership, né tanto meno un sistema di controllo sul numero di partecipanti: ogni Peer può decidere di entrare in una Membership in qualsiasi momento, semplicemente connettendosi ad uno dei suoi nodi (Figura 5.8).

Il primo problema da affrontare è il *Peer Discovery* e, nella fattispecie l'individuazione di un sistema in grado di notificare ai Peer non ancora connessi l'esistenza di Membership attive, e le informazioni necessarie che consentano di effettuarne il *join*.
Tipicamente questo viene realizzato attraverso un approccio centralizzato,

[3]http://www.zlib.net/
[4]http://www.gzip.org/

Figura 5.8: Campi testuali per la configurazione dll'indirizzo IP e della porta d'ascolto, e per il join in una Membership

caratterizzato da un "*Master server*" (o "*Lobby Server*") che si occupa della gestione delle partite attive.

Supponendo che un Player A voglia creare una partita, una volta specificati indirizzo IP e porta (operazione in futuro automatizzabile), da una parte rimane in ascolto di eventuali Peer che vogliono "*joinare*" nella Membership, e dall'altra riferisce al Lobby Server dell'avvenuta creazione.

Un Player B intenzionato a partecipare ad una partita già esistente, interrogherà il Lobby Server, che invierà di conseguenza lista delle partite attive e le informazioni relative ai singoli partecipanti (Figura 5.9).

Un'ulteriore automazione contestualizzabile nel Lobby Server è rappresentata dal *Matchmaking*, ovvero una selezione delle partite attive più adatte al giocatore, basata sulla valutazione di dati statistici espressi da un sistema di *Ranking*. Quest'ultimo, partita dopo partita, memorizza dati utili ad identificare l'abilità (*Skill*) di ogni singolo giocatore.
Alla base di tale automazione è necessario però che sia presente un sistema di autenticazione in grado di identificare i singoli giocatori.

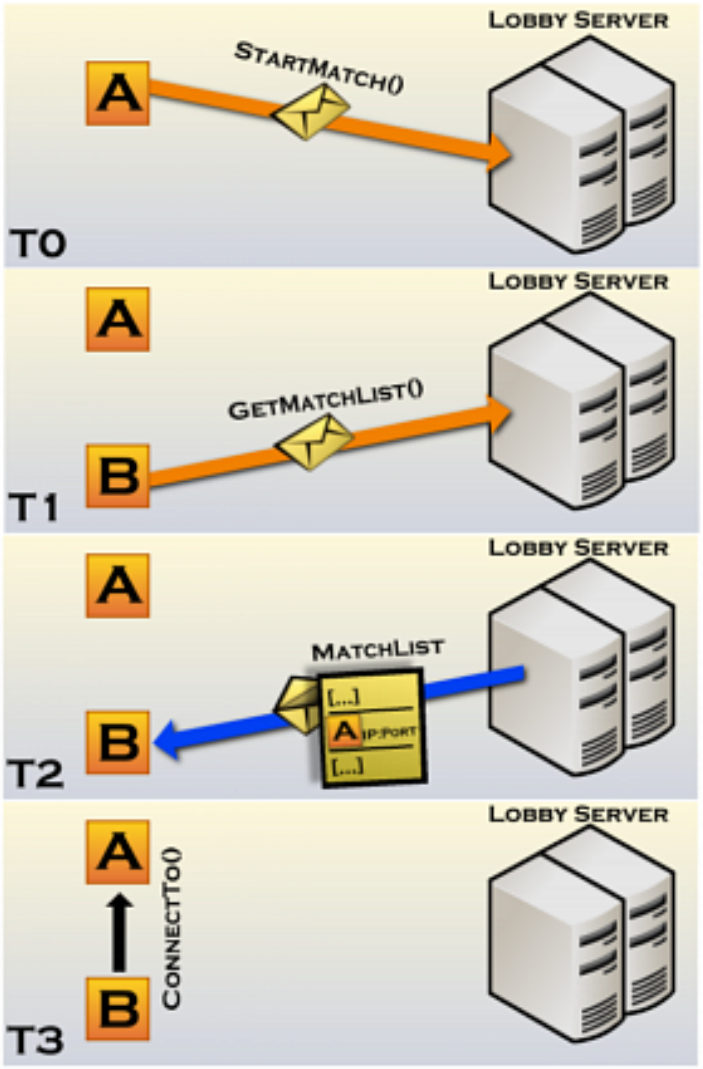

Figura 5.9: Peer Discovery tramite Lobby server

Capitolo 6

Conclusioni

Negli ultimi venti anni, ed in particolare in quest'ultimo decennio, nonostante la crisi economica, l'industria dei videogiochi ha subito una forte crescita, tanto da superare le entrate di altri settori del mercato dell'intrattenimento, come il cinema e l'home video[EntMerch 2008]. Complici diversi fattori: dall'ampliamento del target demografico, alla diffusione di nuove tecnologie e nuovi supporti (si pensi agli SmartPhone) in grado di offrire esperienze videoludiche, anche Multiplayer, in luoghi, tempi e modi impensabili anche solo qualche anno fa.

Parallelamente sono cresciuti i costi e gli investimenti stanziati per lo sviluppo, per il marketing e per le infrastrutture necessarie al supporto Multiplayer.
La realizzazione di un videogioco, intesa come processo produttivo, è oggigiorno un'attività trasversale, caratterizzata da temi, problematiche, ed in ultima analisi, da figure professionali specializzate in ambiti anche notevolmente differenti tra loro.

In questa tesi di progetto si é cercato di affrontare i problemi che stanno dietro all'implementazione di un videogioco Multiplayer Peer-To-Peer, esaminando l'intero processo da un punto di vista generale, e mantenendo il

focus su quelli che sono gli aspetti relativi alla progettazione e allo sviluppo della parte prettamente Software.

Il contesto analizzato è vasto e complesso e non è stato possibile quindi entrare nel dettaglio di ogni singolo aspetto: ad esempio le problematiche legate allo sviluppo di un Engine grafico o di un Engine fisico, grazie all'utilizzo di un Middleware come Unity3D, sono state affrontate solo in maniera marginale.

Anche per quanto riguarda ciò che concerne maggiormente l'argomento di tesi alcune caratteristiche che si possono individuare nella maggior parte dei giochi Multiplayer commerciali, come i sistemi di Anti-Cheating ed i Lobby Server, per questioni di tempo non sono state sviluppate concretamente all'interno del progetto (si veda il capitolo 5 relativo agli sviluppi futuri).

Ciò nonostante, la realizzazione di un videogioco Multiplayer basato su reti Peer-To-Peer, ha permesso di sperimentare con mano quelli che sono i problemi e le difficoltà caratteristiche che emergono durante lo sviluppo di un software videoludico, ed in particolare nel caso in cui quest'ultimo risulti contestualizzato in uno scenario di rete decentralizzato:

In una prima fase sono state analizzate le tecnologie, gli strumenti e le architetture prevalentemente utilizzate nel panorama attuale, al fine di selezionare quelle più adatte all'obiettivo di tesi.

Sono stati valutati quelli che sono i vantaggi e gli svantaggi del modello Peer-To-Peer in contrapposizione al modello Client-Server, e di conseguenza esaminati i sistemi di sincronizzazione utilizzati per mantenere uno stato di gioco condiviso da tutti i giocatori.

Quest'ultimo aspetto rappresenta probabilmente il cuore del problema:

Sostanzialmente, astraendo dalle caratteristiche specifiche di gioco (come le dinamiche di gameplay, la grafica, l'audio e l'interazione utente), il requisito più importante per un MOG è garantire -al medesimo istante di tempo- una rappresentazione dell'ambiente di gioco il più possibile simile per ogni singolo giocatore, nonostante la presenza di fattori quali il Lag ed il Jitter.

Infine, dal punto di vista del design del software è risultata estremamente utile, sia in fase di progettazione che di implementazione, la separazione concettuale tra il contesto logico e gli aspetti prettamente visuali, poiché ha permesso di organizzare, identificare e soprattutto vincolare, gli elementi in grado di apportare informazioni allo stato di gioco, al fine di generalizzare e automatizzare le modalità di aggiornamento di quest'ultimo.

In conclusione l'approccio Peer-To-Peer, rispetto al modello Client-Server, pone una serie di problemi dovuti alla sua natura decentralizzata e "untrusted". Risulta inoltre meno diffuso e relativamente ancora poco sfruttato in ambito videoludico, specialmente nella sua versione "pura". Ma può rappresenta una potenziale alternativa per quelle tipologie di gioco che non richiedono necessariamente la presenza di supporti centralizzati.

Bibliografia

[Abrash 1996] ABRASH M. 1996. *Ramblings in Realtime*

[Bettner 2001] BETTNER P., TERRANO M. 2001. *1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond*, URL: http://zoo.cs.yale.edu/classes/cs538/readings/papers/terrano_1500arch.pdf

[Rhalibi 2005] EL RHALIBI A., MERABTI M., 2005. *Agent-Based Modeling for a Peer-to-Peer MMOG Architecture*

[Due Billing 2006] DUE BILLING O., PETERSEN J., 2006. *Aspects of Peer to Peer game networks*, URL: http://www.p2pgamenetwork.com/downloads/p2pgamenetwork.pdf

[Ferretti 2008] FERRETTI S. 2008. *A Synchronization Protocol For Supporting Peer-to-Peer Multiplayer Online Games in Overlay Networks*, URL: http://sferrett.web.cs.unibo.it/papers/debs08_p083-ferretti.pdf

[Griwodz 2006] GRIWODZ C., HALVORSEN P., 2006. *The Fun of using TCP for an MMORPG*

[Smith 2000] SMITH R.D. 2000, *Synchronizing Distribuited Virtual Worlds*

[Luo 2010] LUO J., CHANG H., 2010, *A scalable architecture for massive Multi-Player Online Games using peer-to-peer overlay*

[Yong-Tae 2008] YONG-TAE H., HONG-SHIK P., 2008, *UDP based P2P game traffic classification with transport layer behaviors*

[Hampel 2006] HAMPEL T., BOPP T., HINN R., 2006, *A Peer-to-Peer Architecture for Massive Multiplayer Online Games*

[Knutsson 2004] KNUTSSON B., LU T., XU W., HOPKINS B., 2004, *Peer-to-Peer Support for Massively Multiplayer Games*

[Gang of Four 2004] GAMME E., RALPH R.H., JOHNSON R., VLISSIDES J., 1995. *Design Patterns Elements of reusable Object Oriented Software*

[Cronin 2002] CRONIN E., FILSTRUP B., KURC A.R., JAMIN S., 2002. *An efficient synchronization mechanism for mirrored game architectures*

[Gautier 1999] GAUTIER L., DIOT C., KUROSE J., 1999. *End-to-end transmission control mechanisms for multiparty interactive applications on the Internet*

[Gregory 2009] GREGORY J., 2009 *Game Engine Architecture*

[Chen 2007] CHEN B. *History Of Multiplayer Games*
URL: http://www.ssagsg.org/LearningSpace/EntertainmentGaming/HistoryMPG.htm

[Ko 2005] KO B.J., RUBENSTEIN D., CALO S., 2005 *Dynamic server selection for large scale interactive online games*

[Hong 2003] HONG E., LEE E.D, KANG K, 2003 *An efficient synchronization mechanism adapting to dynamic network state for networked virtual environment*

[Baughman 2001] BAUGHMAN N., LEVINE B., 2001 *Cheat-proof playout for centralized and distributed online games*

[Paik 2008] PAIK D., YUN C., HWANG J., 2008 *Effective message synchronization methods for multiplayer online games with maps*

[Neumann 2007] NEUMANN C., PRIGENT N., VARVELLO M., 2007 *Challenges in Peer-to-Peer Gaming*

[Mogaki 2007] MOGAKI S., KAMADA M., YONEKURA T., 2007, *Minimization of Latency in Cheat-Proof Real-Time Gaming by Trusting Time-Stamp Servers*

[EntMerch 2008] http://www.entmerch.org/

Ringraziamenti

Desidero innanzitutto ringraziare il Professore Stefano Ferretti per i suoi preziosi suggerimenti e la grande disponibilità dimostratemi.
Inoltre ringrazio sentitamente tutti i professori del corso, in particolare Vania Sordoni e Carla Guerrini, docenti delle materie degli ultimi due esami, forse i più ostici ma allo stesso tempo i più appaganti. Ringrazio tutti i miei compagni di corso con cui ho condiviso lezioni, ore di studio e divertimenti, in particolare Daniele (Viva), Gianmarco (Giammolo), Marco (Bot), Giacomo (Jack), Marco, Nicolò, Matteo, Alessandro (Zacagno), Matteo (Fixxxer), Valentina, Marco (Ketto), Alessio (Torre) e soprattutto mio fratello Riccardo che mi ha sempre aiutato in questi anni di Università e con cui ho condiviso ore e ore di studio.
Infine, ho il desiderio di ringraziare con affetto i miei genitori Carlo e Laura per il sostegno ed il grande aiuto che mi hanno dato sia dal lato economico che dal lato affettivo e per avermi spronato più volte a studiare con più continuità. Ringrazio anche i miei nonni Livio e Renata e tutti i miei parenti.